2. Nociones básicas de R

Notas varias, opiniones y observaciones - Curso de Luis Alejandro Carrera

Patricio Moracho

23/1/2020

```
############################
## Nociones básicas de R ##
############################

## 1. Operadores
## :: ::: Permiten accesar a variables en un espacio de nombres
##      Son útiles para llamar a funciones sin cargar librerías en el espacio de trabajo.
           dplyr::filter()     #Ejecuta la función filter() de dplyr
           tidyr::pivot_wider()    #Ejecuta la función pivot_wider() de tidyr
```

Atención con los triples dobles puntos `:::`, ya que acceden a funciones y variables que el autor del paquete no exportó específicamente, vale decir, que decidió ocultar por alguna razón. Para un usuario común hay pocas chances que necesite usar este operador

```
## $            Extracción y creación de componentes
## [] [[]]      Indexación
```

No está mal comentar aquí, que también podemos pedir ayuda sobre cada operador mediante ``` ``?$`` ``` o ``` ``?[[`` ```. Te comento que los operadores `$` y `[]` (o `[[]]`) son conceptualente equivalentes, por lo que te diría que no separes el sentido en extración o indexación como dos cosas distintas, ya que la idea es la misma, son operadores de "extración" o de "reemplazo" de elementos puntuales en una colección de datos.

```
## ^            Exponenciación (de derecha a izquierda)
## - +      Suma y resta
## :            Operador de secuencias
## %any%        Operadores especiales como %% (resto) y %/% (entero)
## * /      Multiplicación y división
## <            Menor que
## >            Mayor que
## <=           Menor o igual que
## >=           Mayor o igual que
## ==           Igual que
## !=           Diferente a
## ! -      Negación
```

Sin comentarios

```
## &    |       "y" "o" para comparaciones elemento a elemento
## && ||        "y" "o" utilizado en condicionales if
```

Acá hay un tema interesante que es importante dejar bien claro, y es la diferencia entre los operadores dobles y simples. El doble `&&` o `||`, no es que unos son para usar en un `if` y los otros no, o sí, pero con una aclaración importante, Veamos este caso:

```
c(TRUE, TRUE) & c(FALSE, TRUE)
```

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

En este caso el `and` lógico simple, se hace sobre cada elemento del vector y correctamente nos entrega un vector de lógicos del tamaño de los vectores originales. En el segundo caso:

```
c(TRUE, TRUE) && c(FALSE, TRUE)
```

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

El doble `and` solo compara el primer elemento de cada vector, vale decir que hace algo así: `c(TRUE, TRUE)[1] && c(FALSE, TRUE)[1]`, con lo cual siempre el retorno será un único elemento lógico, una salida que es consistentente con la forma en que trabaja un bloque `if`. Es fácil olvidar que en R no existen los datos escalares, todo datos es una colección de al menos un elemento, por lo cual es simple pensar que si tengo un `if` uso el operador doble, pero que pasa si comparamos dos vectores con múltiples elementos, ¿es la comparación del primer elemento la que efectivamente tiene que controlar el bloque `if`? posiblemente no y en estos casos corresponda más bien algo así:

```
if (any(c(TRUE, TRUE) && c(FALSE, TRUE))) {}

if (all(c(TRUE, TRUE) && c(FALSE, TRUE))) {}

## ~            Operador de fórmulas (Ejemplo: lm(y~x1+x2))
## -> ->>       Asignación y superasignación a la derecha
## <- <<-       Asignación y superasignación a la izquierda
## =            Asignación a la izquierda

## 2. Asignación
## En R, contrario a otros lenguajes de programación, se prefiere el operador "<-"
## en lugar del signo igual "=" para realizar asignaciones. Aunque en general el
## comportamiento de ambos operadores es el mismo en el entorno de trabajo, "<-"
## puede emplearse también como "->" de forma tal que sin importar la dirección
## de la flecha, se creará una variable con un valor específico.

## Ejecutar
5 -> w;  w  #5
x <- 6;  x  #6
y = 7;   y  #7
8 = z;   z  #Error in 8 = z : lado izquierdo de la asignación inválida (do_set)
```

Bien explicado con el último ejemplo. No sé si vas a hablar de la asignación "global" la de `<<-`, dónde podemos acceder a un valor fuera del entorno actual de una función, es decir una variable cuya existencia está fuera de la función, ej:

```
f <- function() { x<<-5 }

x <- 0

x
```

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

```
f()
x
```

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

```
## Nota
## La escritura del tipo x <- 6; x es una forma compacta para
## x <- 6
## x
## Más adelante se muestra otra forma de escritura compacta
```

También es posible usar la función identidad `()` (ya ví que lo presentas más abajo)

```
x <- 0
x
```

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

```
(x <- 6)
```

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

```
x
```

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

```
## Por otra parte, cuando empleamos "=", las variables no necesariamente se crean
## en el espacio de trabajo, en particular cuando invocamos funciones:
median(test=1:15)   # Error in is.factor(x): el argumento "x" está ausente, sin valor por omisión
test            # Error: objeto 'test' no encontrado

## Ejecutar:
median(test <- 1:15)    # 8
test                # [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## La variable test está definida en el entorno de trabajo
```

Este ejemplo que estás dando, es una patrón no habitual y creo que potencialmente confuso, en mi opiniñón habría que evitarlo, aunque efectivamente el ejemplo es correcto y grafica muy bien lo que estás diciendo. Tal vez lo presentaría de esta forma:

```
## Esto (la forma recomendable)
test <- 1:15
median(test)
```

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

```r
## Es equivalente a esta forma más compacta
median(test <- 1:15)
```

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

```r
## Pero que de ninguna manera funciona mediante el =
median(test=1:15)
```

```
## Error in is.factor(x): el argumento "x" está ausente, sin valor por omisión
```

Acá empieza algo medular, me gustaría brindarte algunos conceptos más generales sobre R y los datos, por las dudas, que alguno te sea útil.

- R a diferencia del resto de los lenguajes no tiene datos escalares, es decir, no existe un número o una cadena aislada, cualquier dato es una colección de `n` elementos, pudiendo ser `n=0` un vector vacio, `n=1` un vector de un solo elemento o que `n` tome cualquier valor de elementos mientras exista memoria disponible.
- Los tipos de dato básicos son: numericos, enteros, caracteres, logicos, complejos. (hay otro más, el `Raw`, pero no vale la pena mencionarlo)
- Las colecciones u objetos más básicos y habituales son `vector` y `matrix` (es un vector con más de una dimension), luego nos aparecen las `listas`, los dos primeros objetos solo pueden albergar un solo tipo de dato, las `listas` es el único objeto que es capaz de almacenar más de un tipo de dato, por supesto tenemos el `data.frame` o el `tibble` pero hay que señalar que son especializaciones del objeto `lista`.
- La coerción que mencionas más abajo, es un efecto que se produce automáticamente cuando combinas distintos tipos de datos en un objeto que no lo permite, la converión que se produce siempre tiende al valor que pueda englobar toda la colección de datos, si tiene un entero y un numérico la coerción irá hacia el numerico, un numerico y una cadena, la coerción llevara todo a una cadena.

```r
## 3. La función c() y la creación de vectores
## En R se emplea de forma regular la función c() que significa "combine"
## Esta función permite crear vectores de números, caracteres, variables y
## observaciones dependiendo del contexto en el que sea utilizada. Los argumentos en
## c() son coercionados a un tipo común para generar un valor de retorno.
## Los vectores son la estructura de datos básica en R y pueden ser de tipo lógico, numérico
## y caracter. Para verificar el tipo y clase de un vector empleamos las funciones typeof()# y class().
## typeof() nos regresa la forma en que un determinado objeto está guardado en el nivel interno del
## lenguaje. Por su parte, class() se refiere al tipo "abstracto" o "genérico" de un objeto.
## typeof(). Si deseamos saber si un vector pertenece a un tipo específico, empleamos is.
## seguida del atributo de interés.

## Ejecutar:
num_vec <- c(1:3, 4:9, 10:15)
typeof(num_vec); class(num_vec) # "numeric"
is.numeric(num_vec)         # TRUE
is.character(num_vec)       # FALSE
is.logical(num_vec)         # FALSE

char_vec <- c(1:10, "perro", "niño")# El vector es coercionado
typeof(char_vec); class(char_vec)   # "character"
is.numeric(num_vec)         # FALSE
is.character(num_vec)       # TRUE
is.logical(num_vec)         # FALSE
```

```
logi_vec <- c(T,T,F,F,T)
typeof(logi_vec); class(logi_vec)   # "logical"
is.numeric(logi_vec)            # FALSE
is.character(logi_vec)      # FALSE
is.logical(logi_vec)            # FALSE
```

`class()` también sirve para definir una nueva clase de un objeto por ejemplo:

```
x<-1:5
class(x) <- "miclase"
class(x)
```

```
## [1] "miclase"
```

```
typeof(x)
```

```
## [1] "integer"
```

Otro ejemplo del uso de ambas funciones:

```
x <- 1:10
y <- x/5 + rnorm(10)

class(y)    # numeric
```

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

```
typeof(y)   # double, es una especialización de un nuemric
```

```
## [1] "double"
```

```
g <- lm(y ~ x)
class(g)    # g es un objeto de la clase lm
```

```
## [1] "lm"
```

```
typeof(g)   # Sin embargo no deja de ser tambien una list
```

```
## [1] "list"
```

Y como se puede ver, `typeof()` te da el dato de más bajo nivel, mientras `class()` te puede dar realmente el nombre de la clase de cualquier objeto. Cuidado con todo esto, por que involucra el tema de la programación orientada a objetos, en la que imagino, no vas a entrar. A la audiencia a la que apuntas, con que menciones que una funciona apunta más alto que la otra, creo que alcanza,

```
## La coerción de vectores puede hacerse explícitamente empleando as. seguido del atributo de interés

## Ejecutar:
char_vec_num <- as.numeric(char_vec)    # Notificación de NA introducidos por coercion
char_vec_num                    #  [1]  1  2  3  4  5  6  7  8  9 10 NA NA
```

Este problemas ocurre cuando convertimos valores más genericos a más especificos y algún dato no puede conertirse, salvo el caso de un numerico real a un entero en dónde se truncan decimales.

```
as.integer(1.5)
```

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

```
## Para crear el nuevo vector y visualizar de inmediato su resultado, podemos emplear paréntesis
## de apertura y de cierre de la siguiente manera:
(char_vec_num <- as.numeric(char_vec))

## Ejecutar:
(logi_vec_char <- as.character(logi_vec))   #  "TRUE"  "TRUE"  "FALSE" "FALSE" "TRUE"
(logi_vec_num <- as.numeric(logi_vec))  # 1 1 0 0 1

## A notar:
## as.numeric() coerciona en NA las cadenas de caracteres en un vector que también contiene números
## as.numeric() transforma un vector lógico en un vector de 0 (para F) y 1 (para T)
```

`as.numeric()` tranforma en numérico cualquier cadena que puede converirse en un número, el resto las considera `NA`

```
as.numeric(c("1", "1.34", "-1.7e3", "no es un numero"))
```

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

```
## [1]     1.00     1.34 -1700.00       NA
```

```
## 4. Listas y data.frames
## Las listas pueden contener elementos de distinto tipo incluyendo otras listas.
## Por ese motivo, se les denomina "vectores recursivos". Por sus características, una lista no
## coerciona a los elementos que pertenecen a ella como ocurre con la función c().

## Ejecutar:
list_ej <- list(c(1,4,3), 2, "perro", list=(1:4))
str(list_ej)        # List of 3 $ : num [1:3] 1 4 3 $ : num 2 $ : chr "perro" $ list: int [1:4] 1 2 3 4
typeof(list_ej)     # "list"

vec_ej <- c(c(1,4,3), 2, "perro", list=(1:4))
str(vec_ej)         # Named chr [1:9] "1" "4" "3" "2" "perro" "1" "2" "3" "4"
typeof(vec_ej)      # "character"
```

Esto, porque la lista es el primero de los tipos de contenedores que soporta multiples tipos de datos.

```
## Las listas suelen emplearse en la construcción de funciones y su uso puede observarse
## con la función lm(). Al revisar la documentación de esta función, puede leerse lo siguiente:
## "An object of class "lm" is a list containing at least the following components:"
?lm
```

Las listas son la base de cualquier objeto más o menos complejo.

```
## Una data.frame es un conjunto de listas que pueden diferir en el número de vectores (columnas)
## pero no de elementos (renglones). Puede observarse que la estructura básica de una data.frame
## es justamente una lista como revela la función typeof()

df_ej_uno <- data.frame(a=c(1:3), b=c("perro", "gato", "ratón"), stringsAsFactors=F)
typeof(df_ej_uno)       # "list"
class(df_ej_uno)        # "data.frame"
```

Una aclaración, estrictamente hablando, el `data,frame` no es un cojunto de listas, es más bien una lista de vectores, cada columna es un elemento de la lista, y es a su vez un vector de un determinado tipo. Nunca una columna puede tener distintos tipos de dato.

```
## Si los números de renglones no coinciden entre variables, se mostrará un error
df_ej_dos <- data.frame(a=c(1:10), b=letters[1:9])
# Error in data.frame(a = c(1:10), b = letters[1:9]):
#  arguments imply differing number of rows: 10, 9

## 5. Atributos, nombres y factores
## Los atributos permiten alojar metadatos de los objetos mediante una lista de nombres. Se puede
## acceder a ellos mediante la función attributes().
attributes(df_ej_uno)
 $names
 [1] "a" "b"

 $class
 [1] "data.frame"

 $row.names
 [1] 1 2 3

## En este caso, uno de los atributos de una data.frame son los nombres (names) de las columnas
## (variables). Los nombres de los vectores columna se asignan al crear una data.frame (como en el
## caso de df_ej_uno) y pueden reasignarse directamente en el objeto, es decir, sin crear una
## copia modificada del mismo.
names(df_ej_uno) <- c("uno", "dos")
attributes(df_ej_uno)$names [1] "uno" "dos"
```

Èn este caso, creo que es preferible explicar `colnames()` por sobre `names()`, esta última solo aplica a un `data.frame` si se la quieres usar en una matriz ya no te sirve, sin embargo `colnames()` además de tener un nombre más explicativo funciona en los dos objetos antes mencionados.

```
## (Dado que la función attributes() regresa una lista, podemos acceder a sus elementos
## empleando el operador "$" seguido del atributo que nos interesa verificar).
x <- attributes(df_ej_uno)
typeof(x)       # "list"

## NOTA: La función "names <-" se abordará con mayor detalle en los temas de manipulación de datos.

## Un uso importante de los atributos es la definición de factores. Los factores son vectores que
## contienen valores predefinidos y son empleados para alojar datos categóricos.

set.seed(140)   # set.seed permite que el ejemplo sea replicable
num_samp <- rep(c(1,2),10) ; num_samp <- sample(num_samp, rep=F)
```

```
fac_ej <- factor(num_samp)

[1] 2 1 2 1 2 2 1 1 2 2 2 2 1 1 2 1 1 1 2 1
Levels: 1 2     # Levels nos muestra el conjunto de valores permitidos para el vector

## Para asignar una jerarquía a los niveles, podems indicar el argumento "levels" directamente
## en la función factor().
(fac_ej <- factor(fac_ej, levels=c("2","1")))   # Levels: 2 1

## NOTA: Los factores se verán con mayor profundidad en los temas relativos a manipulación de datos
```

En el caso de los factores, no sé si lo mencionarás más adelante, cabe la posibilidad de definir un factor ordinal, con un determinado orden, por ejemplo:

```
set.seed(2019)
x <- sample(LETTERS, 100, replace = TRUE)
x.factor <- factor(x)
x.factor # factor común
```

```
##   [1] Y J E M Q X W L S Z I M Z N Q H S W N H C T O E C D E W Y A N G F F Q P Z
##  [38] W F E Q F O N L U Z N V Q P B Q M A I L O I P S Q H J W W B P M F O V Y M
##  [75] J H G V H H L C Y M U R B Y H G C L W N L L G Y V K
## Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
x.factor_ordered <- factor(x, ordered = TRUE)
x.factor_ordered # factor ordenado
```

```
##   [1] Y J E M Q X W L S Z I M Z N Q H S W N H C T O E C D E W Y A N G F F Q P Z
##  [38] W F E Q F O N L U Z N V Q P B Q M A I L O I P S Q H J W W B P M F O V Y M
##  [75] J H G V H H L C Y M U R B Y H G C L W N L L G Y V K
## 26 Levels: A < B < C < D < E < F < G < H < I < J < K < L < M < N < O < ... < Z
```

Notar como se muestran los `Levels` en cada caso. Los factores ordenados son muy utiles cuando efectivamente la variable tiene un orden que conviene sea respetado en estadísticos y/o gráficas. Por ejemplo:

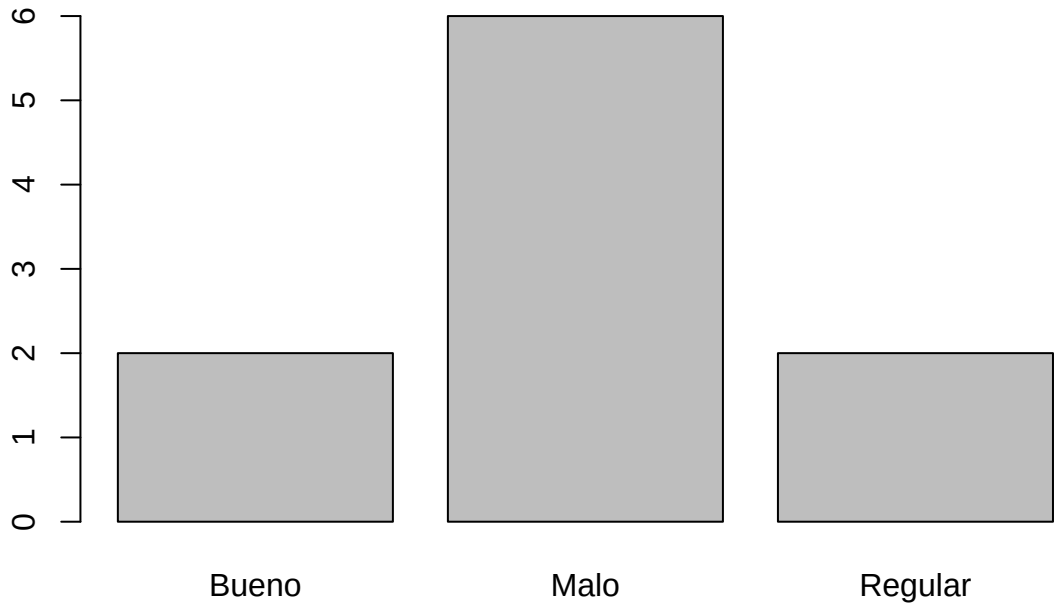
```
set.seed(2019)
niveles <- c("Malo", "Regular", "Bueno")
x <- sample(niveles, 10, replace = TRUE)
x.factor <- factor(x)
# factor común

x.factor_ordered <- factor(x, levels=niveles, ordered = TRUE)
x.factor_ordered # factor ordenado
```

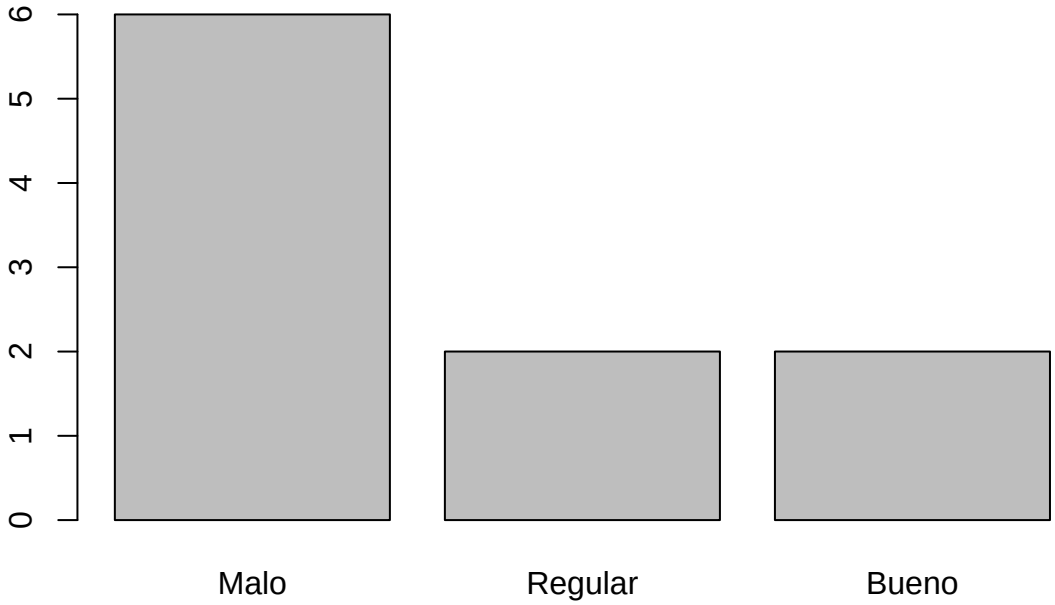
```
##  [1] Malo    Malo    Regular Malo    Malo    Malo    Bueno   Malo    Bueno
## [10] Regular
## Levels: Malo < Regular < Bueno
```

Al definir un factor ordenado, debemos si o sí indicar el orden a respetar con el parámetro 'levels' . Si tuvieramos que graficar los dos vectores en un gráfico de barras de R base, vemos que los niveles se ajustan en el caso del vector ordenado.

```r
barplot(table(x.factor))
```




```r
barplot(table(x.factor_ordered))
```


0
1
2
3
4
5
6
Malo
Regular
Bueno