Introducción a la programación y análisis de texto con R

Clase 2 - Licenciatura en Ingeniería de Medios (UdelaR)

Mag. Elina Gómez (UMAD)

elina.gomez@cienciassociales.edu.uy

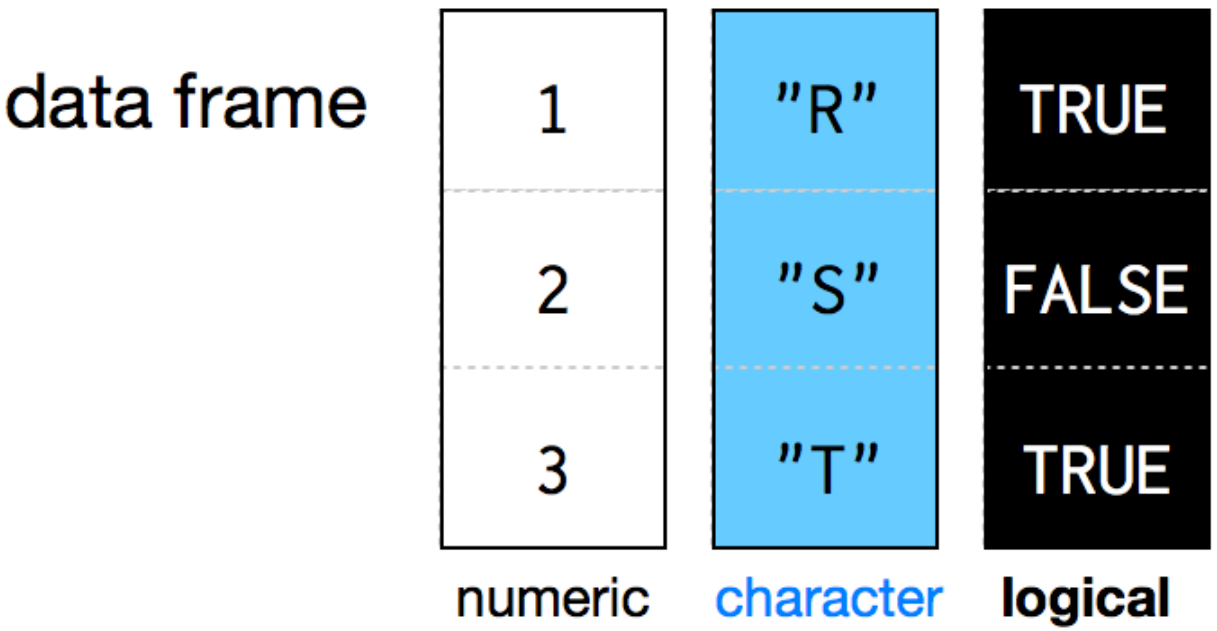
www.elinagomez.com

Dataframes

- Un dataframe o marco de datos (es lo que nos solemos referir como "base de datos").
- Es por lejos la estructura más usada y útil para almacenar y analizar datos
- Formalmente, son la conjunción de dos o más vectores (independientemente de su tipo) en una tabla con dimensiones (Grolemund, 2014)
- Cada vector se transforma en una columna.
- Es una forma de estructurar datos con filas y columnas. Las filas suelen ser las observaciones y las columnas las variables.
- Cada columna .bold[debe] tener la misma longitud (número de observaciones)

Dataframes

Posible estructura de un dataframe o marco de datos



Dataframes

Normalmente los dataframes con los que trabajamos los importamos desde otro formato (lo veremos más adelante), pero también podemos crearlos fácilmente en R.

Supongamos que queremos estructurar los resultados de una encuesta muy corta:

```
# Usamos la función data.frame
encuesta <- data.frame(edad = c(18,24,80,40,76),
                       ideologia = c("Izquierda", "Izquierda", "Derecha",
                                     "Centro", "Derecha"),
                       voto = c("Partido A", "Partido A", "Partido C",
                                "Partido B", "Partido B"))
class(encuesta)
```

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

Dataframes

```
# Con la función head() puedo ver las primeras filas del dataframe
head(encuesta)
```

```
##   edad ideologia      voto
## 1   18 Izquierda Partido A
## 2   24 Izquierda Partido A
## 3   80   Derecha Partido C
## 4   40    Centro Partido B
## 5   76   Derecha Partido B
```

```
# Para la visión extendida usar view() o click en el objeto en el ambiente
```

Dataframes: indexación

De forma similar a los vectores, la indexación [] nos permite acceder a datos dentro de nuestro dataframe. Como los dataframes tienen dos dimensiones (filas y columnas), tenemos que especificar cuáles valores queremos obtener. Para ello la indexación se divide en dos por una coma: antes de la coma nos referimos a las filas, y luego de la coma a las columnas:

Dataframes: indexación

```
# Valor de fila 1 y columna 1
encuesta[1, 1]
```

```
## [1] 18
```

```
# Valor de toda la columna 1 (no fijamos filas entonces nos devuelve todas)
encuesta[, 1]
```

```
## [1] 18 24 80 40 76
```

```
 # Valor de toda la fila 1 (no fijamos columnas entonces nos devuelve todas)
encuesta[1, ]
```

```
##   edad ideologia      voto
## 1   18 Izquierda Partido A
```

Dataframes: indexación

R nos permite utilizar funciones dentro de funciones. Por ejemplo, podemos usar el operador `:` para columnas consecutivas según orden o la función `c()` para referirnos a dos columnas en una indexación:

Dataframes: indexación

```
encuesta[1, 1:2] # Valor de fila 1 y las columnas 1 y 2
```

```
##   edad ideologia
## 1   18 Izquierda
```

```
encuesta[1, c(1,3)] # Valor de fila 1 y las columnas 1 y 3
```

```
##   edad      voto
## 1   18 Partido A
```

Los números negativos nos devuelven lo opuesto:

```
encuesta[1, -3] # Valor de fila 1 y las columnas 1 y 2
```

```
##   edad ideologia
## 1   18 Izquierda
```

Dataframes

Una segunda manera para referirnos a datos dentro un dataframe es el usando el símbolo $. Es la manera más utilizada para refernirnos a una columna de un dataframe, y es muy sencillo de utilizar.

Dataframes

```
# Primero escribimos el nombre del dataframe, seguido por el símbolo $ y
# el nombre de la variable (sin comillas)
encuesta$edad # Esto imprime todos los valores de esa variable
```

```
## [1] 18 24 80 40 76
```

```
# En un dataframe cada variable es un vector y podemos fijarnos su clase
class(encuesta$edad)
```

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

```
mean(encuesta$edad) # Podemos aplicarle funciones (la media en este caso)
```

```
## [1] 47.6
```

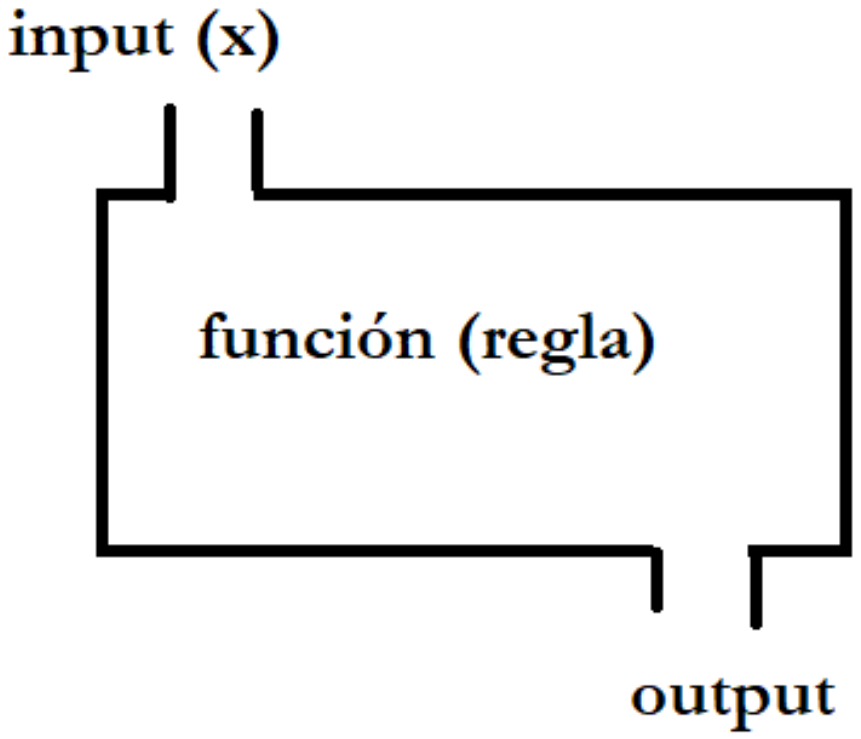
Funciones

- Una función es una serie de instrucciones para realizar una tarea específica. La función suele necesitar un input (generalmente datos) y suele devolver un output (generalmente datos luego de cierta regla)
- Los objetos son cosas, las funciones hacen cosas
- Por ejemplo, en el caso anterior, usamos la función `mean()` para calcular la media de la variable "edad" del dataframe "encuesta"

Funciones

- Usar una función en sencillo: escribimos el nombre de la función, seguido de un paréntesis y dentro los datos a los que le queremos aplicar la función. Pueden ser objetos o directamente valores. Ej. `mean(c(10,20,30))` o `mean(objeto)`
- Dentro de la función se especifican los argumentos, que pueden ser divididos en dos tipos. El primero son los datos a los que se le aplica la función y el resto detalles de cómo se computa la función.

Funciones



Funciones (ejemplo)

```r
# Supongamos que queremos calcular la media de: 12,24,36,48,60
(12 + 24 + 36 + 48 + 60)/5 # Calculo directamente la media
```

```
## [1] 36
```

```r
data_ej <- c(12, 24, 36, 48, 60) # Genero el vector con los 5 números
sum(data_ej) / length(data_ej) # Calculo con dos funciones su media
```

```
## [1] 36
```

```r
mean(data_ej) # Calculo la media directamente con la función mean()
```

```
## [1] 36
```

```r
# También se puede ingresar data directamente en el argumento x
mean(c(12, 24, 36, 48, 60))
```

```
## [1] 36
```

Funciones: R Base

- R viene con un cojunto de funciones
- Las funciones que vienen "por defecto" son las que escribieron los creadores, al igual que en otros softwares no libres.
- La ventaja de R es que cualquiera puede crear nuevas funciones y publicarlas. Colecciones de funciones (generalmente relacionadas) se llaman "paquetes".

Funciones: argumentos

- Las funciones generalmente cuentan con argumentos que van dentro de los paréntesis.
- La mayoría de las funciones cuentan con el argumento "x" que suele ser el objeto al que le pasaremos la función. Al ser la mayoría de las veces el primer argumento, muchas veces no se explicita:

Funciones: argumentos

```
media_fun <- mean(data_ej) # Sin explicitar argumento x

media_fun_x <- mean(x = data_ej) # Explicitando argumento x

identical(media_fun, media_fun_x) # El mismo resultado
```

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

Funciones: argumentos

- Muchas funciones necesitan más de un argumento para funcionar de forma correcta.
- Por ejemplo, pensemos en la función `identical()`: "The safe and reliable way to test two objects for being exactly equal. It returns `TRUE` in this case, `FALSE` in every other case.".
- Por definición, `identical()` necesita dos conjuntos de datos distintos, para testear si son iguales.
- En la documentación `help(identical)` podemos ver que cuenta no solo con el argumento `x`, sino que también con `y`.

Funciones: argumentos

```
# Dos maneras de aplicar la función
identical(media_fun, media_fun_x) # por posición
```

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

```
identical(x = media_fun, y = media_fun_x) # por especificación
```

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

Funciones: argumentos

- A su vez, las funciones muchas veces cuentan con otros argumentos aparte de los datos que usan de insumo. Son detalles de cómo queremos aplicar la regla o el output que recibimos.
- Volvamos a la función `mean()`. Voy a crear un dataframe con la posición que obtuvo Uruguay en los últimos 5 mundiales de futbol masculino

Funciones: argumentos

```r
# Dataframe con el resultado de Urugguay en los últimos 5 mundiales
uru_mundial <- data.frame(year = c(2002, 2006, 2010, 2014, 2018),
                          posicion = c(26, NA, 4, 12, 5))
# Veamos la posición promedio:
mean(uru_mundial$posicion)
```

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

```r
# Como tenemos un dato perdido, la función nos devuelve NA

# Si especificamos el argumento na.rm (no tener en cuenta los datos perdidos):
mean(uru_mundial$posicion, na.rm = TRUE)
```

```
## [1] 11.75
```

Funciones: argumentos por defecto

- Es importante entender que la función `mean()` por defecto tiene el argumento `na.rm = FALSE`. De esta forma, si nosotros solamente le pasamos el argumento `x`, no quitará los datos perdidos.
- Leer la documentación de las funciones es fundamental, y sobretodo prestar atención a los argumentos por defecto.

Funciones: argumentos

```
help(mean)
```

mean {base} → Paquete

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

→ Uso por defecto

Arguments

`x`
: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed f

`trim`
: the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken a

`na.rm`
: a logical value indicating whether `NA` values should be stripped before the computation proceeds.

`...`
: further arguments passed to or from other methods.

Value

Funciones y código

Normalmente al transformar datos utilizamos varias funciones. Esto se puede hacer de varias formas:

- Correr funciones de a una, siempre al mismo objeto
- Ir creando objetos intermedios a medida que aplicamos funciones
- Anidar funciones (se aplicará primero la función más al centro y por último la primera desde la izquierda)
- Pipeline (magrittr)

Funciones y código: ejemplos

Supongamos que queremos aplicar dos funciones a un vector númerico: `mean()` para calcular la media y `round()` para redondear el resultado a 1 dígito después de la coma.

Funciones y código: ejemplos

```
data_ej <- c(12, 21, 33, 41, 27, 23)

# Correr funciones de una, almismo objeto
resultado_A <- mean(data_ej) # Primero estimo la media
resultado_A <- round(resultado_A, digit = 1) # Redondeo

# Correr funciones de una, con objetos intermedios
resultado_B_1 <- mean(data_ej) # Creo un primer objeto con la media
resultado_B_2 <- round(resultado_B_1, digits = 1) # Creo 2do objeto con la media redondeada

# Corro las dos funciones en la misma línea
resultado_C <- round(mean(data_ej), digits = 1)

# Pruebo que los resultados sean iguales:
identical(resultado_A, resultado_B_2)
```

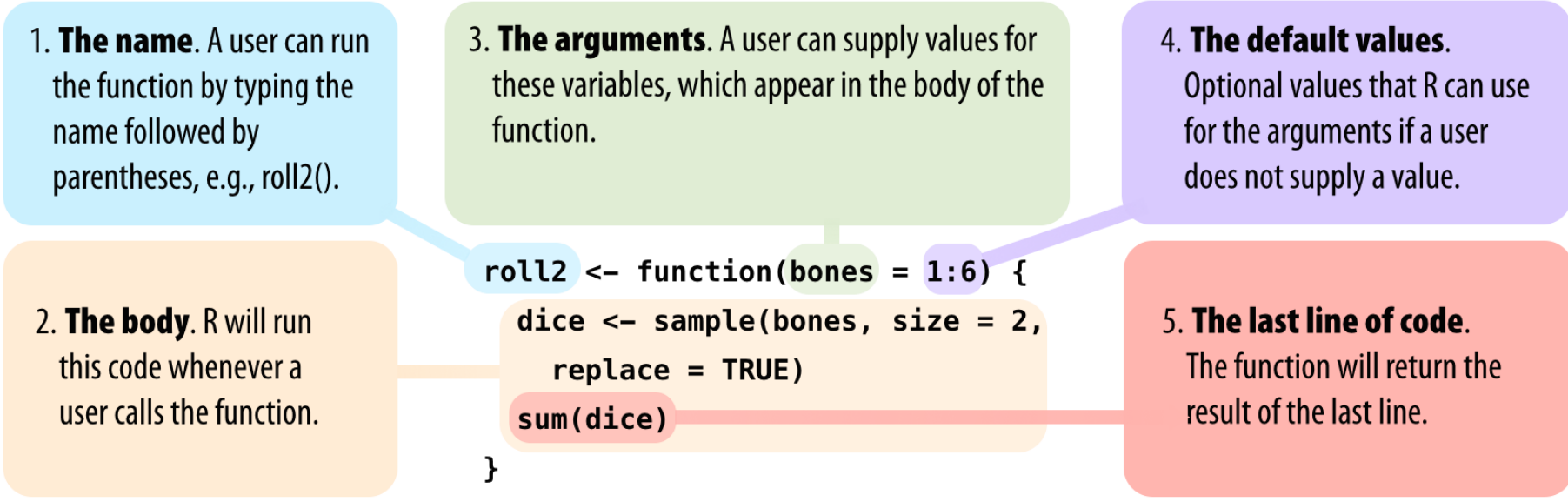
```
## [1] TRUE
```

```
identical(resultado_A, resultado_C)
```

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

Crear funciones

- R permite crear tus propias funciones con facilidad.
- Esto es muy útil para cuando queremos aplicar determinadas líneas de código a varios objetos distintos.



Crear funciones

Supongamos que tenemos varios dataframes con la cantidad de vacas (en miles) y población (en millones) para un puñado de países en 2017, y que queremos agregar una tercera columna que indique la cantidad de vacas per capita. Estos son los datos que queremos transformar:

```
data
```

```
##        pais humanos vacas
## 1   Uruguay     3.4 11800
## 2 Argentina    43.8 53500
## 3    Brasil   209.5 22600
## 4    Mexico   128.6 16500
```

```
data_2
```

```
##            pais humanos vacas
## 1       Uruguay     3.4 11800
## 2 Nueva Zelanda     4.5  9900
## 3     Australia    43.8 53500
## 4         Japón   126.3  3800
```

Crear funciones

```
# Ahora quiero calcular la cantidad de vacas per capita.
# Podría hacer:

data$vacas_pc <- (data$vacas / 1000) / data$humanos
data
```

```
##        pais humanos vacas  vacas_pc
## 1   Uruguay     3.4 11800 3.4705882
## 2 Argentina    43.8 53500 1.2214612
## 3    Brasil   209.5 22600 0.1078759
## 4    Mexico   128.6 16500 0.1283048
```

```
# Ahora me gustaría tener una tabla un poco más prolija:
# números redondeados y agrego "per"

data$vacas_pc <- round(data$vacas_pc, digits = 1)
data$vacas_pc <- paste(data$vacas_pc, "per", sep = " ")

print(data)
```

```
##        pais humanos vacas vacas_pc
## 1   Uruguay     3.4 11800  3.5 per
## 2 Argentina    43.8 53500  1.2 per
## 3    Brasil   209.5 22600  0.1 per
## 4    Mexico   128.6 16500  0.1 per
```

Crear funciones

Como este paso lo voy a repetir muchas veces, creo una función que llamo `calc_vacas()` que realice todos estos cambios: .codefont[

```
calc_vacas <- function(x, y){
  vacas_pc <- (x / 1000) / y   # Calculo la proporción de x / 1000 sobre y
  vacas_pc_1 <- round(vacas_pc, digits = 2) # Redondeo
  vacas_pc_2 <- paste(vacas_pc_1, "per", sep = " ")
 return(vacas_pc_2)
}

# Aplico la función a data_2
data_2$vacas_pc <- calc_vacas(x = data_2$vacas, y = data_2$humanos)
data_2
```

```
##            pais humanos vacas vacas_pc
## 1       Uruguay     3.4 11800 3.47 per
## 2 Nueva Zelanda     4.5  9900  2.2 per
## 3     Australia    43.8 53500 1.22 per
## 4         Japón   126.3  3800 0.03 per
```

Crear funciones

Copiando y pegando código

```
data$vacas_pc <- (data$vacas / 1000) / data$humanos
data$vacas_pc <- round(data$vacas_pc, digits = 1)
data$vacas_pc <- paste(data$vacas_pc, "per", sep = " ")

data_2$vacas_pc <- (data_2$vacas / 1000) / data_2$humanos
data_2$vacas_pc <- round(data_2$vacas_pc, digits = 1)
data_2$vacas_pc <- paste(data_2$vacas_pc, "per", sep = " ")
```

Con una función

```
data$vacas_pc <- calc_vacas(x = data$vacas, y = data$humanos)

data_2$vacas_pc <- calc_vacas(x = data_2$vacas, y = data_2$humanos)
```

Errores y advertencias

- Cuando utilizamos funciones podemos encontrarnos con errores (errors) y advertencias (warnings). La principal diferencia entre ellas es que el error implica que la función no se pudo aplicar, mientras que en la advertencia la función fue aplicada pero que algo no funcionó como esperado. También es posible ver mensajes (messages) que simplemente informan algo sobre la función

```
vector_ej <- rnorm(n = 10, mean = 10, sd = 5) # Creo valores aleatorios

mean(Vector_ej) # Aplico función para obtener la media
```

```
## Error in eval(expr, envir, enclos): objeto 'Vector_ej' no encontrado
```

¿A qué se debe este error?

Errores y advertencias

```r
vector_ej <- rnorm(n = 10, mean = 10, sd = 5) # Creo valores aleatorios

mean(vector_ej) # Aplico función para obtener la media
```

```
## [1] 13.1754
```

Errores y advertencias

En el caso anterior, es sencillo entender porque la función no corrió. Sin embargo, muchas veces podemos no entender que es lo que salió mal y copiar y pegar el error en un buscador de internet es un buen primer paso.

```
vector_1 <- c("10", "35%", "35", "50") # Vector de caracteres que contiene números
vector_1
```

```
## [1] "10"  "35%" "35"  "50"
```

```
vector_2 <- as.numeric(vector_1) # Transformo a vector número
```

```
## Warning: NAs introducidos por coerción
```

```
vector_2 # Los valores que además del número tenían (%) no pueden pasarse a números
```

```
## [1] 10 NA 35 50
```

```
vector_1 <- gsub("%", "", vector_1) # Quito los % del vector original
vector_1 # Sin valores perdidos
```

```
## [1] "10" "35" "35" "50"
```

Errores y advertencias

```
vector_2 <- as.numeric(vector_1) # Transformo a vector número
vector_2 # Los valores que además del número tenían (%) no pueden pasarse a números
```

```
## [1] 10 35 35 50
```

Paquetes

- Los paquetes son conjuntos de funciones, documentación de ayuda y a veces datos
- El conjunto de funciones que vienen por defecto en R se le denomina "Base". Por ej. las funciones que hemos utilizado hasta ahora -`mean()`, `identical()`, `help()`- están dentro del R Base.
- El repositorio principal donde se alojan los paquetes de R se llama Comprehensive R Archive Network CRAN. Hay más de 10.000 paquetes alojados en CRAN. Aquí hay dos listas con algunos de los paquetes más útiles: support.rstudio y towardsdatascience

Paquetes

- Los paquetes alojados en CRAN son testados antes de ser publicados. Por eso es recomendable tener cuidado con utilizar paquetes que no han sido publicados allí aún.
- Los paquetes normalmente están relacionados a alguna temática, por ejemplo existen paquetes específicos para manipulación de datos dplyr, visualización de datos ggplot2 o importar y exportar datos readr. Dado que R es libre, cualquier persona puede crear y publicar un paquete (publicarlo en CRAN requiere ciertos procesos igualmente), esto facilita que haya paquetes muy específicos para ciertas tareas que pueden ser de gran utilidad. Por ejemplo, existen paquetes para conectarse con la API de Twitter rtweet, para utilizar los datos de Gapminder o para analizar datos electorales y de opinión pública en Uruguay opuy

Paquetes: instalación

- Cuando abrimos R solamente tenemos cargado por defecto el paquete Base.
- La primera vez que queremos utilizar un paquete tenemos que descargarlo con el siguiente comando:

```
# Para descargar paquetes de CRAN, utilizamos la siguiente función:
install.packages("nombre_del_paquete")
install.packages("dplyr") # Ejemplo

# Existen otros paquetes no alojados en CRAN, que se instalan con
# el paquete "devtools"
```

Paquetes: cargar

- Luego de instalarlo (una sola vez por versión de R), tenemos que cargar el paquete utilizando la función `library()`
- Esto lo debemos realizar en cada sesión de R que vayamos a utilizar (cada vez que abrimos R).

```
library(nombre_del_paquete)
library(dplyr) # Ejemplo
```

Paquetes

En ocasiones los paquetes tienen funciones que se llaman igual, lo que puede llevar a errores. En estos casos -una vez instalado el paquete- podemos usar una función sin cargar el paquete de la siguiente manera:

```
df_1 <- data.frame(col_a = c(1, 2, 3),
                   col_b = c(2, 4, 6))

df_2 <- data.frame(col_a = c(1, 2, 3),
                   col_c = c(-2, 1, 6))

# Uso la función rbind.fill() que a diferencia de rbind() del Base,
# permite unir dataframes con columnas que no matchean de forma exacta,
# agregando NAs

# No cargo todo el paquete plyr sino que llamo la función específica usando ::
df_3 <- plyr::rbind.fill(df_1, df_2)
```

Operadores aritméticos

Operador	Descripcion	Ejemplo	Resultado
+	Suma	2+2	4
-	Resta	2-2	0
*	Multiplicación	2*2	4
%	División	2%2	1
^	Potencia	2^3	8

Operadores relacionales

Operador	Descripcion	Ejemplo	Resultado
$<$	Menor a	$4 < 4$	FALSE
$>$	Mayor a	$6 > 2$	TRUE
$<=$	Menor o igual a	$4 <= 4$	TRUE
$>=$	Mayor o igual a	$6 >= 2$	TRUE
%in%	Está incluido dentro de	2 %in% c(0, 1, 2)	TRUE
$==$	Igual a	hola == hello	FALSE
$!=$	Distinto a	hola != hello	TRUE

Operadores booleanos

Operador	Prueba
a & b	a y b son verdaderos
a | b	al menos una de a o b son verdaderas
!a	a no es verdadera
isTrue(a)	a es verdadera