Nota 1

Santiago Casanova y Ernesto Barrios

Cómo Instalar R

En esta primera nota vamos a explicar paso a paso cómo se debe instalar R en la máquina. Para la parte de instalación se cubre a fondo en las secciones **Windows** y **Mac** mientras que para **Linux** se presenta el camino básico sin entrar a detalles para las diferentes distribuciones.

Windows

1. Primero visitar http://www.r-project.org.
2. Hacer click en **download R**.
3. En la lista de mirrors, buscar *https://cran.itam.mx/* y hacer click.
4. Hacer click en *Download R for Windows* y guardar el archivo ejecutable.
5. Correr el archivo .exe y seguir las instrucciones de instalación.

Mac

1. Primero visitar http://www.r-project.org.
2. Hacer click en **download R**.
3. En la lista de mirrors, buscar *https://cran.itam.mx/* y hacer click.
4. Hacer click en *Download R for macOS*.
5. Hacer click en el link del archivo de la versión más reciente de R.
6. Correr el archivo .pkg y seguir las instrucciones de instalación.

Linux

1. Primero visitar http://www.r-project.org.
2. Hacer click en **download R**.
3. En la lista de mirrors, buscar *https://cran.itam.mx/* y hacer click.
4. Hacer click en *Download R for Linux*.
5. Selecciona tu distribución de Linux y sigue las instrucciones para instalar desde la terminal.

¿Qué es R? Breve Historia y Resumen

Sus Inicios

R fue implementado inicialmente por dos profesores de la universidad de Auckland, Robert Gentleman and Ross Ihaka, en la década de los 90. El nombre viene de que Gentleman y Ross se basaron en el lenguaje de análisis estadístico llamado S que surgió en los 80. Al ser una "nueva versión" de S, decidieron llamarle **R**[1].

¿Qué es?

R es un lenguaje intérprete.¿Qué significa esto? Esta afirmación se puede dividir en dos partes para analizar individualmente.

[1] https://mran.microsoft.com/documents/what-is-r

1. R es un lenguaje:

- Esto significa que el trabajo en R se tiene que hacer a través de una serie de comandos, funciones y scripts con una sintaxis específica.

2. Es un lenguaje intérprete:

- Una vez que el usuario escribe sus comandos, funciones y scripts, R da una respuesta. Ya sea un resultado, la creación de un objeto, una figura gráfica o bien un mensaje de error.

El concepto Open Source

R es completamente gratis para usar y es mantenido por un grupo de alrededor de 20 desarrolladores. Además existe una gran comunidad que agrega y construye sobre el R básico (conocido como base R en inglés) a través del desarrollo de paquetes de ayuda para distintos usos. Estos paquetes pueden incluir desde funciones para leer archivos csv hasta métodos estadísticos complejos. Esta gran variedad de paquetes y bibliotecas se distribuye y regula a través de CRAN (*The Comprehensive R Archive Network*) y esta organización de recursos hace que R sea cada día más poderoso como lenguaje y herramienta.

Abrir R por primera vez

Ahora que ya está instalado R en la computadora, podemos hacer una prueba de uso. Localiza donde está guardado el programa y ábrelo. Esto abrirá la consola de R donde se pueden iniciar a escribir comandos y recibir respuestas. En este ejemplo abrimos la consola y escribimos `print('Hello World')`. La consola nos regresa la impresión "Hello World"

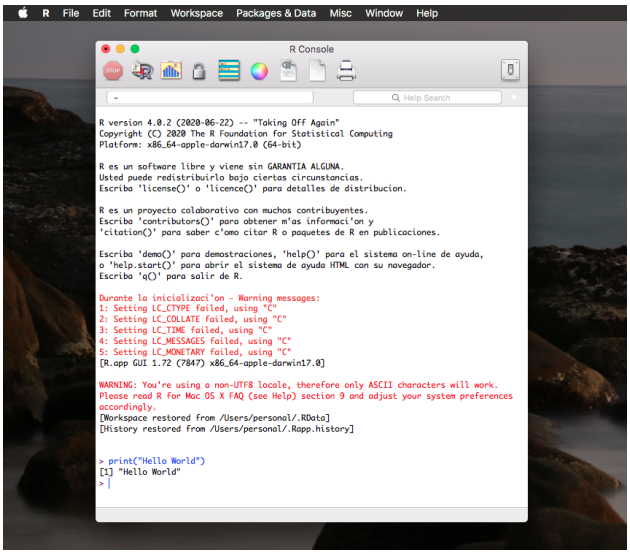


Figure 1: La consola de R con un primer comando

```
print("Hello World")
```

```
## [1] "Hello World"
```

Con esto ya estamos listos para empezar a usar R como una herramienta.

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.

Nota 3 - Funciones y Manipulación de Datos

Santiago Casanova y Ernesto Barrios

Funciones

En esta sección vamos a aprender a definir y usar funciones para facilitar la organización y el uso de código en el futuro.

Definición de funciones

Las funciones se definen como cualquier objeto en `R`. El nombre del objeto seguido del operador de asignación `<-` pero en este caso después viene la palabra clave `function()`.

Dentro de los paréntesis escribimos los arguementos que queremos que reciba nuestra función. Después, entre llaves, se escribe la lógica a seguir de la función.

Veámoslo en acción para entender mejor este concepto.

```
divide_dos_num <- function(num1, num2){ #El código a correr por la función se escribe entre llaves
  num1/num2
}
```

Nótese que, similar a cómo no es necesario utilizar `print()` para que la consola responda, la respuesta de la función no requiere ni `print()` ni `return`.

Si llamamos a nuestra función:

```
divide_dos_num(10,5)
```

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

podemos verificar que funciona como esperamos.

Ejecución Controlada

if, if else, else e ifelse

Es posible que queramos controlar o limitar cuándo corre nuestra función y cuándo no. En nuestro ejemplo no queremos que el segundo número sea cero porque obtendremos `NaN` por respuesta. Utilizaremos la función `if()` para controlar la ejecución.

```
divide_dos_num <- function(num1, num2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(10,5)
```

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

```r
divide_dos_num(10,0)
```

```
## [1] "No es posible dividir entre cero"
```

Si queremos evaluar más de dos condiciones utilizamos `if` en combinación con `else if` y `else` para manejar los casos que no cumplan ninguna de las dos. Es importante notar que el orden de las condiciones si se toma en cuenta. Se evaluará la primera que se cumpla

Veamos un ejemplo:

```r
divide_dos_num <- function(num1, num2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else if(num1 == 0){
    resp <- 0
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(10,5)
```

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

```r
divide_dos_num(0,0)
```

```
## [1] "No es posible dividir entre cero"
```

```r
divide_dos_num(0,10)
```

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

Aunque nuestra segunda llamada a la función cumplía ambas condiciones, solamente obtenemos el resultado de que se cumpla la primera condición (ya que R evalúa de arriba hacia abajo y se detiene cuando encuentra una condición verdadera).

Si queremos evaluar más de una condición en la misma expresión, podemos utilizar los operadores `&` y `|` que simbolizan `AND` y `OR` respectivamente.

Aplicamos esto a nuestro ejemplo anterior.

```r
divide_dos_num <- function(num1, num2 = 2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else if(num1 == 0 | num2 == 1){ #operador OR |
    resp <- 'Es trivial'
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(0,5)
```

```
## [1] "Es trivial"
```

```r
divide_dos_num(0,0)
```

```
## [1] "No es posible dividir entre cero"
```

```r
divide_dos_num(20,1)
```

```
## [1] "Es trivial"
```

Nótese también que podemos definir valores por *default* para los parámetros. En este caso declaramos que `num2` es igual a 2 por default y por lo tanto, si el usuario no proporciona un valor para ese parámetro, la función puede correr con un valor predeterminado.

```r
divide_dos_num(10) #Será dividido entre dos porque no proporcionamos otro parámetro
```

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

Otra forma de controlar la ejecución con una operación lógica en `R` es con la función `ifelse`. Esta condensa las funcionalidades de `if` y `else` en una función llamable con 3 parámetros. El primer parámetro es la expresión a evaluar, el segundo el resultado en caso de que la expresión sea `TRUE` y el tercero el resultado en caso de que la expresión sea `FALSE`.

Esta función es especialmente útil para darle valor a una variable u obtener una respuesta rápida a partir de una condición.

```r
pos_neg <- function(numero){
  ifelse(numero > 0, 'positivo', 'negativo')
        #expresión    #resultado T #resultado F
}

pos_neg(-2)
```

```
## [1] "negativo"
```

```r
pos_neg(3)
```

```
## [1] "positivo"
```

Como se podrán haber dado cuenta, en este ejemplo no estamos evaluando todas las condiciones posibles. Si corremos

```r
pos_neg(0)
```

```
## [1] "negativo"
```

nos regresa como resultado que cero es negativo ya que no cumple la condición `x>0`. Para estos casos podemos anidar nuestros `ifelse` para que manejen más de una condición.

```r
pos_neg_cero <- function(numero){
  ifelse(numero == 0, 'cero', ifelse(numero > 0, 'positivo', 'negativo'))
}
```

En este caso evaluamos primero si el número es igual a cero. Si no lo es, regresamos al `ifelse` que habíamos planteado en la función anterior.

```r
pos_neg_cero(-1)
```

```
## [1] "negativo"
```

```r
pos_neg_cero(1231)
```

```
## [1] "positivo"
```

```r
pos_neg_cero(0)
```

```
## [1] "cero"
```

for y while

En esta sección no analizaremos muy a fondo el funcionamiento o propósito de los ciclos `for` y `while` sino que veremos su sintaxis específica en `R`.

el `for` se escribe de la siguiente manera:

```
for(i in c(1,2,3)){# el rango debe ser un vector por el cual pueda correr la i
  print(i)
}
```

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

Recordemos las maneras de escribir rangos que analizamos en la nota anterior. Ya sea con la sintaxis `inicio:fin`, con el uso de una función generadora como `seq()` o bien proporcionando un vector explícito como el rango buscado.

```
#secuencia del 1 al 5
for(j in seq(5)){
  print(j)
}
```

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

```
#secuencia de dos en dos del 3 al 10
for(k in seq(3,10, 2)){
  print(k)
}
```

```
## [1] 3
## [1] 5
## [1] 7
## [1] 9
```

```
#rangos específicos (no necesitan ser numéricos). Estos son útiles cuando tratamos con columnas de un d

for(l in c('opcion1', 'opcion2', 'opcion3')){
  print(l)
}
```

```
## [1] "opcion1"
## [1] "opcion2"
## [1] "opcion3"
```

Por otro lado el while es muy sencillo e intuitivo. Se escribe la función `while()`y como parámetro se proporciona una expresión que tenga como salida `TRUE` o `FALSE`.

Nuevamente, el código a correr se escribe entre llaves (`{}`)

Mientras la condición tenga salida `TRUE` se corre el código dentro de las llaves del while.

```
i = 5
while(i>0){
  print(i)
```

```
  i = i-1
}
```

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

Nótese que en muchos casos es necesario inicializar la condición antes de llamar a la función while.

Pseudo-aleatorios

`R` tiene muchas maneras de generar números aleatorios (o más puntualmente, pseudo-aleatorios). La forma más básica es con la función `runif()`, la cual selecciona n números aleatorios en un rango dado, todos con la misma probabilidad (es decir, uniforme) de ser seleccionados.

Por default, el rango es 0 a 1 pero este puede ser modificado.

La sintaxis es: `runif(cantidad, mínimo, máximo)`

```
# 5 números del 0 al 1
runif(5)
```

```
## [1] 0.50983629 0.41249484 0.07316865 0.22262843 0.23699363
```

```
# 5 números del 3 al 8
runif(5, 3, 8)
```

```
## [1] 6.718136 5.263347 7.926920 7.054992 4.442439
```

Si quisiéramos convertirla en una función discreta podemos aplicar la función `round()` alrededor de `runif()`

```
round(runif(5,3,8))
```

```
## [1] 7 8 3 4 4
```

Nótese que aunque en ambos casos usamos los mismos argumentos en la función, obtuvimos resultados diferentes (aparte de que los segundos están redondeados). Si quisiéramos obtener los mismos números cada vez que corremos el experimento podemos recurrir a la función `set.seed()`.

En su uso más básico, la función `set.seed()` recibe un parámetro numérico llamado semilla que hace que las funciones sean repetibles. No importa cuándo o dónde se corra la función pseudo-aleatoria, si tienen la misma semilla antes, obtendremos el mismo resultado.

```
set.seed(13)
round(runif(5,100,112))
```

```
## [1] 109 103 105 101 112
```

Y si lo volvemos a correr:

```
set.seed(13)
round(runif(5,100,112))
```

```
## [1] 109 103 105 101 112
```

La consola nos regresa exactamente lo mismo.

Otra función muy común para simulación pseudo-aleatoria es `rbinom()`. Esta recibe 3 parámetros: el número de ensayos, el número de objetos y la probabilidad de éxito. (Es decir una simulación binomial)

La sintaxis es: `rbinom(n, tamaño de ensayo, probabilidad)`

Para ilustrar esto, imaginemos que se tiene una moneda honesta, vamos a realizar 8 lanzamientos y queremos analizar el número de águilas que obtenemos. El resultado de esta simulación con `rbinom()` se ve así:

```
# rbinom(número de lanzamientos, número de monedas, probabilidad de éxito (obtener águila))
set.seed(205)
rbinom(8, 1, 0.5)
```

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

Vemos que en este caso obtuvimos 5 águilas en nuestro experimento.

Veamos una simulación más compleja. Un equipo de futból, con 11 jugadores, sabe que la probabilidad de que uno de ellos se lesione cada partido es de 5%. Podemos simular el número de lesiones en el calendario de 10 partidos con la función `rbinom()`

```
# rbinom(número de partidos, número de jugadores, probabilidad de lesión)
set.seed(46)
rbinom(10, 11, 0.05)
```

```
##  [1] 0 0 1 0 0 1 1 1 1 2
```

En este caso vemos que se lesionó 1 en el tercer partido, 1 en el sexto, etc. Si queremos obtener el total de lesionados podemos utilizar operaciones de vectores.

```
set.seed(46)
sum(rbinom(10, 11, 0.05))
```

```
## [1] 7
```

Por último analizaremos la función `rnorm()`. Esta toma 3 argumentos: el número de experimentos, la media y la desviación estándar para generar números aleatorios con distribución normal (o también conocida como gaussiana). Veamos un ejemplo.

```
# 5 números con media cero y desviación estándar 1
rnorm(5)
```

```
## [1] -0.6224851  0.4292219  0.7208422  1.6993732  0.2467891
```

Si revisamos la media y desviación estándar de este experimento podemos confirmar nuestros argumentos iniciales (con cierto margen de error por el tamaño de la muestra).

```
vector_normal_estandar <- rnorm(300)

# media cercana a cero
mean(vector_normal_estandar)
```

```
## [1] 0.02438401
```

```
#desviación estándar cercana a 1
sd(vector_normal_estandar)
```

```
## [1] 1.057297
```

Podemos hacer lo mismo utilizando otros parámetros. Por ejemplo obtengamos 8 números con media 5 y desviación estándar 3.

```
rnorm(8,5,3)
```

```
## [1]  9.540575  1.207150 10.519307 -1.347276  6.183360  3.048465  3.984006
## [8]  2.467898
```

Como se podrán haber dado cuenta, r + el nombre de alguna distribución nos permite obtener muestras aleatorias de esa distribución con diferentes parámetros. Ejemplos más avanzados incluyen distribución

exponencial `rexp()`, distribución Poisson `rpois()`, geométrica `rgeom()`, etc.

Manipulación de Datos

Santiago Casanova y Ernesto Barrios

Manipulación de datos

En notas anteriores vimos una introducción a los arreglos o `data.frames`, comparadores lógicos y operaciones con vectores. Todos estos conceptos ahora nos serán útiles para aprender a manipular los datos que tenemos almacenados.

Recordemos cómo se ve el arreglo `mtcars`

```
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
```

Una forma de obtener columnas individuales es utilizando el operador `$` seguido del nombre de la columna. Si queremos que la consola nos regrese la columna `mpg` escribimos:

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

Y el resultado es el vector que forma la columna `mpg`. Al ser un vector le podemos aplicar todas las técnicas y operaciones que ya conocemos para los vectores. Por ejemplo, si quisiéramos obtener el dato en la posición dos escribimos:

```
mtcars$mpg[2]
```

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

Ahora vamos a crear nuestra propia columna. Para hacer esto, usamos la notación del operador `$` pero ahora con un nombre de columna que no exista. Después usamos el operador de asignación `<-` para asignar algo a dicha columna.

```
mtcars$like <- rep(0, nrow(mtcars))
head(mtcars)
```

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

En este caso utilizamos la función `rep()` para repetir el cero n veces donde n es el número de filas que tiene el arreglo `mtcars`. Sin embargo, `R` es un lenguaje con muchas comodidades y podemos asignar solo un cero y automáticamente lo recicla a lo largo de la columna.

```
mtcars$like <- 0
head(mtcars)
```

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

Ahora nos gustaría cambiar algunos valores de esta columna. Para hacer esto seguimos exáctamente el mismo proceso que usamos para modificar vectores. Seleccionamos el elemento que queremos y le asignamos un valor nuevo.

```
mtcars$like[18] <- 1
mtcars$like[12] <- 1
mtcars$like[2] <- 1
mtcars$like[28] <- 1
mtcars$like[20] <- 1
mtcars$like[21] <- 1

mtcars$like
```

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

De la misma forma, al ser un vector, podemos usar todas las técnicas y operaciones que conocemos que funcionan para vectores. Por ejemplo:

```
sum(mtcars$like)
```

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

```
max(mtcars$cyl)
```

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

La primera nos regresa la suma de la columna `like`. Es fácil ver que nos regresará 6 ya que en la sección anterior le asignamos 6 `1` en diferentes posiciones. La segunda nos regresa el valor máximo de la columna `cyl`.

Ahora vamos a analizar cómo podemos utilizar pruebas lógicas para obtener valores de un arreglo. Si corremos la expresión:

```
mtcars$cyl >=8
```

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

Vemos que nos regresa un vector booleano con valores `TRUE` o `FALSE` dependiendo si los elementos del vector escogido `mtcars$cyl` cumplen la condición. Ahora lo que podemos hacer es pasar este vector lógico como argumento dentro de los corchetes del arreglo. Esto nos dará las filas que tengan `TRUE` en nuestra prueba lógica.

```
mtcars[mtcars$cyl >=8, ]
```

```
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb like
```

```
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2    0
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4    0
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3    1
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3    0
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3    0
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4    0
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4    0
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4    0
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2    0
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2    0
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4    0
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2    0
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4    0
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8    0
```

Ponemos la prueba lógica seguida de una coma porque queremos obtener los renglones que cumplan esta condición, como lo vimos en la sección de arreglos de la nota anterior.

Si quisiéramos que nos regrese estas filas pero sólamente una selección de columnas, podemos usar un vector con los índices (o los nombres) de las columnas deseadas después de la coma.

En estos próximos ejemplos agregaremos otra condición para limitar los resultados. Ahora buscamos todas las filas que cumplan que `cyl` sea mayor o igual a 8 y que `disp` sea mayor a 400.

```
#Un vector de índices columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, c(1,4,5)]
```

```
##                      mpg  hp drat
## Cadillac Fleetwood  10.4 205 2.93
## Lincoln Continental 10.4 215 3.00
## Chrysler Imperial   14.7 230 3.23
```

```
#Un rango de índices columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, 2:5]
```

```
##                    cyl disp  hp drat
## Cadillac Fleetwood   8  472 205 2.93
## Lincoln Continental  8  460 215 3.00
## Chrysler Imperial    8  440 230 3.23
```

```
#Un vector con nombres de columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, c('mpg','cyl', 'disp')]
```

```
##                      mpg cyl disp
## Cadillac Fleetwood  10.4   8  472
## Lincoln Continental 10.4   8  460
## Chrysler Imperial   14.7   8  440
```

Si sólo buscamos una sola columna, también se puede utilizar el operador `$` después de los corchetes para indicar que queremos que nos regrese esa columna. Nótese que aún es necesario escribir la coma.

```
mtcars[mtcars$cyl >=8 & mtcars$disp > 400,]$mpg
```

```
## [1] 10.4 10.4 14.7
```

De igual manera podemos notar que cuando seleccionamos más de una columna la consola nos regresa un arreglo, mientras que cuando sólo seleccionamos una columna (ya sea con índice, nombre o el operador `$`) la consola regresa un vector.

Esto es crucial ya que nos permite aplicar todas las operaciones y manipulaciones de vectores que ya conocemos.

Esta sintaxis no sólo sirve para obtener los datos a través de la consola. Naturalmente también podemos asignar estos resultados a una nueva variable. Vamos a crear un *subset* de `mtcars` que sólo incluya las filas con `cyl` igual a 4.

```
cars_4_cyl <- mtcars[mtcars$cyl == 4, ]
head(cars_4_cyl)
```

```
##                mpg cyl  disp hp drat    wt  qsec vs am gear carb like
## Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1    0
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2    0
## Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2    0
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1    1
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2    0
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1    1
```

Nótese que ninguna de las operaciones anteriores había modificado el arreglo `mtcars`. Hasta que no asignemos nada, estas solo son operaciones de salida en la consola.

Ahora usemos lo que sabemos sobre crear columnas y números pseudo-aleatorios para crear una columna `tank` que indique el tamaño del tanque de gasolina de los coches.

```
set.seed(13)
cars_4_cyl$tank <- round(rnorm(nrow(cars_4_cyl), 18, 5))
cars_4_cyl
```

```
##                mpg cyl  disp  hp drat    wt  qsec vs am gear carb like tank
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1    0   21
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2    0   17
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2    0   27
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1    1   19
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2    0   24
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1    1   20
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1    1   24
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1    0   19
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2    0   16
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2    1   24
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2    0   13
```

Estamos creando la columna `tank` con números enteros (gracias a `round()`) con media 20 y desviación estándar 8 (con la función `rnorm()`). Para la cantidad de números aleatorios a generar utilizamos `nrow()` para que la función nos regrese los suficientes para todas las filas de nuestro arreglo.

Veamos el resumen de nuestra nueva columna.

```
summary(cars_4_cyl$tank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   13.00   18.00   20.00   20.36   24.00   27.00
```

Las columnas de un arreglo son vectores del mismo tamaño por lo que podemos hacer operaciones entrada-a-entrada entre ellos. Si queremos calcular la distancia total de cada coche nos basta con multiplicar la columna `mpg` (*miles per gallon*) por nuestra nueva columna `tank` en galones.

```
cars_4_cyl$distancia_maxima <- cars_4_cyl$mpg*cars_4_cyl$tank
summary(cars_4_cyl$distancia_maxima)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   278.2   447.4   518.7   544.6   646.8   729.6
```

Ahora filtremos nuestro arreglo con varias condiciones. Queremos todas las columnas de las filas que cumplan que `mpg` sea mayor a 30 y la distancia máxima sea menor a 400.

```
cars_4_cyl[cars_4_cyl$mpg > 30 & cars_4_cyl$distancia_maxima < 400,]
```

```
##  [1] mpg              cyl              disp             hp
##  [5] drat             wt               qsec             vs
##  [9] am               gear             carb             like
## [13] tank             distancia_maxima
## <0 rows> (or 0-length row.names)
```

Esto nos regresa un arreglo de 11 columnas sin embargo tiene cero filas. Ninguna cumple las condiciones que le pedimos.

A lo largo de esta sección hemos visto las marcas y modelos de los coches a un lado del arreglo, sin embargo no son parte de una columna. Si queremos asignarlas a una columna propia podemos hacer:

```
cars_4_cyl$marca_modelo <- rownames(cars_4_cyl)
head(cars_4_cyl)
```

```
##                  mpg cyl  disp hp drat    wt  qsec vs am gear carb like tank
## Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1    0   21
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2    0   17
## Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2    0   27
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1    1   19
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2    0   24
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1    1   20
##                distancia_maxima   marca_modelo
## Datsun 710                478.8     Datsun 710
## Merc 240D                 414.8      Merc 240D
## Merc 230                  615.6       Merc 230
## Fiat 128                  615.6       Fiat 128
## Honda Civic               729.6    Honda Civic
## Toyota Corolla            678.0 Toyota Corolla
```

Otra forma de ver un arreglo completo es con la función `View()`. En vez de regresar algo a la consola, abre el arreglo en otra pestaña donde lo podemos filtrar y buscar datos individuales a manera de interfaz gráfica.

```
View(cars_4_cyl)
```

Podemos ver que nuestra nueva columna de marca y modelo está ya incluida.

Manipulación de texto

Veremos ahora una serie de funciones para manipular texto (o vectores de texto). Estas son especialmente útiles para la limpieza de columnas de datos.

La primera que analizaremos es `grepl()`. Esta sirve para buscar un patrón de caracteres en un vector. Usemos la columna de marca y modelo como vector ejemplo. La sintaxis es `gsub(patrón, vector)`

```
grepl('Fiat', cars_4_cyl$marca_modelo)
```

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

Obtenemos un vector booleano que por sí solo no nos es muy útil. Sin embargo, este se puede escribir dentro de los corchetes de indexación de un arreglo para obtener un resultado más útil.

```r
cars_4_cyl[grepl('Fiat', cars_4_cyl$marca_modelo), ]
```

```
##            mpg cyl disp hp drat    wt  qsec vs am gear carb like tank
## Fiat 128  32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1    1   19
## Fiat X1-9 27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1    0   19
##           distancia_maxima marca_modelo
## Fiat 128             615.6     Fiat 128
## Fiat X1-9            518.7    Fiat X1-9
```

Ahora vemos que la función nos es útil para buscar datos específicos dentro de una cadena en un arreglo, no solo el dato completo de la columna (Esto se lograría con arreglo[dato == buscado]).

También es necesario en muchas ocasiones reemplazar un patrón por otro en un vector de cadenas. Para esto usamos la función `gsub()`. Funciona de manera similar a `grepl()` pero ahora también recibe el parámetro `reemplazo`. Entonces la sintaxis queda `gsub(patrón, reemplazo, vector)`.

```r
gsub('Fiat', 'Mazda',  cars_4_cyl$marca_modelo)
```

```
##  [1] "Datsun 710"     "Merc 240D"      "Merc 230"       "Mazda 128"
##  [5] "Honda Civic"    "Toyota Corolla" "Toyota Corona"  "Mazda X1-9"
##  [9] "Porsche 914-2"  "Lotus Europa"   "Volvo 142E"
```

En este caso la salida es un vector de cadenas en lugar de uno booleano. Este vector es la versión modificada de nuestro vector original. (Notemos que en realidad el vector original no ha sido modificado, este vector respuesta es aparte).

En muchos casos es necesario separar una cadena (más adelante lo veremos para columnas) en dos o más secciones. Para eso podemos usar la función `strsplit()`. Como lo indica su nombre, separa cadenas. Se usa la siguiente sintaxis: `strsplit(cadena, separador)`, donde el separador es un caracter que indique una separación.

```r
strsplit("palabra1 palabra2", " ")
```

```
## [[1]]
## [1] "palabra1" "palabra2"
```

En este caso queremos separar nuestra cadena en el espacio por lo que escribimos un espacio como parámetro de separador. Nótese que en este caso la salida no es un vector sino una lista. Esto es porque el resultado de una sola separación es inherentemente un vector. Si quisiéramos aplicar la función a todo un vector, necesitamos que la función regrese una "colección de vectores". En este caso es una lista de vectores.

```r
strsplit(c("palabra1 palabra2", "palabra3 palabra4"), " ")
```

```
## [[1]]
## [1] "palabra1" "palabra2"
##
## [[2]]
## [1] "palabra3" "palabra4"
```

Este ejemplo ilustra por qué es necesario que el resultado sea una lista. Sin embargo, esto presenta otros problemas. Supongamos que tenemos una columna que incluye nombres y apellidos y la queremos separar en una columna sólo de nombres y otra solo de apellidos. ¿Cómo logramos esto?

Probemos varias opciones para obtener **solo el apellido** del siguiente vector.

```r
nombre_apellidos <- c("Ana Perez", "Ernesto Barrios", "Santiago Casanova")

# strsplit original
strsplit(nombre_apellidos, " ")
```

```
## [[1]]
## [1] "Ana"   "Perez"
##
## [[2]]
## [1] "Ernesto" "Barrios"
##
## [[3]]
## [1] "Santiago" "Casanova"
```

```
# tomamos la segunda posición
strsplit(nombre_apellidos, " ")[2]
```

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

```
# tomamos el segundo elemento de la lista
strsplit(nombre_apellidos, " ")[[2]]
```

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

Ninguna de estas opciones cumple lo que queremos. No hay manera de obtener solo el segundo resultado sólo con índices por lo que tendremos que explorar más opciones. Cómo primer instinto podemos pensar en usar un `for()` para lograr esto.

```
# Declaramos un vector vacío
apellidos <- c()

for(i in 1:length(nombre_apellidos)){
  apellidos[i] <- strsplit(nombre_apellidos[i], ' ')[[1]][2]
}
```

En este ejemplo estamos tomando el elemento **i** del vector (desde 1 hasta el largo del vector), separándolo por el espacio y, una vez separado, tomando el primer elemento de la lista resultante (`[[1]]`). Ya que tenemos acceso al vector tomamos el segundo elemento. Ese elemento es asignado a la posición **i** de nuestro vector apellidos.

Mientras que sí es una solución viable no es muy eficiente y se presta a errores. En esta sección vamos a aprender a vectorizar operaciones.

Vectorización de Funciones

Para resolver el problema anterior primero tenemos que definir una función que haga lo que queremos. Ya analizamos el problema en el paso anterior entonces lo podemos aplicar de manera casi idéntica. Con la única excepción de que ahora lo pensamos como si la función recibiera un solo elemento a la vez.

```
extrae_apellido <- function(nombre_completo){
  strsplit(nombre_completo, " ")[[1]][2]
}
```

Verificamos que funcione como queremos:

```
extrae_apellido("Enrique Juarez")
```

```
## [1] "Juarez"
```

Ahora utilizaremos la familia de funciones `apply()` para aplicar esta función a todos los elementos de un vector. Primero analizaremos `sapply()` y `lapply()`. La sintaxis para estas es: `*apply(vector, función)`.

```
sapply(nombre_apellidos, extrae_apellido) #La función se escribe sin paréntesis
```

```
##         Ana Perez   Ernesto Barrios Santiago Casanova
##           "Perez"         "Barrios"        "Casanova"
```

```
lapply(nombre_apellidos, extrae_apellido)
```

```
## [[1]]
## [1] "Perez"
##
## [[2]]
## [1] "Barrios"
##
## [[3]]
## [1] "Casanova"
```

La función `sapply()` nos regresa un vector con los resultados que además tiene nombres y estos son los elementos del vector original. Por otro lado, `lapply()` regresa una lista donde cada resultado está en su propio vector.

Declaremos ahora una función más compleja y por lo tanto más útil que la que tenemos actualmente. Esta nos va a permitir indicarle qué caracter usar para el separador y también qué parte del nombre completo queremos extraer. Nótese que vamos a establecer valores predeterminados para estos parámetros.

```
extrae_de_nombre <- function(nombre_completo, separador = " ", posicion = 2){
  strsplit(nombre_completo, separador)[[1]][posicion]
}

# Usando los valores default
extrae_de_nombre("Ana López")
```

```
## [1] "López"
```

```
# Si los especificamos
extrae_de_nombre("Ana López", " ", 2)
```

```
## [1] "López"
```

```
# Extraer el nombre
extrae_de_nombre("Ana López", posicion =1)
```

```
## [1] "Ana"
```

```
# Con diferente separador
extrae_de_nombre("Ana-López", separador = "-")
```

```
## [1] "López"
```

Para usar parámetros con las funciones `lapply()` o `sapply()` se usa la siguiente sintaxis: `*apply(vector, función, parámetro = valor)`.

```
sapply(nombre_apellidos, extrae_de_nombre, posicion = 1)
```

```
##         Ana Perez   Ernesto Barrios Santiago Casanova
##             "Ana"         "Ernesto"        "Santiago"
```

Sin embargo estas funciones solo reciben un parámetro por función. En este caso funciona porque nuestra función tiene valores *default* para los argumentos que no especificamos. Sin embargho, si queremos declarar todos individualmente, es necesario utilizar la función `mapply()`. Funciona igual que las demás funciones de la familia apply pero en este caso recibe argumentos infinitos y por lo tanto cambia la sintaxis: 'mapply(función, arg1, arg2, ... , vector)

```r
nombres2 <- c("Ernesto-J-Barrios", "Santiago-J-Casanova", "Ana-P-Lopez")

mapply(extrae_de_nombre, separador = "-", posicion = 3, nombres2)
```

```
##          -       <NA>       <NA>
##  "Barrios" "Casanova"    "Lopez"
```

Esto concluye la sección de vectorización de funciones.

Nota 5 - Graficación

Santiago Casanova y Ernesto Barrios

Graficación Básica

En esta sección aprenderemos conceptos básicos de visualización de información en la paquetería base de R. La función base para graficar es `plot()`. Esta es una función sobrecargada, lo que significa que reconoce el tipo de información que le proporcionamos y nos imprime el resultado correspondiente.

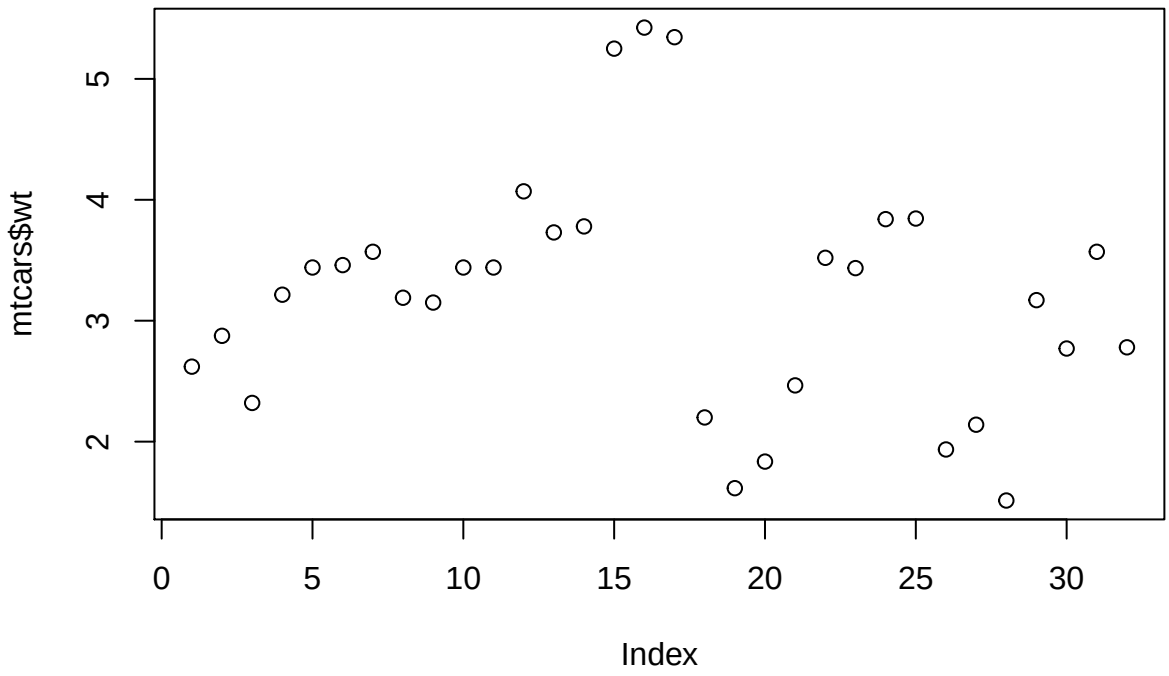
Por ejemplo, siguiendo los ejemplos de la nota anterior (Manipulación de Datos), usemos el dataset `mtcars` para graficar.

```
head(mtcars, 4)
```

```
##                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
```

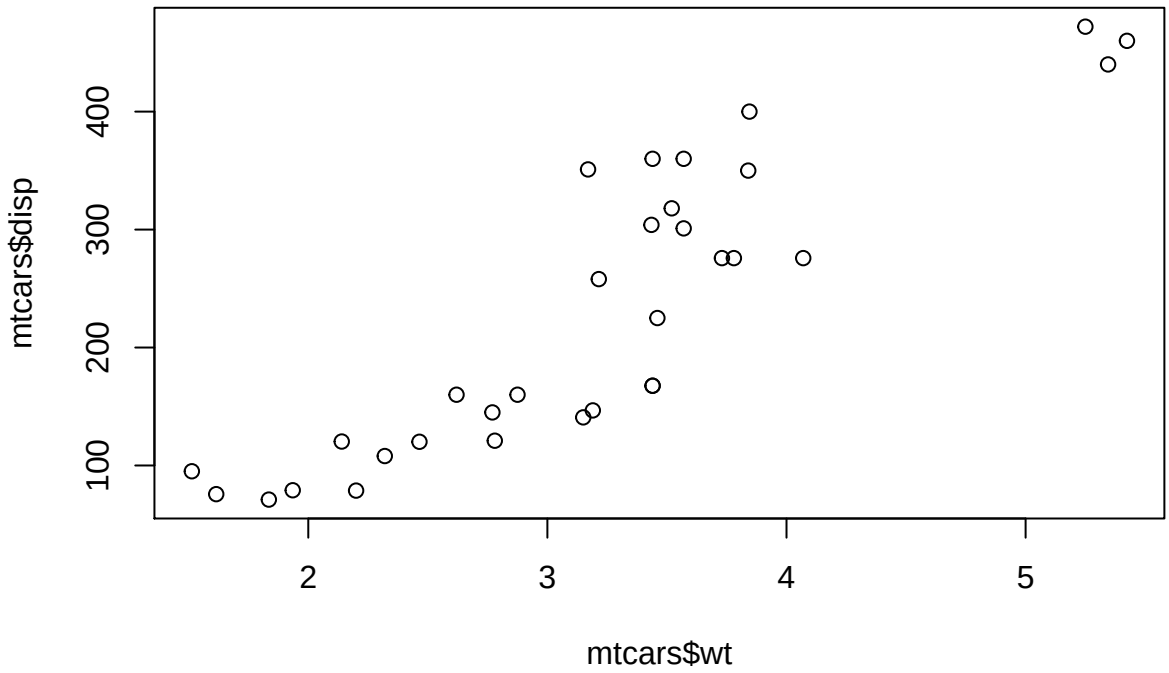
Si le proporcionamos un sólo vector numérico a la función `plot()`, esta nos graficará una figura de dispersión con el vector contra su índice en el vector.

```
plot(mtcars$wt)
```



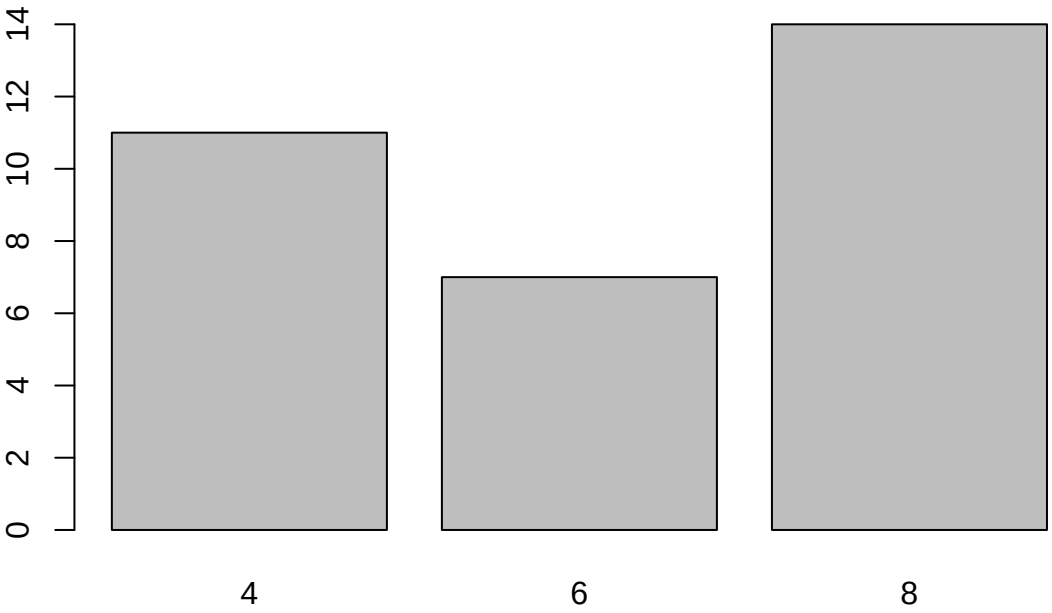
Si le proporcionamos dos vectores nos graficará uno contra el otro en el eje x y eje y.

```
plot(mtcars$wt, mtcars$disp)
```



Ahora veamos qué sucede si le damos un vector de factores a la función `plot()`. Para esto tenemos que convertir la columna `cyl` a datos categóricos en lugar de numéricos.

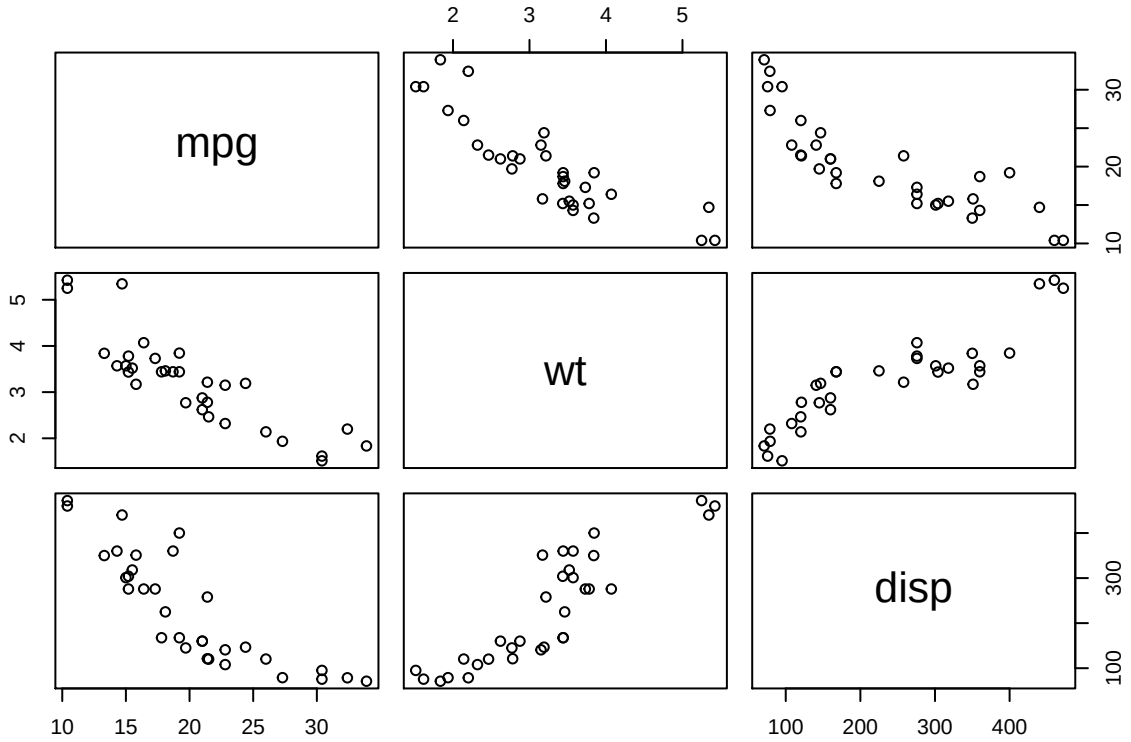
```
plot(as.factor(mtcars$cyl))
```



La salida es un gráfico de barras. Tenemos una barra por cada nivel del factor y la altura representa el número de repeticiones. Nótese que este es diferente al histograma que veremos más tarde.

Ahora veamos qué sucede si le proporcionamos un arreglo a la función `plot()`. En principio esto no suena posible pero veremos que nos da un resultado muy interesante. Primero vamos a restringir nuestro arreglo a solo 3 columnas.

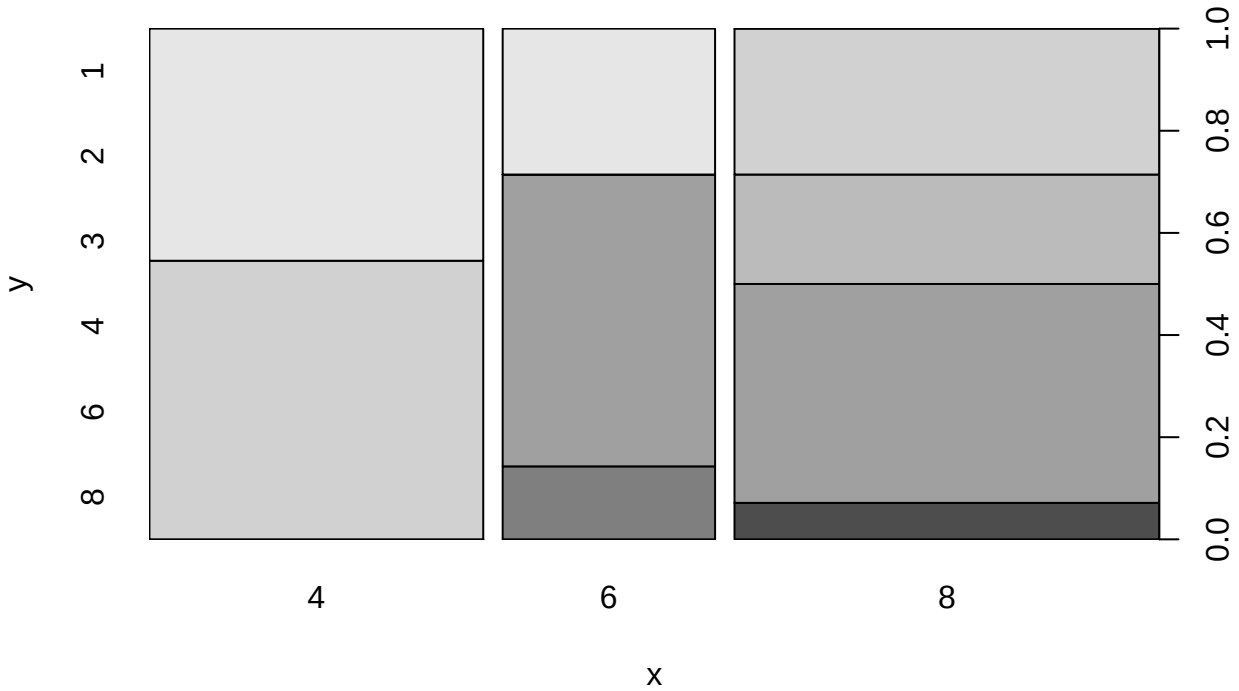
```
arreglo_gráfica <- mtcars[,c("mpg", "wt", "disp")]

plot(arreglo_gráfica)
```



La salida es una matriz de dispersión que compara todas las variables contra todas. Veamos que en términos matriciales es "simétrica".

Ahora analizaremos otras combinaciones. Si le proporcionamos dos factores:

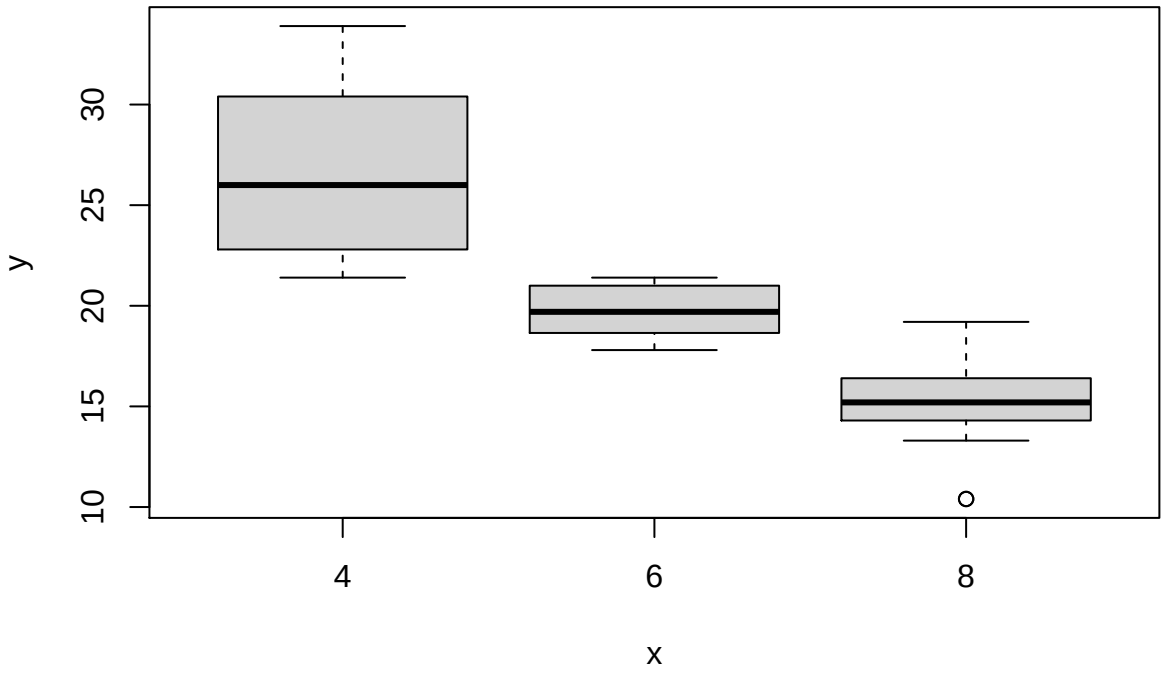
```
plot(as.factor(mtcars$cyl), as.factor(mtcars$carb))
```



Nos da un gráfico de mosaico que combina ambos factores.

Para un factor y una variable numérica:

```
plot(as.factor(mtcars$cyl), mtcars$mpg)
```



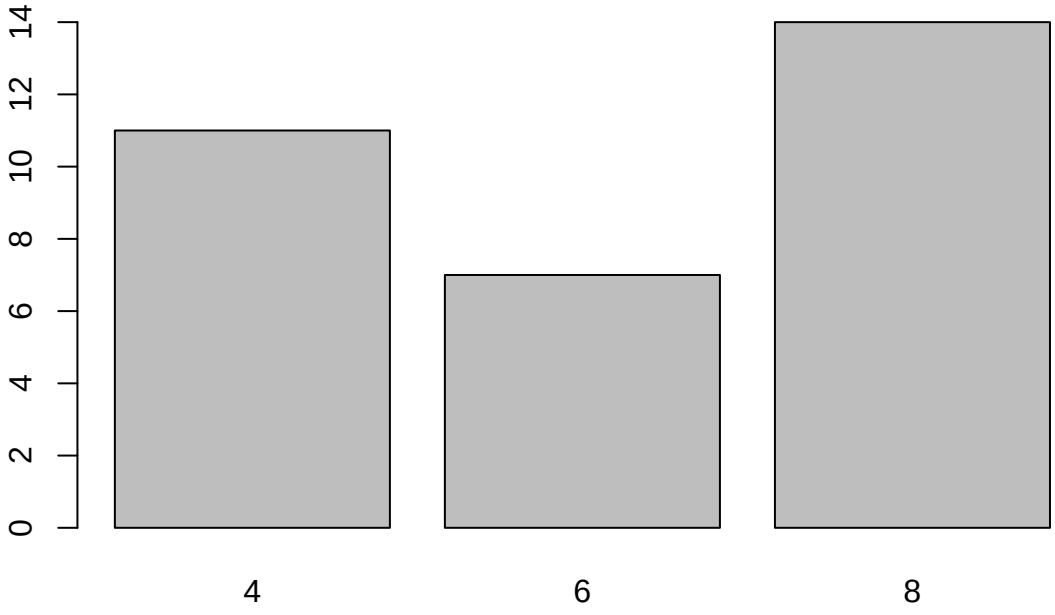
Nos da un gráfico de caja y brazos para la variable numérica, separada por cada nivel del factor.

Gráficos específicos

Estos resultados también se pueden obtener con las funciones específicas `boxplot()` y `barplot()` para dejar en claro cuál es el resultado buscado. Por lo mismo no es necesario convertir a factores los datos antes de graficar en estos casos.

Las siguientes funciones nos regresarán exactamente los mismos resultados que obtuvimos con la función plot.

```
barplot(table(mtcars$cyl))
```

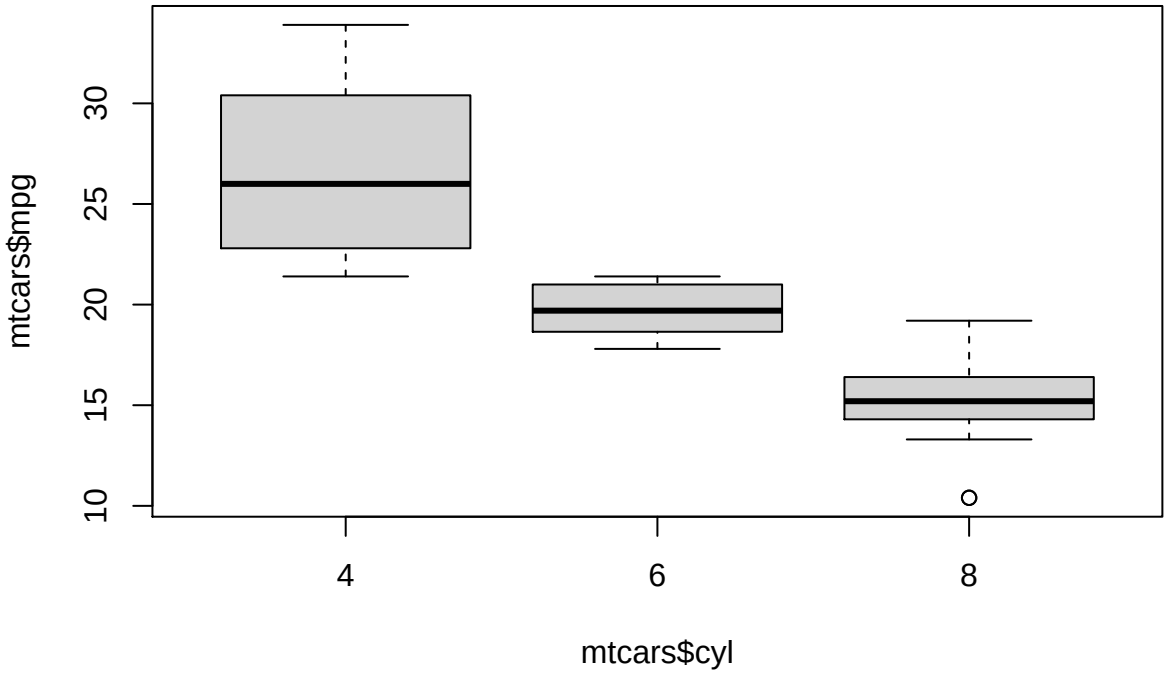


En este caso usamos la función table para que nos diera un recuento de cada valor (con nombres)

```
table(mtcars$cyl)
```

```
##
##  4  6  8
## 11  7 14
```

```
boxplot(mtcars$mpg~mtcars$cyl)
```



Para este resultado indicamos que queremos que una variable sea agrupada por otra con el símbolo `~`. Veamos también que esta versión incluye nombres para los ejes.

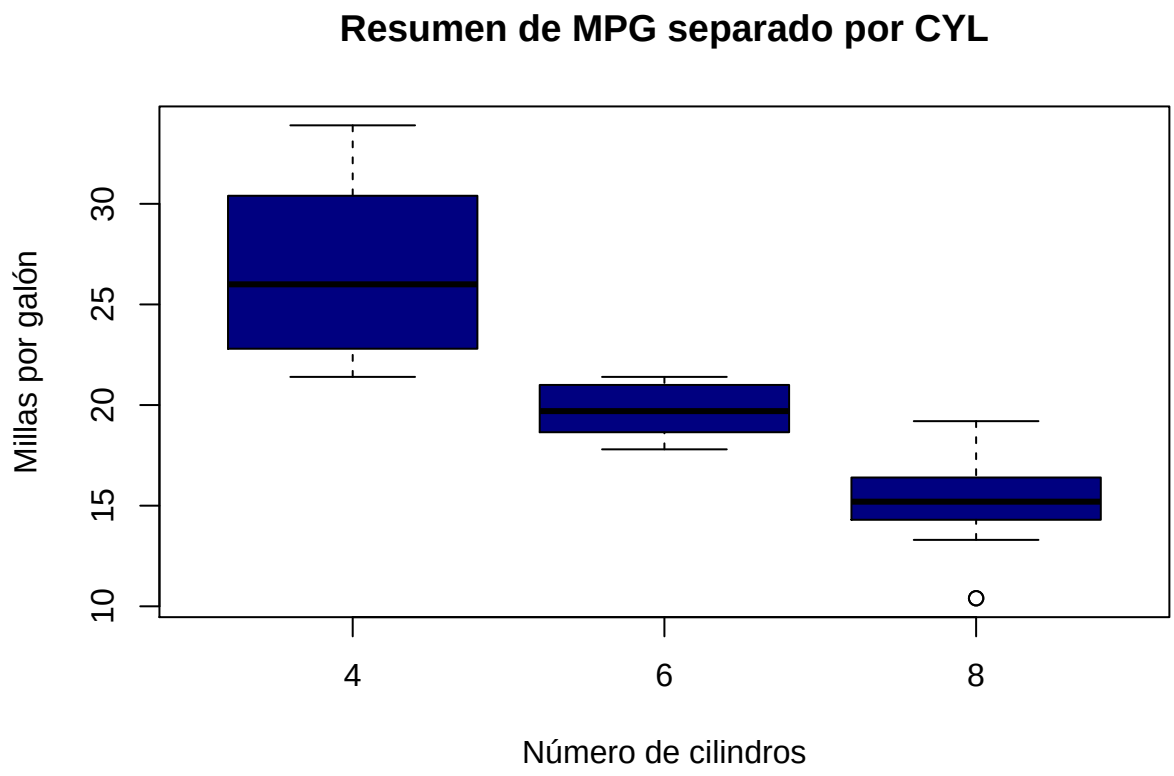
Personalización de gráficos

Todas las funciones de graficación tienen una serie de parámetros que podemos modificar para darle diferentes colores o etiquetas a los datos. Los principales son los siguientes:

`col`: el color de los datos `main`: el título de la figura `xlab/ylab`: el título de cada eje

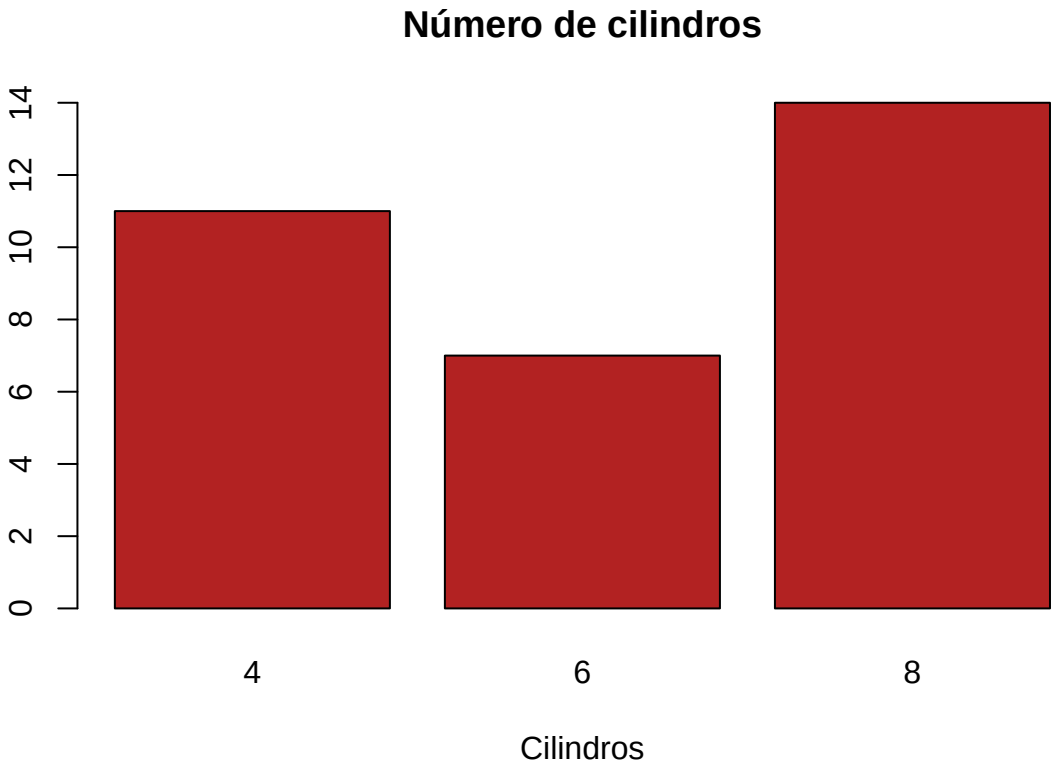
Probémoslos con el gráfico anterior.

```
boxplot(mtcars$mpg~mtcars$cyl, col = "navy", main = "Resumen de MPG separado por CYL", xlab = "Número d
```



Otro ejemplo completo:

```
barplot(table(mtcars$cyl), col = 'firebrick', xlab = 'Cilindros', main = 'Número de cilindros')
```

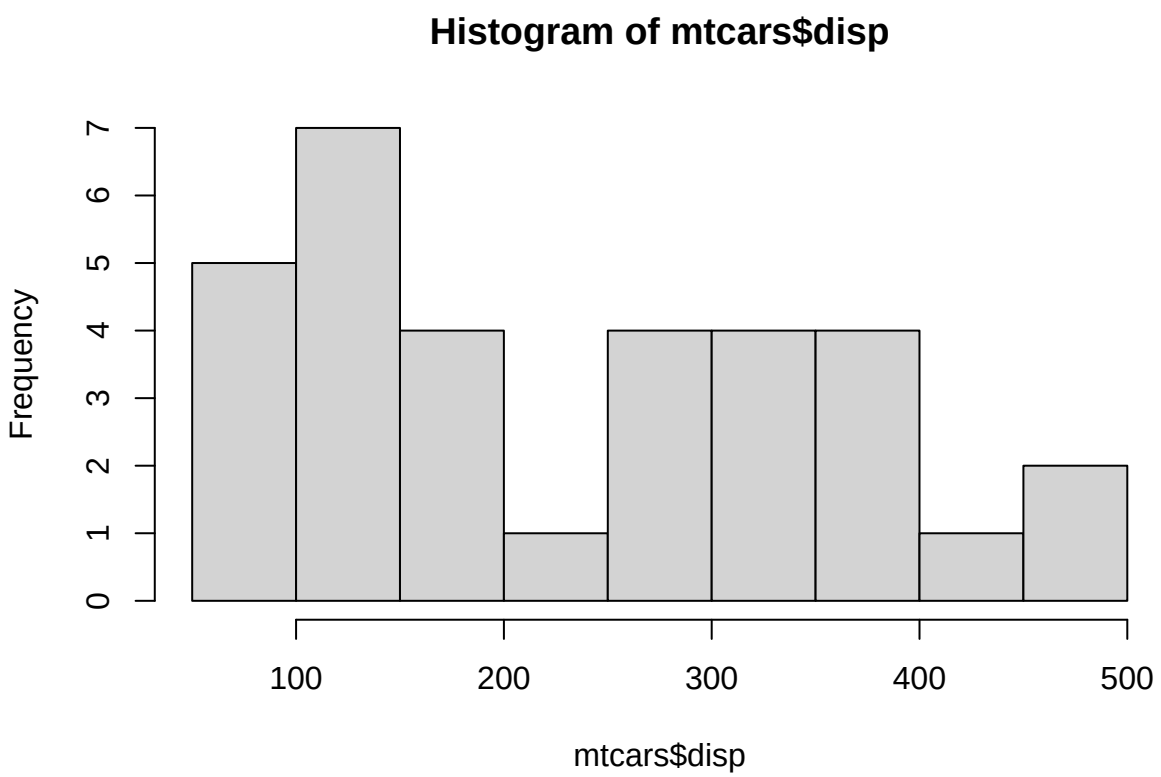


Histograma

El histograma es visualmente similar al gráfico de barras pero cumple otro propósito. El histograma toma una serie de datos (continuos o discretos) y los agrupa en "cubetas" o particiones regulares. El eje "y" representa el número de observaciones que caen en cada cubeta y por lo tanto esta visualización es útil para representar densidades.

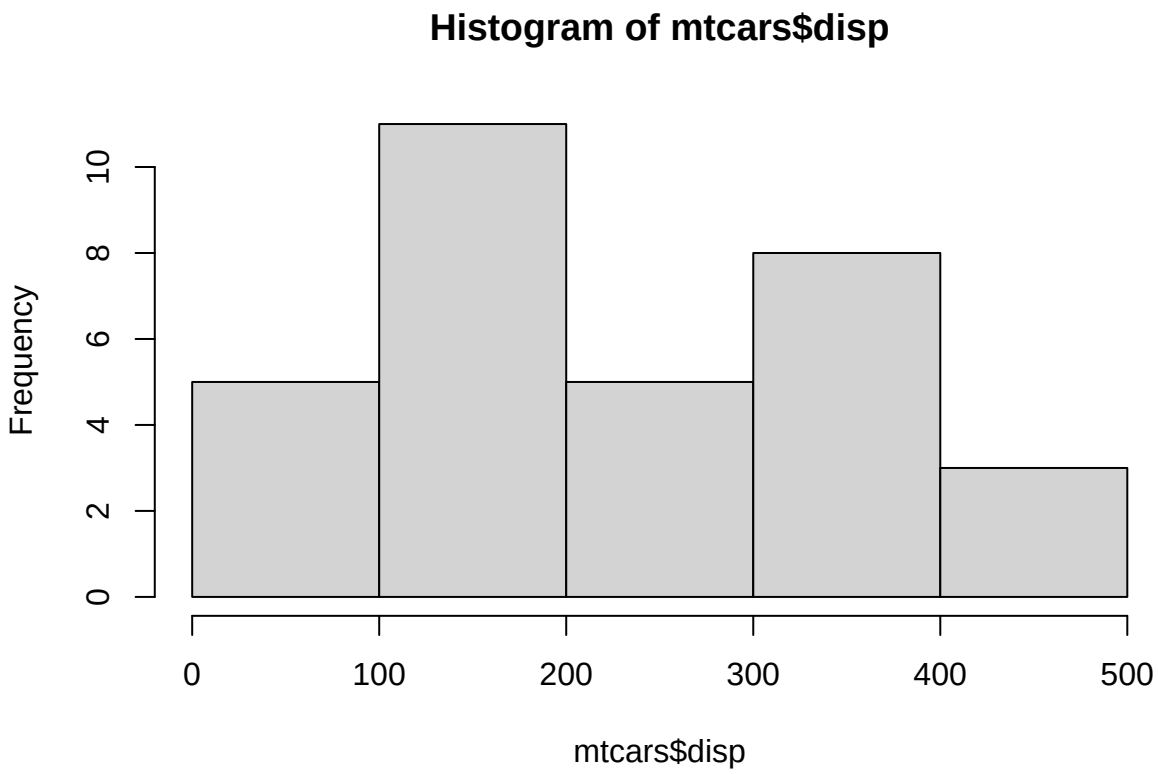
Usamos la función `hist()` para obtener un histograma.

```
hist(mtcars$disp)
```

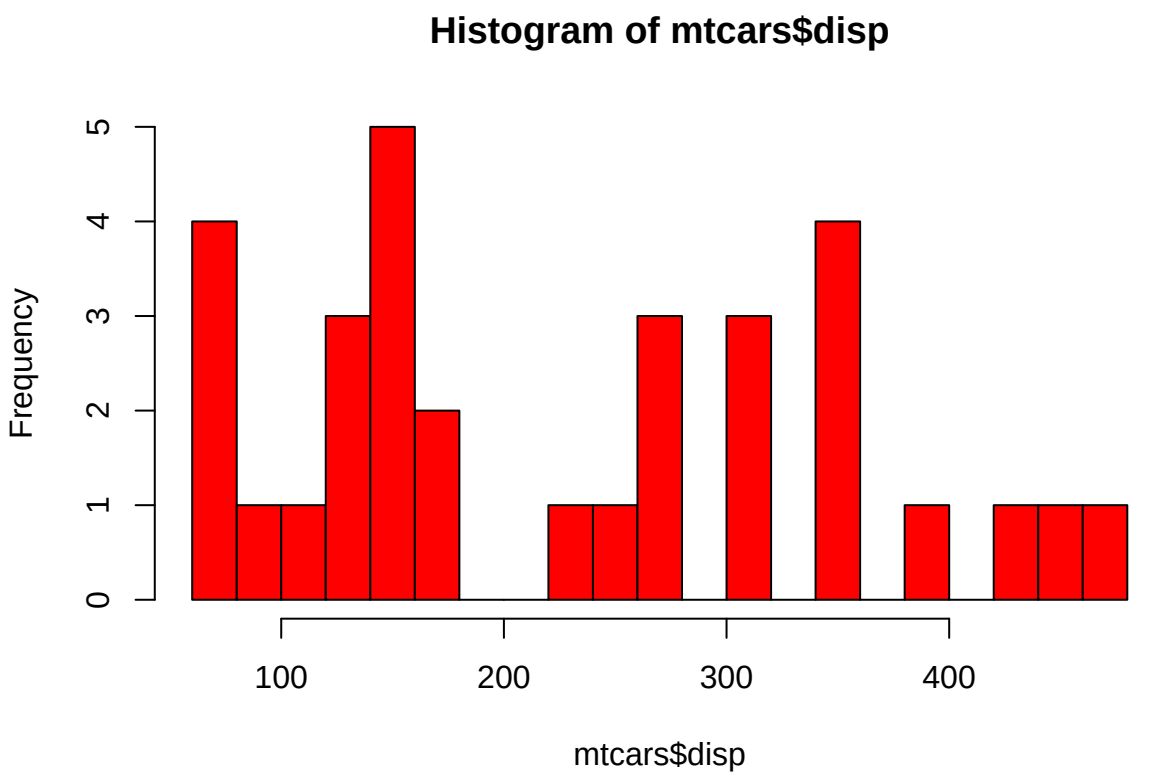


Si sólo le damos nuestro vector como parámetro, `R` ya nos da un título general y para cada eje describiendo el propósito del histograma. Además de los parámetros estéticos (xlab, col, etc.) también podemos pedirle un número diferente de cubetas con `breaks`.

```
hist(mtcars$disp, breaks = 5)
```



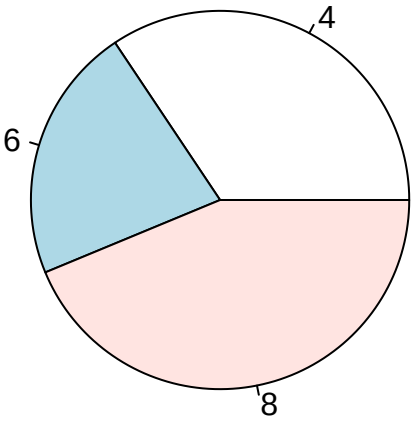

```r
hist(mtcars$disp, breaks = 20, col = "red")
```



Pie

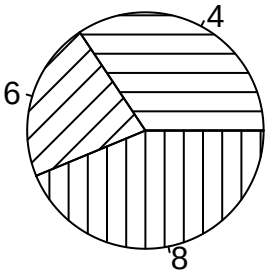
Por último analizaremos cómo crear un gráfico de pie, aunque por lo general no es recomendado utilizarlo en la práctica. Esto es tan sencillo como usar la función `pie()` y proporcionarle un vector de datos discretos agrupados.

```
pie(table(mtcars$cyl))
```



Una característica especial de los gráficos de pie en `R` es la propiedad `density` que, en conjunto con la propiedad `angle`, cambia de colores a líneas cada sección del pie. Además, es modificable el radio,

```
pie(table(mtcars$cyl), density = 10, angle = c(0,45,90), radius = 0.5)
```



Por último nótese que `R` construye el gráfico en el sentido contrario a las manecillas del reloj, empezando a 90º. Esto también se puede cambiar con la propiedad `clockwise`.

Apendice: Ayuda y Ambiente

Santiago Casanova y Ernesto Barrios

Ayuda dentro de R

R es un lenguaje muy bien documentado y tanto las funciones base como las de paqueterías tienen sus respectivas descripciones. Una manera de llegar a esta información es usando la función `help()`, dándole como parámetro la función buscada.

```
help(sapply)
```

En este documento no aparece el resultado porque no es una salida de consola. La documentación aparece en la pestaña `help` de RStudio.

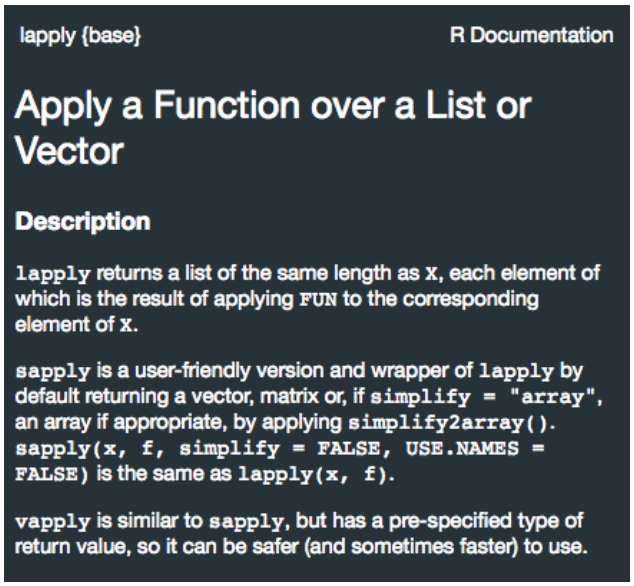


Figure 1: Sección de ayuda para la función sapply

Otra forma de llegar a esta pestaña es escribiendo `?` antes de la función buscada.

```
?table
```

La función `help.search()` funciona de manera similar pero también permite buscar conceptos con palabras clave, no solamente funciones. Esta función busca todos los resultados que contengan la palabra clave (o similares) en el sistema de ayuda de R. Es una especie de Google dentro de R.

Hablando de Google, siempre es una gran opción para buscar ayuda o aprender conceptos nuevos de R. Páginas como StackOverflow o Github son excelentes recursos para aprender y entender más sobre R.

Ambiente R

Para trabajar con archivos es necesario tener la dirección absoluta o relativa (path) de estos, o bien indicarle a la sesión de R a qué directorio queremos que apunte. Esto se hace con la función `setwd()` y la dirección del

directorio.

Las direcciones se escriben de la siguiente manera:

1. En Mac: `~/Documents/carpeta/archivo.pdf` de forma relativa o `Users/usuario/Documents/carpeta/archivo.pdf` de forma absoluta
2. En Mac: `...\documents\archivo.pdf` de forma relativa o `C:\Windows\system\documents\archivo.pdf` de forma absoluta

Por último, se puede usar la función `quit()` o `q()` para cerrar la sesión de `R` desde la consola.