II. Nociones básicas de R como lenguaje de programación

Abril 2020

Nota preliminar

En este módulo se presentan los elementos básicos para iniciar a trabajar con R como lenguaje de programación. Se espera que los contenidos aquí presentados permitan al usuario desarrollar la intuición necesaria para abordar con éxito la manipulación de data.frames, que constituye la razón de ser de este curso. Por ello, se abordarán entre otros temas: el uso de operadores, la creación de vectores de distintos tipos, la coerción de vectores, el funcionamiento de las listas y las características de las data.frames.

1. Operadores

De forma general, podemos decir que un operador es un símbolo que tiene una función predefinida y que le indica al lenguaje que debe realizar una determinada tarea. En **R**, los operadores más importantes son los siguientes:

Operador	Descripción
`::`	Permite accesar a variables en un espacio de nombres. Se utiliza para llamar a funciones de librerías que no se han cargado en el espacio de trabajo y/o para invocar funciones "enmascaradas" (tema tratado en el M1)
`$`, `[ ]`, `[[ ]]`	Operadores de "extracción" o "reemplazo" de elementos en una colección de datos
`^`	Exponenciación (de izquierda a derecha)
`-`, `+`	Resta y suma
`:`	Operador para la creación de secuencias
`%any%`	Operadores especiales como %% (resto de una división) y %/% entero (parte entera de una división)

Operador	Descripción
*, /	Multiplicación y división
<, >, <=, >=	Menor que, mayor que, menor o igual que, mayor o igual que
==, !=	Igual que, diferente a
!, -	Negación lógica y negación
&, |	"y", "o" para comparaciones elemento a elemento
~	Operador de fórmulas ((Ejemplo: lm(y~x1+x2))
->, ->>	Asignación y superasignación a la derecha
<-, <<-	Asignación y superasignación a la izquierda
=	Asignación a la izquierda

> ***NOTA PM:*** Hay un concepto que me parece importante que manejes y trasmitas, es algo particular de **R** y para nada común e otros lnguajes. Lo habitual es que los operadores sean simplemente estructuras léxicas del lenguaje, en **R** en realidad son funciones.

Nota: Para obtener ayuda sobre operadores, podemos emplear ? seguida del símbolo entre comillas sencillas. Por ejemplo: ?+, ?:, ?=, nos remitirá a la información de operadores aritméticos, al operador de secuencias y los operadores de asignación

El operador `::`, como vimos en el primer módulo, se utiliza para llamar a funciones de librerías que no se han cargado en el espacio de trabajo y/o para invocar funciones "enmascaradas".

`$, [ ], [[ ]]` se conocen genéricamente como "operadores de subconjuntos". El primero se utiliza regularmente en listas y data.frames; el segundo en vectores, matrices y data.frames y el tercero fundamentalmente en listas. Más adelante se verá su uso en vectores y listas y en el módulo relativo a tratamiento de datos, en data.frames.

Los operadores `^, -, +, %%, %/%, * y /` son aritméticos y su uso es directo. Para observar cómo funcionan, copiemos, peguemos y ejecutemos el siguiente código:

```
2^3
4-2
10+7
10%%3
7%/%3
2*17
56/19
```

El operador `:` se emplea para crear secuencias número a número.

```
1:5
-20:15
0:11
```

Los operadores <, >, <=, >=, == y != se denominan "lógicos" o "booleanos" porque nos arrojan como resultado `TRUE` si se cumple la condición o comparación o `FALSE` en caso contrario. Ejecutemos las siguientes comparaciones y observemos los resultados:

```
2<3
3>1
11 <= 20
15 >= 13
"perro" == "gato"
"Perro" != "perro"
```

La última comparación quizás pueda llamar la atención porque parece tratarse de cadenas de caracteres con el mismo valor ("perro"), por lo cual el resultado de la comparación debería ser `FALSE` (es decir, que no son diferentes). Sin embargo, **R**, al igual que la mayoría de los lenguajes de programación **distingue entre mayúsculas y minúsculas** (*"case sensitive"*), por lo cual una **P** es tratada de forma diferente que una **p**. Esto es de importancia porque muchos de los errores asociados al uso de este lenguaje tiene que ver con el uso equivocado de mayúsculas y minúsculas, en particular cuando creamos variables, bases de datos o empleamos alguna función.

El operador `!` permite "negar" aquello que se encuentra a su derecha. Por tanto, si una comparación nos arroja `TRUE` el resultado será `FALSE` y viceversa. Este operador es de mucha utilidad en el manejo de data.frames, como veremos en módulos posteriores.

```
5>3   # TRUE
!5>3  # FALSE
!("perro"!="Perro") # FALSE
```

Los operadores `&` y `|` se tratarán más adelante cuando veamos de forma breve vectores y data.frames y su uso se retomará en el módulo de manipulación de bases de datos.

Finalmente, el operador `~` se verá en el módulo correspondiente a manipulación de bases de datos y obtención de estadísticos.

2. Asignación

En R, contrario a otros lenguajes de programación, se prefiere el operador `<-` en lugar del signo igual `=` para realizar asignaciones, es decir, para crear objetos con valores específicos. 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.

Ejecutemos el siguiente código seleccionando dos líneas a la vez ¿Qué resultado se muestra en consola? ¿Se registra algún error?

```
5 -> w
w

x <- 6
x

y = 7
y

8 = z
z
```

Cuando se crea un objeto (por ejemplo, una variable), su valor no se muestra de forma automática en la consola sino que tiene que imprimirse. Esto se hace usualmente en dos líneas de código como vimos en el ejemplo anterior. Sin embargo, existen dos formas compactas para "crear e imprimir" en una sóla línea de código. La primera es utilizando punto y coma `;` después de la asignación e indicando el nombre de la variable. La segunda es con la llamada "función identidad" que consiste en el uso de paréntesis `( )` al principio y al final de la asignación:

```
# Uso de punto y coma
ab <- 4; ab
```

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

```
# Función identidad
(ac <- -10)
```

```
## [1] -10
```

La asignación global con `<<-` se emplea cuando creamos funciones y nos permite acceder al valor de una variable fuera del entorno de dicha función. En otras palabras, con una asignación global tenemos una variable que existe por fuera de la función y que puede sustituir a otra creada en nuestro entorno de trabajo. Veamos un ejemplo:

```r
f <- function() { x<<-5 }
x <- 0; x
f(); x
# El valor de x en el entorno de trabajo es 5
```

Después de realizar asignaciones, tenemos objetos en nuestro entorno de trabajo los cuales podemos manipular utilizando algunos de los operadores que vimos anteriormente. Puesto que la mayor parte de las operaciones se hacen elemento a elemento, los objetos no necesitan tener una extensión similar y sus componentes son "reciclados" para llevar a cabo las tareas asignadas.

```r
# Secuencias de igual tamaño
(x <- 1:5)
(y <- 8:12)
x*y
y>x
x>y

# Secuencias de diferente tamaño
xx <- 3:10
yy <- 5:7
xx+yy
xx-yy
xx/yy
```

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 (o "colecciones") de números, caracteres, variables y observaciones dependiendo del contexto en el que sea utilizada. **Los vectores son la estructura de datos básica en R siendo los más comunes los de tipo lógico (TRUE, FALSE), numérico (con decimales), enteros (positivos y negativos) y caracter (cadena de texto).** Cuando generamos un vector, sus componentes se separan con comas.

Ejercicios:

- Crear un vector de números con los siguientes elementos: 1, los números del 2 al 8 y los números del -4 al 5 y asignarlo a un objeto llamado "nums".
- Crear un vector de 5 cadenas de caracteres con los nombres: Luis, Ana, Gaby, Ernesto y Andrés y asignarlo a un objeto llamado "nombres".

Regularmente buscaremos construir vectores de un mismo tipo. Sin embargo, es común que los elementos de un vector no cumplan esa condición pues podemos encontrar números con cadenas de caracteres o cadenas de caracteres con valores lógicos. Cuando esto sucede, los elementos son "coercionados" ("forzados") de manera automática para generar un valor de retorno común, es decir, que englobe a toda la colección de datos.

Para ilustrar cómo opera la coerción, verificaremos el tipo y clase de un vector con las funciones **`typeof()`** y **`class()`**. La primera nos regresa la forma en que un determinado objeto está guardado en el nivel interno del lenguaje y la segunda el tipo "abstracto" o "genérico" de un objeto. En otras palabras, **`typeof()`** nos regresa el dato de más bajo nivel mientras que **`class()`** nos da la "clase" de un objeto, que es un dato de más alto nivel. Por otra parte, Si deseamos saber si un vector pertenece a un tipo específico, empleamos **`is.`** seguida del atributo de interés.

Ejecutemos únicamente la primera línea de cada bloque de código (el resultado no se imprimirá en consola). ¿El primer vector será numérico o de caracteres? ¿El segundo será numérico o lógico? Verifiquemos nuestra intuición corriendo el resto de las líneas de cada bloque.

```
# Bloque uno
char_vec <- c(1:10, "perro", "niño", "15", "22")
typeof(char_vec); class(char_vec)
is.numeric(num_vec)
is.character(num_vec)
is.logical(num_vec)

# Bloque dos
logi_vec <- c(TRUE,TRUE,FALSE,FALSE,TRUE,5)
typeof(logi_vec); class(logi_vec)
is.numeric(logi_vec)
is.character(logi_vec)
is.logical(logi_vec)
```

La coerción de vectores puede hacerse manualmente empleando **`as.`** seguido del atributo de interés. Ejecutemos una a una las siguientes líneas de código. ¿Cuál es el resultado?

```
(char_vec_num <- as.numeric(char_vec))
(logi_vec_char <- as.character(logi_vec))
(logi_vec_num <- as.logical(logi_vec))
```

En el primero y tercer casos, la coerción nos arrojó valores **`NA`**. Esto ocurre cuando ciertos valores genéricos no pueden convertirse a otros más específicos. En el vector **char_vec**, las cadenas "perro"

y “niño” no tendrían un valor numérico concreto, mientras que “15” y “22” pueden fácilmente ser forzadas a números. En el caso de **logi_vec**, el dígito 5 no tiene sentido dentro de una escala que sólo adopta valores 0 (para `FALSO`) y 1 (para `VERDADERO`).

En consecuencia, podemos decir que **`as.numeric()`** transforma en número cualquier cadena susceptible de adquirir esa cualidad y en caso contrario arroja `NA`. Esta misma función convierte cualquier vector lógico en un vector de ceros y unos (0 para `FALSE` y 1 para `TRUE`)

Un caso particular de coerción automática sucede cuando tenemos vectores con números enteros y decimales pues los segundos tienen precedencia sobre los primeros. Por otra parte, cuando coercionamos a entero utilizando **`as.integer()`**, R lo hace sobre el número más bajo:

```
# Coerción a número flotante ("double")
double_vec <- c(1:10, 2.9, 3.5, -0.9)
typeof(double_vec); class(double_vec)
```

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

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

```
# Coerción a entero
as.integer(double_vec)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  2  3  0
```

Finalmente, como curiosidad, podemos invocar **`class()`** para definir un objeto de una clase personalizada, por ejemplo:

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

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

```
typeof(x)
```

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

4. Factores

Un factor es un vector que puede contener sólo valores predefinidos y es utilizado cuando guardamos datos categóricos. Aunque pueden verse como una extensión de cadenas de caracteres, lo cierto es que los factores son de hecho, enteros, por lo que su tratamiento requiere cierto cuidado. Pensemos, por ejemplo, en el siguiente vector de factores:

```
# Con set.seed garantizamos que el ejemplo sea replicable
set.seed(2019)
# Creamos un vector de niveles
niveles <- c("Bueno", "Regular", "Malo")
# Generamos una muestra aleatoria de niveles
x <- sample(niveles, 20, replace = TRUE)
# Convertimos a factor
x <- factor(x)
x
```

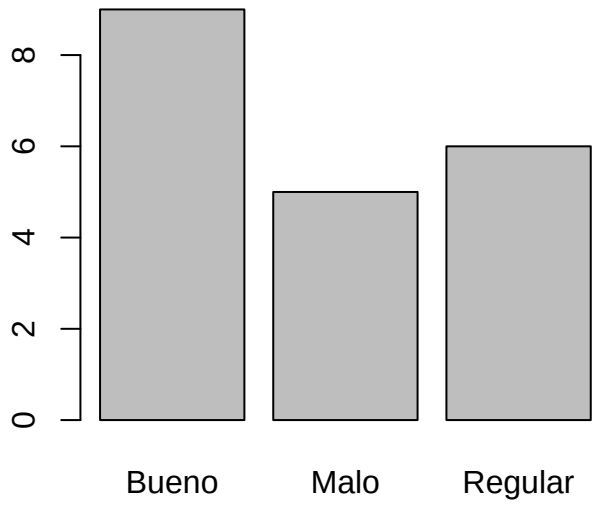
```
##  [1] Bueno   Bueno   Regular Bueno   Bueno   Bueno   Malo    Bueno   Malo
## [10] Regular Bueno   Bueno   Regular Regular Bueno   Malo    Malo    Regular
## [19] Regular Malo
## Levels: Bueno Malo Regular
```

```
str(x)
```

```
##  Factor w/ 3 levels "Bueno","Malo",..: 1 1 3 1 1 1 2 1 2 3 ...
```

Como puede observarse, **R** ordena por default alfabéticamente por lo que la escala resulta Bueno - Malo - Regular y los valores enteros asignados son ascendentes (1 para “Bueno”, 2 para “Malo” y 3 para “Regular”). De esta manera, al graficar las categorías se muestran en un orden que regularmente no es el esperado:

```
barplot(table(x))
```



R nos permite modificar este comportamiento de dos formas: estableciendo una ordenación simple haciendo uso del argumento **`levels`** o fijando una jerarquización explícita con **`levels`** y **`ordered=TRUE`**. La primera opción es útil por ejemplo, si deseamos visualizar las categorías en cierta disposición (Bueno-Regular-Malo) y la segunda si queremos hacer manifiesto que las categorías se ordenan de menor a mayor:

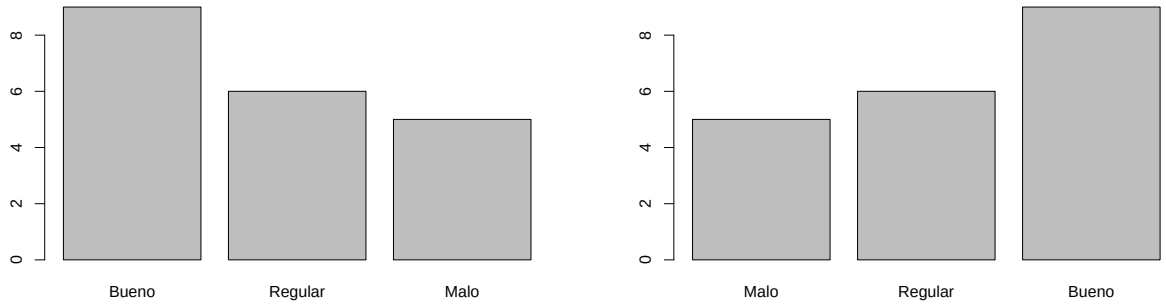
```
(x_levels <- factor(x, levels=c("Bueno", "Regular", "Malo")))
```

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

```
(x_ordered <- factor(x, levels=c("Malo", "Regular", "Bueno"),
    ordered=TRUE))
```

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

```r
barplot(table(x_levels))      #Izquierda
barplot(table(x_ordered))     #Derecha
```



Otro aspecto que merece mencionarse es el uso de números (ya sea en cadenas de caracteres,como enteros o como *doubles*) como factores. Pensemos por ejemplo en el siguiente vector:

```r
(fac_vec <- factor(c("519","200","102","519", "200",
    "330", "102", "102", "330", NA, NA)))
```

```
##  [1] 519  200  102  519  200  330  102  102  330  <NA> <NA>
## Levels: 102 200 330 519
```

```r
str(fac_vec)
```

```
##  Factor w/ 4 levels "102","200","330",..: 4 2 1 4 2 3 1 1 3 NA ...
```

Si quisiéramos tratar esta información como números, intuitivamente recurriríamos a coercionar con **as.numeric()**. Sin embargo, esto falla porque los factores son enteros y en principio no tienen el valor que representan:

```r
as.numeric(fac_vec)
```

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

Para poder extraer las cifras de un vector como **fac_vec** coercionamos a caracter y despues a numérico. El primer paso le "quita" al elemento su cualidad de entero y el segundo intenta obtener la cantidad. Para hacerlo en un sólo momento, "anidamos" funciones de tal manera que la primera función invocada corresponda al segundo paso y la segunda al primero:

```r
# En dos pasos
fac_char <- as.character(fac_vec)
(num_char <- as.numeric(fac_char))
```

```
##  [1] 519 200 102 519 200 330 102 102 330  NA  NA
```

```r
# Con funciones "anidadas"
(num_fac_vec <- as.numeric(as.character(fac_vec)))
```

```
##  [1] 519 200 102 519 200 330 102 102 330  NA  NA
```

```r
# ¿Obtenemos el mismo resultado?
identical(num_char, num_fac_vec)
```

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

Debido a este tipo de comportamiento, al llamar a funciones como **`read.table()`**, **`read.csv()`**, entre otras que se abordarán en el módulo III, se indica el argumento **`stringsAsFactors=FALSE`**. Sin embargo, cuando estudiemos expresiones regulares veremos que hay funciones que coercionan sus resultados a factores, por lo que si necesitamos números, será necesario recurrir al procedimiento descrito.

5. Listas y data.frames

En términos de complejidad, `vector` es la colección de objetos más básica, seguida por `matrix` (vectores de más de una dimensión), `list` y `data.frame`. Los dos primeros pueden albergar sólo un tipo de dato mientras que las listas son los únicos objetos capaces de almacenar diversidad de datos (incluyendo otras listas y `data.frames`). Por su parte, las `data.frame` y otros objetos como el `tibble` son especializaciones del objeto `list`.

En lo que sigue, describiremos brevemente las listas y las data.frames por ser de uso común en el manejo de datos. Sin embargo, puede consultarse documentación sobre matrices en [esta liga]

Las listas se denominan "vectores recursivos" y se crean con la función **`list()`**. Gracias a sus características no coercionan los elementos que pertenecen a ella como ocurre con la función **`c()`**. Para comprender con mayor claridad, Combinemos un mismo conjunto de elementos usando **`c()`** y **`list()`**. ¿Cuáles son las diferencias más evidentes entre los objetos creados?

```r
# Vector
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"
##  - attr(*, "names")= chr [1:9] "" "" "" "" ...
```

```r
# Lista
list_ej <- list(c(1,4,3), 2, "perro", list=(1:4))
str(list_ej)
```

```
## List of 4
##  $     : num [1:3] 1 4 3
##  $     : num 2
##  $     : chr "perro"
##  $ list: int [1:4] 1 2 3 4
```

Las listas suelen emplearse en la construcción de funciones. Un ejemplo canónico del uso de listas es la función **`lm()`** para el cálculo de regresiones lineales. Aunque un objeto creado con dicha función es de clase **`lm`**, en el nivel interno del lenguaje es una lista:

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

g <- lm(y ~ x)
class(g)
```

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

```r
typeof(g)
```

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

Una `data.frame` podría definirse como una lista de vectores donde cada columna (variable) es un elemento de la lista. A su vez, cada columna es un vector de determinado tipo. Cuando utilizamos la función **`data.frame`** para crear un objeto manualmente, el número de columnas puede diferir pero no el número de elementos (renglones). Comprobemos que la estructura básica de una **`data.frame`** es una lista con un sencillo ejemplo:

```r
(df_uno <- data.frame(a=c(1:3), b=c("perro", "gato", "ratón"),
                      stringsAsFactors=FALSE))
```

```
##   a     b
## 1 1 perro
## 2 2  gato
## 3 3 ratón
```

```r
class(df_uno)
```

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

```r
typeof(df_uno)
```

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

6. Acceso a elementos

En un vector, se puede acceder a los elementos **por posición o por condición** mediante el operador `[ ]`. En el primer caso, usamos índices (es decir, dígitos) que se refieren a una determinada ubicación dentro del vector) y en el segundo los operadores lógicos que vimos en la primera sección.

```r
set.seed(2020)
(x <- sample(c(1:35,4:24,-3:10),20, replace=TRUE))
```

```
##  [1] 28 22  5 17  4 10 10 17 24 10 29  3  6  4 16 15  7 18 20 16
```

Para acceder únicamente a un elemento o a una secuencia específica de ellos, bastará con **`x[num]`** o **`x[num:num]`** , donde "num" es un número natural cualquiera (esto es, no negativo o cero). Si utilizamos un número o números negativos, obtendremos todos los elementos excepto aquellos a los que hace referencia el índice o índices y si ingresamos una cantidad mayor al número de elementos que tenemos en nuestro vector, recibiremos como resultado una **`NA`** por cada índice fuera de los límites. ¿Cuántas **`NA`** se muestran en consola al ejecutar la última línea de código del siguiente bloque? ¿Por qué?

```
# Ejemplos
x[12]     # Elemento en la posición 12
x[-5]     # Todos los elementos excepto el ubicado en la posición 5
x[-1:-5]  # Todos los elementos excepto los primeros 5
x[-(1:5)] # Mismo resultado que el anterior
x[24]     # NA
x[15:23]
```

Estas reglas aplican para la selección de dos o más elementos consecutivos. Sin embargo, valiéndonos de la función `c()` podemos crear un vector de índices como `x[c(1:5,15:20)]` para imprimir los elementos en las posiciones 1 a 5 y 15 a 20 o `x[c(1, 3, 5)]` para los elementos 1, 3 y 5 . Si, por el contrario, queremos todos menos las x en las posiciones 1 a 5 y 15 a 20, tenemos alternativas como:

```
# Con índices negativos
x[c(-1:-5,-15:-20)]
```

```
## [1] 10 10 17 24 10 29  3  6  4
```

```
# Haciendo negativo el vector con el operador -
x[-c(1:5,15:20)]
```

```
## [1] 10 10 17 24 10 29  3  6  4
```

El acceso por condiciones se denomina también "acceso por valores" aunque esto es inexacto debido al uso de operadores lógicos. Supongamos que de nuestro vector `x` queremos saber cuáles números son mayores o iguales a 15. El siguiente código genera un vector de la misma longitud de `x` y para cada elemento indica `TRUE` o `FALSE` dependiendo si cumple la condición de ser mayor o igual a 15.

```
x>=15
```

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

```
# table(x>=15) para un conteo rápido de verdaderos y falsos
```

Como podemos ver, tenemos en total 11 números mayores o iguales a 15 (11 `TRUE`). Por tanto, si hacemos `x[x>=15]` lo que le decimos a **R** es: "Muestra los elementos de x para los cuales la condición x>=15 es verdadera".

Valiéndonos de los operadores lógicos, podemos acceder a elementos estableciendo diversas condiciones. ¿Qué números se obtienen al ejecutar las siguientes líneas de código?

```r
# x menor a 5 o x mayor o igual 20
x[x<5 | x>=20]

# x mayor a 20 y diferente a 25
x[x>20 & x!=25]

# x mayor o menor a 1
x[x>30 | x<1]
```

Para acceder a los elementos de una lista, podemos emplear los operadores **`[ ],[[ ]]` y `$`**.Los primeros dos nos permiten establecer posiciones y nombres (entre comillas) y el último indicar nombres. Debe señalarse que al igual que sucede con vectores, los operadores se sitúan a la derecha de los objetos.

Veamos un ejemplo con algunas bases de datos que se cargan por default en la instalación de R. Ejecutemos el siguiente código desde un script y observemos los resultados:

```r
mylist <- list(iris=data.frame(iris),
    cars=data.frame(mtcars), arboles=data.frame(trees))
str(mylist)
```

Se aprecia que la lista contiene 3 elementos de la clase `data.frame` y, para cada uno, se despliegan también sus nombres de columnas.

Trabajemos con el tercer elemento para observar el comportamiento de los operadores arriba mencionados. Para ello, realicemos tres asignaciones y observemos la estructura de los objetos creados:

```r
arb1 <- mylist[3]       # Equivalente a mylist["arboles"]
arb2 <- mylist[[3]]     # Equivalente a mylist[["arboles"]]
arb3 <- mylist$arboles

str(arb1)
str(arb2)
str(arb3)
```

Podemos deducir entonces que, en este caso, `[ ]` nos regresa una sub-lista; `[[ ]]` una `data.frame` y **`$`** otra `data.frame`. En general, `[ ]` tendrá como valor de retorno una sub-lista y los operadores restantes el objeto como tal (un vector o una `data.frame`, por ejemplo). Es fundamental tomar en cuenta que si no asignamos un nombre a los elementos de nuestras listas, no podremos acceder a ellos utilizando **`$`** por lo que tendríamos que recurrir a índices.

```
lista_dos <- list(data.frame(iris),
    cars=data.frame(mtcars), arboles=data.frame(trees))
lista_dos$iris
```

```
## NULL
```

Las listas permiten trabajar con un mayor nivel de complejidad. Analicemos la estructura de la siguiente lista:

```
lista_tres <- list(x=c(1:5), y=c(2:6), animales=c("perro","gato",
    "perico", "ballena"), piedras=data.frame(rock))
str(lista_tres)
```

```
## List of 4
##  $ x       : int [1:5] 1 2 3 4 5
##  $ y       : int [1:5] 2 3 4 5 6
##  $ animales: chr [1:4] "perro" "gato" "perico" "ballena"
##  $ piedras :'data.frame':    48 obs. of  4 variables:
##   ..$ area : int [1:48] 4990 7002 7558 7352 7943 7979 9333 8209 8393 6425 ...
##   ..$ peri : num [1:48] 2792 3893 3931 3869 3949 ...
##   ..$ shape: num [1:48] 0.0903 0.1486 0.1833 0.1171 0.1224 ...
##   ..$ perm : num [1:48] 6.3 6.3 6.3 6.3 17.1 17.1 17.1 17.1 119 119 ...
```

¿Cómo podríamos acceder, por ejemplo, a la columna "area" de la `data.frame` "piedras"? Recordando que con `[[ ]]` y `$` accedemos al objeto como tal, lo siguiente sería extraer su primer elemento con `[ ]` de la misma forma que hicimos con vectores:

```
piedras_area <- lista_tres[[4]][1]
area_piedras <- lista_tres$piedras[1]
identical(piedras_area, area_piedras)
```

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

Ejercicio. De nuestra "lista_tres", extraer:

1. El tercer elemento del vector "x" y asignarlo al objeto "xx"
2. El elemento "gato" del vector "animales" y asignarlo al objeto "yy"

3. Las columnas 1 y 3 de la `data.frame` "piedras" y asignarlo al objeto "zz"

El acceso a elementos es una habilidad clave en el manejo de datos. Como vimos con las listas, nos permite crear subconjuntos de información y este mismo razonamiento es válido para vectores, matrices y, por supuesto, para `data.frames`, como se explicará en el cuarto módulo. Su utilidad se extiende también a la modificación de valores. Supongamos que tenemos un vector como el siguiente y queremos cambiar los "NA" por ceros así como homogeneizar en 15 todos los valores que sean iguales o mayores a esa cantidad.

```
set.seed(2300)
x <- c(sample(c(1:35),15, replace=TRUE), c(rep("NA",5)))
(x <- sample(x))
```

```
##  [1] "14" "35" "10" "15" "9"  "15" "17" "5"  "NA" "NA" "13" "19" "22" "10" "NA"
## [16] "NA" "7"  "NA" "18" "35"
```

En primer lugar, se aprecia que tenemos una cadena de caracteres, por lo que tendremos que realizar tres tareas:

- Cambiar los "NA" por "0"
- Coercionar a numérico
- Homogeneizar valores iguales o mayores a 15

Cuando queremos extraer un subconjunto, el objeto que guardará esa información es al que apunta la flecha de asignación. Cuando modificamos valores, queremos que nuestra asignación opere sobre el subconjunto de datos de interés sin modificar el resto de los datos. Con esto en mente, para cambiar "NA" por "0" hacemos lo siguiente:

```
x[x=="NA"] <- "0"
```

Posteriormente, puesto que queremos coercionar a numérico todo el vector y mantener su mismo nombre, sobreescribimos nuestro vector con `x<- as.numeric(x)` y finalmente homogeneizamos los valores mayores o iguales a 15 siguiendo la lógica del primer paso: `x[x>=15] <- 15`.

En una sóla línea los tres pasos se resumen en:

```
x[x=="NA"] <- "0"; x<- as.numeric(x); x[x>=15] <- 15; x
```

```
##  [1] 14 15 10 15  9 15 15  5  0  0 13 15 15 10  0  0  7  0 15 15
```

7. Atributos

Los atributos permiten alojar metadatos de los objetos mediante una lista de nombres. Se puede acceder a ellos mediante la función **`attributes()`**.

```
df_uno <- data.frame(a=c(1:3), b=c("perro", "gato", "ratón"),
                     stringsAsFactors=FALSE)

attributes(df_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` y pueden reasignarse directamente en el objeto, es decir, sin crear una copia modificada del mismo.

```
names(df_uno) <- c("uno", "dos")
names(df_uno)
```

```
## [1] "uno" "dos"
```

Cabe destacar que **`names()` es un atributo que sólo aplica a objetos de clase `data.frame`**. Como alternativa existe `colnames()` que además de tener un nombre más explicativo e intuitivo, funciona también en matrices.

```
colnames(df_uno) <- c("UNO", "DOS")
colnames(df_uno)
```

```
## [1] "UNO" "DOS"
```