Nota2 - Introducción

Santiago Casanova y Ernesto Barrios

Uso Básico

Ahora vamos a empezar a familiarizarnos con el ambiente de R y específicamente su sintaxis especial.

Declaración de variables

`R` tiene un operador especial para asignar valores a variables, diferente a otros lenguajes de programación. Además, las variables son flexibles y pueden pasar de contener un tipo de dato a otro sin problema. Por los mismo no es necesario especificar el tipo de dato como se hace en C o en Java. El operador de asignación es `<-`.

También se puede asignar variables con `=` pero, por convención, `=` se reserva para operaciones dentro de funciones o paréntesis.

Vamos a asignar algunos valores a una serie de variables:

```
numero <- 20
numero2 <- 1234.56
entero <- 20L
texto1 <- "ejemplo"
texto2 <- 'también se puede con comillas simples'
booleano <- T
booleano = TRUE
booleano2 = F
booleano2 = FALSE

typeof(numero)
```

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

```
typeof(entero)
```

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

Como describimos, no se necesita especificar qué tipo de dato queremos en cada variable ya que esto puede cambiar más adelante. Hablando de tipos de datos, vamos a ver cuáles son las opciones que maneja `R`.

Tipos de Datos

1. Numérico: Ambos tipos `integer` y `double` pertenecen a la clase numérica.

- Si se quiere declarar específicamente como entero se puede escribir con una `L` al final del número. Ejemplo: `2L`. De lo contrario `R` lo guardará como `double`.

2. Caracter: No hay diferencia entre `string` y `character`. En `R` todos los literales entre comillas son considerados modo y tipo `character`.
3. Booleano (lógico): Como vimos en el ejemplo anterior, se escribe `TRUE` o `FALSE` todo con mayúsculas, o bien, sólo `T` o `F`.

Valores no-disponibles (Not Available)

`R` tiene varias maneras de manejar los valores no disponibles dependiendo del tipo de dato. Un valor no disponible actúa como un marcador de posición en una estructura donde *debería* haber un dato. Esto lo veremos más a fondo cuando lleguemos al tema de estructuras de datos.

Cuando el valor no está disponible se representa con `NA`. De la misma manera, los valores `NA` pueden ser más específicos. `NA_real_`, `NA_integer_`, `NA_character_` y `NA_complex_` describen puntualmente el tipo de dato que falta pero en última instancia todos son tratados como `NA` por `R`. Por ejemplo:

```
NA
```

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

```
NA_real_
```

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

tienen la misma salida.

Además del `NA`, R reconoce `NaN` como **Not a Number** y es específico para cuando el resultado de una operación matemática resulta en algo imposible. Por ejemplo la división de 0 entre 0. Es diferente a `NA` porque no indica que falte un valor sino que el valor resultó en un no-número.

```
print(0/0)
```

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

Por último tenemos el valor `NULL` que indica la ausencia de todo dato. Puede ser usado para des-asignar variables. Por ejemplo:

```
var <- 'Ejemplo'
print(var)
```

```
## [1] "Ejemplo"
```

```
var <- NULL #Asignamos NULL para quitarle el valor a la variable
print(var)
```

```
## NULL
```

Operadores

La mayoría de los operadores son muy similares o iguales a otros lenguajes de programación por lo que no los analizaremos a fondo.

```
#suma
print(20 + 3)
```

```
## [1] 23
```

```
#resta
print(20 - 3)
```

```
## [1] 17
```

```
#multiplicación
print(20 * 3)
```

```
## [1] 60
```

```
#división
print(20 / 3)
```

```
## [1] 6.666667
```

```
#potencia (diferente a otros)
print(20 ^ 3)
```

```
## [1] 8000
```

```
#módulo
print(20 %% 3)
```

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

Operadores lógicos

En R, los operadores lógicos básicos son prácticamente iguales a otros lenguajes de programación pero además, R reconoce una serie de pruebas lógicas especiales que veremos a fondo más adelante.

```
#mayor que (o mayor/igual)
print(20 > 3) # >=
```

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

```
#menor que (o menor/igual)
print(20 < 3) # <=
```

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

```
#comparar (igualdad)
20 == 3
```

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

```
#comparar (desigualdad)
20 != 3
```

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

Hasta ahora hemos usado la función `print()` para obtener una respuesta de la consola. Sin embargo, como podemos ver en las dos últimas salidas de los ejemplos anteriores, R también regresa el resultado de una operación sin el uso específico de `print()` cuando está en una sesión. De ahora en adelante lo omitiremos en nuestros ejemplos.

Otra forma de obtener valores lógicos es revisando el tipo de dato que tenemos en una variable. Para esto, R nos presenta una familia de funciones que revisan el tipo de dato y regresan un valor lógico. La familia de funciones `is.*` ejecuta exactamente este mismo proceso en una sola función.

```
palabra <- 'palabra'
numeros <- 123.2
numeros_pal <- '123.2'

#Preguntamos si son de tipo caracter
is.character(palabra)
```

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

```
is.character(numeros)
```

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

```
#Preguntamos si son de tipo numérico
is.numeric(palabra)
```

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

```
is.numeric(numeros)
```

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

Una familia de funciónes similar es `as.*`. Como podría ser obvio, en lugar de revisar si es el tipo que buscamos, intenta convertir el dato.

```
as.numeric(palabra) #Nos avisa que no pudo convertir, por lo tanto tenemos NA
```

```
## Warning: NAs introduced by coercion
```

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

```
as.numeric(numeros) #Nada cambia
```

```
## [1] 123.2
```

```
as.numeric(numeros_pal) #Ahora no tiene comillas alrededor, es de tipo numerico
```

```
## [1] 123.2
```

Ambas `is.*` y `as.*` funcionan con una gran cantidad de tipos, inculyendo `numeric`, `character`, `na`, `factor`, `function`, etc.

Estructuras de Datos

Vector Atómico

La estructura más básica para recopilar datos en `R` es un **Vector Atómico**. Estos se construyen con la sintaxis `c(a1, a2)` que concatena los valores deseados. Un vector atómico sin valores no es considerado una estructura de datos. `R` lo reconoce como un valor nulo `NULL` que analizamos en la sección anterior.

```
c()
```

```
## NULL
```

Ahora veamos un par de ejemplos de vectores.

```
c(1,2,3,4,5)
```

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

```
primer_vector <- c(1,2,3, 'a', 'b', 'c', NA, NA_real_ ,TRUE)
print(primer_vector)
```

```
## [1] "1"    "2"    "3"    "a"    "b"    "c"    NA     NA     "TRUE"
```

El primer vector concatena exclusivamente datos numéricos, por lo que el vector entero será de tipo numérico. Sin embargo, el segundo vector incluye datos de tipo numérico, caracter y lógico. Por esto, `R` considera a todo el vector de tipo `character` y en la respuesta los números de nuestro vector están encerrados en comillas.

Es importante notar que los `NA` no se convierten a tipo `character` porque `NA` no es un tipo de dato, representa la ausencia de uno.

Nótese que el vector atómico es la estructura *default* de `R`. Por lo tanto si no se especifica cómo guardar una colección de datos, `R` la guardará en un vector

```
#La sintaxis de dos números separados por dos puntos ":" denota un rango
vec <- 1:5
vec
```

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

Vemos que la consola nos regresa un vector.

Por último, cada elemento de un vector puede tener nombre y se le asigna de la siguiente manera:

```
vec_con_nombres <- c(elemento1 = 1234, elemento2 = "abcd")
vec_con_nombres
```

```
## elemento1 elemento2
##    "1234"    "abcd"
```

Por el otro lado, si le queremos quitar los nombres a un vector podemos utilizar la función `unname()` con el vector como parámetro

```
unname(vec_con_nombres)
```

```
## [1] "1234" "abcd"
```

Lista

Las listas agregan un grado de complejidad a la estructura "lineal" que son los vectores. Cada elemento de una lista puede ser una estructura de datos diferente por lo que siempre mantendrá los datos con su tipo original. Para construir una lista se utiliza la función `list()`

Las listas pueden llegar a ser muy complejas (y por lo tanto muy útiles para recopilar datos diversos) pero en su forma más básica las podemos usar como una especie de vector que respeta los tipo de datos.

```
list(1,2,3,4,5)
```

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

```
primer_lista <- list(1,2, 'a', 'b', NA,TRUE)
print(primer_lista)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] "a"
##
## [[4]]
## [1] "b"
##
## [[5]]
```

```
## [1] NA
##
## [[6]]
## [1] TRUE
```

Inmediatamente podemos ver la diferencia entre las respuestas de la consola con vectores y con listas. Todos los valores que ingresamos mantienen su tipo original (los valores numéricos no tienen comillas alrededor), pero además cada elemento está separado en un renglón.

Como dijimos, las listas no solo pueden contener datos individuales, sino también estructuras. Si combinamos los conceptos de lista y vector, podemos crear una lista de vectores, cada uno de un tipo diferente y un nombre propio.

```
lista_compleja <- list(vector1 = c(1,2,4, 12873812), vector2 =  c('a', 'b','c','g', 'zh', 'ya'), vector
lista_compleja
```

```
## $vector1
## [1]        1        2        4 12873812
##
## $vector2
## [1] "a"  "b"  "c"  "g"  "zh" "ya"
##
## $vector_na
## [1]  NA  NA NaN
##
## $numero
## [1] 201
```

Las listas permiten guardar datos en varios niveles y dimensiones.

Esto presenta la pregunta ¿cómo accedemos a estos datos?

Acceso a datos a través índices

Los índices en `R` inician en `1` y se escriben dentro de `[]` para acceder al dato correspondiente.

En nuestro ejemplo del primer vector:

```
#La segunda posición del vector
primer_vector[2]
```

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

Esto nos indica que en la posición 2 de `primer_vector` tenemos un "2" de tipo `character` (notar comillas).

Si queremos modificar algún elemento del vector, llamamos al elemento que queremos cambiar como hicimos arriba y le asignamos un nuevo valor con el operador `<-`.

```
primer_vector[2] <- 4
primer_vector
```

```
## [1] "1"    "4"    "3"    "a"    "b"    "c"    NA     NA     "TRUE"
```

Vemos que el valor en la segunda posición cambió a `4`.

En el caso de los vectores que tienen nombres también podemos usar el nombre para acceder a los valores.

```
vec_con_nombres['elemento1']
```

```
## elemento1
##    "1234"
```

El proceso para acceder a elementos de listas es un poco diferente. Si intentamos hacer lo mismo, obtendremos otra lista de un solo elemento que contiene al dato buscado, en lugar de el dato buscado por sí solo.

```
lista_compleja[1]
```

```
## $vector1
## [1]        1        2        4 12873812
```

Para obtener sólamente el primer elemento de la lista se usa un corchete doble `[[]]`

```
lista_compleja[[1]]
```

```
## [1]        1        2        4 12873812
```

Ahora vemos que nuestra salida es un vector, el elemento que teníamos guardado como primer elemento de `lista_compleja`.

Como nuestro resultado es un vector esto significa que va a responder exactamente como lo haría un vector. Si quisiéramos acceder al tercer elemento de este vector escribimos:

```
lista_compleja[[1]][3]
```

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

En el caso de nuestra lista con nombres también podemos usar el nombre para obtener elementos. Por ejemplo:

```
lista_compleja[['vector2']]
```

```
## [1] "a"  "b"  "c"  "g"  "zh" "ya"
```

Ya que la lista simplemente está guardando al vector como un elemento, todas las propiedades de la estructura funcionan igual cuando la llamamos a través de índices.

Los elementos de las listas con nombre también se pueden acceder a través del operador de acceso `$`. Al usarlo, ya no utilizamos el doble corchete ni las comillas sino que usamos el operador y el nombre directamente.

```
lista_compleja$vector2
```

```
## [1] "a"  "b"  "c"  "g"  "zh" "ya"
```

Por último veamos la función `unlist()`, la cual convierte una lista a un vector.

```
unlist(primer_lista)
```

```
## [1] "1"    "2"    "a"    "b"    NA     "TRUE"
```

En un caso extremo, una lista pude contener vectores, funciones, arreglos, modelos, etc.

Factores

Los factores son una especie de híbrido entre tipo de dato y estructura de dato. Toman una serie de datos de tipo caracter y les dan importancia categórica. Las cadenas dejan de ser una colección de letras y se vuelven una "categoría" agrupable.

Es un concepto complicado así que veamos un ejemlpo.

```
medallas <- c('Oro','Plata','Bronce','Oro','Plata','Plata','Plata','Oro')
factor(medallas)
```

```
## [1] Oro    Plata  Bronce Oro    Plata  Plata  Plata  Oro
## Levels: Bronce Oro Plata
```

Ahora el vector nos indica cuáles son los niveles del vector proporcionado. Esto puede ser útil para graficar o mucho más adelante, para entrenar modelos de clasificación.

También le podemos indicar cuáles son los niveles de un factor, aunque el vector que estamos convirtiendo no los contenga. Esto lo hacemos con el argumento `levels`.

```
medallas2 <- c('Oro','Plata','Oro','Plata','Plata','Plata','Oro')
fac_medallas <- factor(medallas2, levels = c('Oro','Plata','Bronce'))
fac_medallas
```

```
## [1] Oro   Plata Oro   Plata Plata Plata Oro
## Levels: Oro Plata Bronce
```

```
#Revisamos sólo los niveles
levels(fac_medallas)
```

```
## [1] "Oro"    "Plata"  "Bronce"
```

Incluso le podemos indicar a `R` que las categorías tienen orden y cuál es la jerarquía dentro de estas.

```
fac_medallas_2 <- factor(medallas2, levels = c('Oro','Plata','Bronce'), ordered = T)
fac_medallas_2
```

```
## [1] Oro   Plata Oro   Plata Plata Plata Oro
## Levels: Oro < Plata < Bronce
```

```
levels(fac_medallas_2)
```

```
## [1] "Oro"    "Plata"  "Bronce"
```

Intuitivamente vemos que esa jerarquía no es correcta. Modifiquemosla con la función `ordered()`.

```
ordered(fac_medallas_2, c('Bronce','Plata','Oro'))
```

```
## [1] Oro   Plata Oro   Plata Plata Plata Oro
## Levels: Bronce < Plata < Oro
```

Data Frames

Hasta ahora hemos visto estructuras de datos simples, en el caso de los vectores, o compuestas sin relación, en el caso de las listas. Los data frames nos permiten organizar datos que se relacionan entre sí. Podemos pensar que los data frames se comportan como lo haría una hoja de excel, con filas y columnas que contienen datos de cualquier tipo.

Para crear un data frame usamos la función `data.frame()` y le proporcionamos vectores atómicos como columnas.

```
#Creamos un arreglo con tres vectores. Cada uno corresponde a una columna de nuestro arreglo
arreglo <- data.frame(nombre = c('Ernesto', 'Santiago'),
                      clave = c(21,18),
                      booleano = c(T,F))

arreglo
```

```
##     nombre clave booleano
## 1  Ernesto    21     TRUE
## 2 Santiago    18    FALSE
```

Nótese que, como las columnas están generadas por vectores atómicos, cada una solo puede tener un tipo de dato.

Así como las columnas tienen nombre, también le podemos asignar un nombre o identificador a las filas utilizando `row.names` como un argumento.

```
#Ahora también incluimos el argumento row.names. Este no va a ser una columna nueva
arreglo <- data.frame(nombre = c('Ernesto', 'Santiago'),
                      clave = c(21,18),
                      booleano = c(T,F),
                      row.names = c(21,18))

arreglo
```

```
##      nombre clave booleano
## 21  Ernesto    21     TRUE
## 18 Santiago    18    FALSE
```

Esto nos da un identificador en el arreglo que no es parte de los datos pero que es útil para organizar la información.

Para acceder a los valores individuales usamos la misma sintaxis que usamos para los vectores pero ahora con dos posiciones separadas por una coma. La primera corresponde a la **fila** que queremos y la segunda a la **columna**.

```
#Primera columna y primera fila
arreglo[1,1]
```

```
## [1] "Ernesto"
```

Es el dato correspondiente a la primera fila y primera columna.

Si solo proporcionamos un índice sin usar la coma, `R` regresa la columna que corresponda al índice.

```
#La primera columna
arreglo[1]
```

```
##      nombre
## 21  Ernesto
## 18 Santiago
```

Sin embargo la forma propia de acceder este valor es con un espacio en blanco en la sección de las filas y el índice de la columna después de la coma.

```
#La primera columna
arreglo[,1]
```

```
## [1] "Ernesto"  "Santiago"
```

Por otro lado, si queremos solo la primera fila, escribimos la coma pero dejamos en blanco lo que viene después.

```
arreglo[1, ]
```

```
##     nombre clave booleano
## 21 Ernesto    21     TRUE
```

De manera similar a las listas con nombre, podemos usar el operador de acceso `$` para seleccionar columnas por nombre.

```
arreglo$clave
```

```
## [1] 21 18
```

El resultado es de nuevo un vector.

Explorar las estructuras

Podemos también explorar las estructuras de datos para obtener información básica que nos ayuda a comprender los datos y el tipo de estructura con el que estamos trabajando.

En el caso de los arreglos, si queremos saber el número de renglones y columnas usamos `dim()`. Esta función regresa un vector con dos entradas numéricas que nos indican el número de renglones y columnas. Alternativamente podemos usar `ncol()` o `nrow()` para obtener estos datos individualmente.

```
#Ambos datos
dim(arreglo)
```

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

```
#Solo columnas
ncol(arreglo)
```

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

```
#Solo renglones
nrow(arreglo)
```

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

```
#Obtenemos renglones usando los índices del vector resultado de dim()
dim(arreglo)[1]
```

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

En el caso de los vectores, como solo estamos trabajando con una dimensión (ya sea vector renglón o vector columna), no se usa `dim()`. Para obtener el *largo* de los vectores se usa `length()`. Si intentamos usar `dim()` con un vector, la consola nos regresará un resultado nulo.

```
#No regresa nada útil
dim(primer_vector)
```

```
## NULL
```

```
#Solo tiene una entrada con el largo del vector
length(primer_vector)
```

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

Además de las dimensiones de nuestras estructuras, también podemos explorar otras propiedades. La función `summary()` funciona tanto para vectores como para arreglos y nos regresa datos básicos que nos ayudan a entender cómo está construido el arreglo o vector.

Por otro lado, podemos ver una sección pequeña del *data frame* o arreglo, ya sea la parte superior o inferior, con las funciones `head()` y `tail()`. Estas funciones toman dos argumentos: el primero debe ser un arreglo y el segundo (opcional) indica cuántas filas debe regresar la función. Si no se proporciona el segundo argumento, las funciones regresan 6 filas por default.

`R` tiene una serie de **arreglos ejemplo** cargados en la memoria en todo momento. Para nuestro ejemplo de exploración de datos vamos a utilizar el **data frame** `mtcars`.

```
mtcars
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

```
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

A primera vista es demasiada información para procesar rápidamente. Usemos las técnicas de exploración de esta sección para conocer lo más posible acerca de `mtcars`.

```
#El tamaño
dim(mtcars)
```

```
## [1] 32 11
```

```
#Los nombres de las columnas
colnames(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
#Los nombres de los renglones
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```r
#Primeras filas
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
#Primeras 2 filas
head(mtcars, 2)
```

```
##               mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```r
#Últimas 2 filas
tail(mtcars, 2)
```

```
##                mpg cyl disp  hp drat   wt qsec vs am gear carb
## Maserati Bora 15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
## Volvo 142E    21.4   4  121 109 4.11 2.78 18.6  1  1    4    2
```

```r
#Resumen
summary(mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

En este último vemos que nos da el resumen de cada vector (columna) de nuestro arreglo ejemplo. Esto es el mínimo, el máximo, la media, la mediana y los cuantiles. Si hubiera algún valor `NA`, `summary()` nos daría el número total como parte del reporte.

Operaciones de vectores

Cuando tenemos una serie de datos numéricos guardada en un vector le podemos aplicar varias operaciones como suma, promedio (media) o mediana, entre otras. Por ejemplo:

```
vector_numerico <- c(123,546.7,333,32,1)
vector_numerico2 <- c(2,2,4,4,5,2)

#sumar todas las entradas
sum(vector_numerico)
```

```
## [1] 1035.7
```

```
#obtener la media
mean(vector_numerico)
```

```
## [1] 207.14
```

```
#obtener la mediana
median(vector_numerico)
```

```
## [1] 123
```

```
#redondear los valores
round(vector_numerico)
```

```
## [1] 123 547 333  32   1
```

```
#redondear a un dígito
round(vector_numerico, 1)
```

```
## [1] 123.0 546.7 333.0  32.0   1.0
```

```
#sacar el mínimo
min(vector_numerico)
```

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

```
#sacar el máximo
max(vector_numerico)
```

```
## [1] 546.7
```

```
#obtener la desviación estándar
sd(vector_numerico)
```

```
## [1] 229.8853
```

```
#multiplicar dos vectores (del mismo tamaño)
vector_numerico*vector_numerico2
```

```
## Warning in vector_numerico * vector_numerico2: longer object length is not a
## multiple of shorter object length
```

```
## [1]  246.0 1093.4 1332.0  128.0    5.0  246.0
```

```
#contar ocurrencias de un valor
table(vector_numerico2) #útil para graficar
```

```
## vector_numerico2
## 2 4 5
## 3 2 1
```

```r
#Usando vectores como conjuntos (funciona para no numéricos también)
union(vector_numerico, vector_numerico2) #Unión
```

```
## [1] 123.0 546.7 333.0  32.0   1.0   2.0   4.0   5.0
```

```r
intersect(vector_numerico, vector_numerico2) #Intersección
```

```
## numeric(0)
```

```r
setdiff(vector_numerico, vector_numerico2) #Diferencia (elementos del primero que no están en el segund
```

```
## [1] 123.0 546.7 333.0  32.0   1.0
```

```r
setequal(vector_numerico, vector_numerico2) #Si son iguales o no (sin importar el orden)
```

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

La función `table()` recibe un vector y cuenta las repeticiones de cada valor. Regresa un vector con nombres donde el nombre indica el valor que se está contando y el valor del vector las veces que se repite. En este caso tenemos que el 2 se repite 3 veces, el 4 2 veces y el 5 una vez.

Es claro que como las columnas de un arreglo son vectores, también se puede aplicar este tipo de funciones para hacer operaciones con columnas numéricas de arreglos.

Otras Funciones

Para cerrar esta sección vamos a presentar varias funciones útiles para el trabajo con `R`.

La función `rep()` repite un argumento las veces que le indiquemos. La función `seq()` crea una secuencia de números. La función `sqrt()` regresa la raíz cuadrada del número ingresado.

Veamoslas en acción.

```r
#Repetir el número 5, 8 veces
rep(5,8)
```

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

```r
#Repetir la palabra "Hola" 3 veces
rep('Hola', 3)
```

```
## [1] "Hola" "Hola" "Hola"
```

```r
#Una secuencia del 1 al 7
seq(7)
```

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

```r
#Una secuencia del 4 al 8
seq(4,8)
```

```
## [1] 4 5 6 7 8
```

```r
#Una secuencia de los pares de 10 a 18
seq(10,18,2) #El tercer argumento nos dice cada cuanto se hace la secuencia
```

```
## [1] 10 12 14 16 18
```

```r
#La raíz cuadrada de 2
sqrt(2)
```

```
## [1] 1.414214
```

Uso de RStudio

Hasta ahora hemos usado solamente la consola para interactuar con R pero hay maneras más amigables de escribir comandos y recibir respuestas.

Una interfaz o IDE (Integrated Development Enviroment) como RStudio nos permite escribir archivos ejecutables en los que podemos incluir todo un programa y editarlo sin tener que correr cada línea todas las veces.

Aquí incluimos una guía para instalar RStudio en diferentes sistemas operativos.

Windows

1. Primero visitar https://www.rstudio.com/products/rstudio/download/#download.
2. Hacer click en **Download** en la sección que dice *RStudio Desktop*.
3. Hacer click en **Download RStudio for Windows** si aparece el botón.
4. Si no aparece el botón, buscar *Windows 10/11* (o el que corresponda) en la lista **All Installers** y guardar el archivo ejecutable.
5. Correr el archivo .exe y seguir las instrucciones de instalación.

Mac

1. Primero visitar https://www.rstudio.com/products/rstudio/download/#download.
2. Hacer click en **Download** en la sección que dice *RStudio Desktop*.
3. Hacer click en **Download RStudio for Mac** si aparece el botón.
4. Si no aparece el botón, buscar *macOS 10.15+* (o el que corresponda) en la lista **All Installers** y guardar el archivo ejecutable.
5. Correr el archivo .dmg y seguir las instrucciones de instalación.