Introducción a R y RStudio

Estructura de datos

Índice

Ideas clave 3

3.1. Introducción y objetivos 3

3.2. Estructuras de datos 3

3.3. Vectores 4

3.4. Factores 17

3.5. Matrices 23

3.6. Arrays 37

3.7. Data frames 40

3.8. Listas 63

3.10. Cuaderno de ejercicios 70

Ideas clave

3.1. Introducción y objetivos

Hasta el momento, las variables que hemos definido son un solo número. Esto no resulta útil para trabajar, por ejemplo, con conjuntos de datos. Ahora vamos a ver que hay diferentes tipos de *contenedores* para almacenar datos múltiples.

Para ello, revisaremos las estructuras de datos más comunes en R y sus principales propiedades, así como la coerción a una estructura específica. Aprenderemos a definir vectores y operar con ellos; a crear matrices, listas y *data frames*; a seleccionar elementos, añadir filas y columnas, etc.

Al finalizar este tema, habrás alcanzado los siguientes objetivos:

- Conocer las estructuras de datos en R y aprender a trabajar con ellas
- Ser capaces de crear las distintas colecciones en R.
- Saber manipular los diferentes conjuntos de datos que aporta R.

En el siguiente video, *"Los objetos de R"*, se introducen los tipos de objetos con que podemos trabajar en R:

 Accede al vídeo

3.2. Estructuras de datos

R es capaz de manejar una variedad de tipos de datos, que se almacenan en diferentes estructuras de datos. Los conjuntos de datos en R se organizan por su dimensión (1, 2 o varias dimensiones) y según si son homogéneas (todos los objetos

son del mismo tipo) o heterogéneas (los objetos que la conforman pueden ser de diferentes tipos). A continuación, mostramos los cinco tipos de estructuras de datos usados con mayor frecuencia:

DIMENSIÓN	Homogénea	Heterogénea
1	Vectores	Listas
2	Matrices	Data frame
n	Array	

Tabla 1: Estructuras de datos

Veamos a continuación las características de cada una de ellas.

En el video *"Estructura de datos"*, se presentan las principales estructuras que podemos utilizar en R para almacenar datos:

Accede al vídeo

3.3. Vectores

En lenguaje R, los vectores incluyen dos tipos de estructuras: **vectores atómicos** y **listas**. Estas se diferencian en cuanto a los tipos de sus elementos: para los vectores atómicos, todos los elementos deben tener el mismo tipo; para las listas, los elementos pueden tener diferentes tipos. En lo que sigue, vamos a referirnos a vectores (por vectores atómicos) y listas, según corresponda.

Un vector corresponde a un **conjunto ordenado de elementos** que reúnen la condición de **pertenecer al mismo tipo de dato atómico**. Los vectores son la estructura de datos más básica en R.

Todos los vectores tienen tres propiedades:

- **Tipo**: que se puede determinar con la función `typeof()`. Un vector tiene el mismo tipo que los datos que contiene. Si tenemos un vector que contiene datos de tipo numérico, el vector será también de tipo numérico. Los vectores son *atómicos*, pues sólo pueden contener datos de un sólo tipo, no es posible mezclar datos de tipos diferentes dentro de ellos.
- **Longitud**: que se puede determinar con la función `length()`. Es el número de elementos que contiene un vector. La longitud es la única dimensión que tiene esta estructura de datos.
- **Atributos**. Un atributo es una parte de información que se puede adjuntar a un vector o cualquier objeto de R. El atributo no afectará a ningún valor del objeto. Podríamos denominar a los atributos como *metadata* que describen características de los datos que contienen. Los atributos más comunes de un vector atómico son los nombres, dimensiones y clases.

Como los vectores son la estructura de datos más sencilla de R, datos simples como el número 3, son en realidad vectores. En este caso, un vector de tipo numérico y largo igual a 1.

Si comprobamos de nuevo el contenido de algunas de las variables que creamos en los ejemplos en el Tema 2, notaremos la indicación [1], previa al valor asignado, indicando que el primer elemento del vector corresponde al primer elemento del renglón que se muestra.

```
r <- 3
r
## [1] 3
```

Esto es porque R, por defecto, crea vectores de longitud 1 para nuestra variable.

Podemos verificar que un objeto es un vector utilizando la función `is.vector()`:

```
is.vector(r)
## [1] TRUE
```

3.3.1. Creación de vectores

¿Y cómo hacemos si queremos vectores de mayor longitud?

Bueno en principio debemos decir que en R hay varias formas de crear vectores: utilizando secuencias, mediante generadores de datos aleatorios o uniendo dos o más vectores. Veamos cada una de ellas.

Usando la función `c()` (combinar o concatenar)

Esta forma de crear vectores de mayor longitud o introducir nuevos elementos al vector ya creado, concatena varios elementos del **mismo tipo**. Si los elementos que intentamos combinar pertenecen a distintos tipos de datos, la función realiza automáticamente la **coerción** de elementos siguiendo la jerarquía `logical > integer > numeric > complex > character` (recordar, de las más restrictivas a las más flexibles). Esto puede ser modificado por la función `as.`, como ya se lo hemos mencionado anteriormente.

```
x <- c(1, -2, 3.5)
y <- c('Lunes','Martes','Miércoles','Jueves','Viernes')
```

Como vemos, los vectores pueden ser no numéricos. La única restricción es que los objetos a concatenar **deben ser del mismo tipo**. Pero ¿qué pasa si intentamos concatenar objetos que no son del mismo tipo?

```
z <- c(2, '3', 4)
w <- c(1>2, 3)
z
## [1] "2" "3" "4"
class(z)
## [1] "character"
w
## [1] 0 3
class(w)
## [1] "numeric"
```

R automáticamente transforma las componentes a un mismo tipo. En el caso del vector `z`, los elementos que queremos concatenar son de tipo numérico y carácter. Luego (otra vez recordar de lo más restrictivo a lo más flexible), la coerción se produce al tipo carácter mientras que, en el vector `w`, las componentes son de tipo lógico (`1>2` es `FALSE`) y numérico. Luego la coerción es al tipo numérico.

Observación: Al crear vectores de tipo `character`, obligatoriamente debemos utilizar las comillas al entrar cada uno de sus elementos. En efecto:

```
y <- c('Lunes','Martes','Miércoles','Jueves',Viernes) # si nos olvidamos las comillas...
Error: object 'Viernes' not found
```

Como lo mencionamos, también podemos crear un vector concatenando dos vectores que ya existen:

```
s <- c(x,w)
s
## [1]  1.0 -2.0  3.5  0.0  3.0
```

Merece la pena destacar que el orden en la concatenación es importante:

```
s2 <- c(w,x)
s2
## [1]  0.0  3.0  1.0 -2.0  3.5
s == s2
## [1] FALSE FALSE FALSE FALSE FALSE
```

Creando sucesiones de números

En R se pueden crear secuencias de números de distintas manera, entre otras, usando el operador `:` o las funciones `seq()` y `rep()`.

Las secuencias creadas con `:` son **consecutivas con incrementos o decrementos de 1**. Estas secuencias **pueden empezar con cualquier número**, incluso si este es negativo o tiene cifras decimales

```
# Secuencia de 2 a 8 con números enteros
2:8
## [1]  2 3 4 5 6 7 8

# También podemos crear la secuencia de 8 a 2
```

```
8:2
## [1] 8 7 6 5 4 3 2

# empezar con negativos
-2:4; -2:-6
## [1] -2 -1  0  1  2  3  4
## [1] -2 -3 -4 -5 -6

# o con decimales
23.45:27
## [1]  23.45 24.45 25.45 26.45
```

Para crear vectores a partir de sucesiones de números con incrementos o decrementos diferentes a la unidad, utilizamos la función `seq()`:

```
# Sequencia de 2 a 8 con pasos de 2
seq(from = 2, to = 8, by = 2)
## [1] 2 4 6 8

# Sequencia de longitud 9 desde 2 hasta 8
seq(from = 2, to = 8, length.out = 9)
## [1] 2.00 2.75 3.50 4.25 5.00 5.75 6.50 7.25 8.00
```

Finalmente, la función `rep()` nos permite crear vectores mediante la repetición de elementos:

```
# Repite el número uno cinco veces
rep(1, 5)
# [1] 1 1 1 1 1

# también otros tipos de datos
rep(TRUE, 3)
## [1] TRUE TRUE TRUE
rep('hola',2)
## [1] "hola" "hola"

# repetir la secuencia 7, 2 tres veces
rep(c(7,2), 3)  # explicitamente rep(c(1,2),times=3)
## [1] 7 2 7 2 7 2

# repetir 7 tres veces y luego el 2 tres veces
rep(c(7,2), each = 3)
## [1] 7 7 7 2 2 2

# también nos puede interesar repetir un número distinto de veces cada elemento
rep(c(7,2), times = c(2,3)) # dos veces el 7 y tres veces el 2
## [1] 7 7 2 2 2

# podemos también indicar la longitud que queremos
```

```
rep(1:4, each = 2, length.out = 5)  # devuelve los primeros 5 elementos de
rep(1:4, each = 2)
## [1] 1 1 2 2 3

rep(1:4, each = 2, length.out = 11) # los 8 elementos de rep(1:4, each = 2)
y completa hasta obtener 11 elementos recomenzando el ciclo
## [1] 1 1 2 2 3 3 4 4 1 1 2
```

A partir de la función vector()

La función vector() crea un vector del tipo y longitud que se especifican como argumentos en el momento de su declaración:

```
vector(mode = "numeric", length = 5)
## [1] 0 0 0 0 0
vector(mode = "integer", length = 5)
## [1] 0 0 0 0 0
vector(mode = "logical", length = 5)
## [1] FALSE FALSE FALSE FALSE FALSE
vector(mode = "character", length = 5)
## [1] "" "" "" "" ""
```

Otra posibilidad es hacer uso de las funciones *wrapper* (del inglés, envoltorio o contenedoras, es una función que llama a una o varias funciones, unas veces únicamente por convenio y otras para adaptarlas con el objetivo de hacer una tarea ligeramente diferente) que existen para cada tipo de datos. Las siguientes instrucciones son equivalentes a las anteriores:

```
vector_numeric <- numeric(5)
vector_integer <- integer(5)
vector_logical <- logical(5)
vector_character <- character(5)
```

En ocasiones, que tal vez ahora nos resulten incomprensibles, será necesario inicializar un *vector vacío* en R que luego, por ejemplo, iremos llenando dentro de un bucle `for`. Para esto podríamos utilizar la función `c()` sin especificar ningún argumento para crear la estructura vacía o, equivalentemente, asignar al vector el valor `NULL`:

```
vacio <- c()  # inicializamos un vector sin elementos
# es equivalente usar vacio <- NULL
# y lo llenamos dentro del bucle for
for(i in 1:5){
   vacio[i] <- 2*i
```

```
}
vacio
## [1]  2  4  6  8 10
```

Esta forma de proceder no es la más recomendada al menos por dos motivos:

1. Debemos tener presente que, si necesitamos llenar un vector vacío, siempre resultará más eficiente preasignar lugar en la memoria creando un vector de la longitud final. Por supuesto que, en estos ejemplos tan sencillos, no notaremos ninguna diferencia, pero para tareas más complejas, la reducción del tiempo de ejecución puede ser notable.
2. Resulta siempre aconsejable inicializar vectores con la longitud final requerida y rellenándolos con valores `NA` y no con un valor numérico particular (como 0). De otro modo puede resultar imposible distinguir cuáles elementos fueron efectivamente ingresados durante el proceso iterativo y cuáles corresponden a la inicialización.

   Retomando el ejemplo anterior, sería:

```
vacio <- rep(NA,5)  # inicializamos un vector con NA
# y lo llenamos dentro del bucle for
for(i in 1:5){
   vacio[i] <- 2*i
}
vacio
## [1]  2  4  6  8 10
```

Mediante un generador de número aleatorios

A veces, necesitamos crear series de datos "inventados" para comprobar, mediante simulaciones, modelos, estudios o hipótesis.

En R hay varias funciones para lidiar con la generación de números aleatorios.

- **La función `sample()`**: es quizás la función más simple para generar una serie de datos aleatorios.

  Su sintaxis es `sample(x, size, replace = FALSE)`, es decir, debemos indicarle como argumento el vector *origen* (`x`) del que se extraen los datos (números, letras,

etc.), después la longitud (`size`) de la sucesión a generar y si se puede o no repetir elemento (`replace`), es decir, si la extracción se realiza con o sin reposición. Como ejemplo, simulemos el experimento de tiran un dado 5 veces. Como vamos a trabajar con números aleatorios, para que se posible reproducir exactamente los resultados, vamos a necesitar usar semillas. Para configurar la semilla del generador de números aleatorios de R y hacer un ejemplo reproducible, primero debes llamar a la función `set.seed()` y establecer la semilla inicial.

```
set.seed(1) # semilla de inicio.
sample(x = 1:6, size = 5, replace = TRUE)
## [1] 1 4 1 2 5
```

Notar que, como estamos simulando tirar un dado, los números se eligen entre 1, 2, 3, 4, 5, o 6 (las caras del dado) y con reposición pues, en los 5 tiros, puede ocurrir más de una vez el mismo resultado (es decir, que salga más de una vez el mismo lado del dado).

- **Funciones de generación aleatoria de números según alguna distribución estadísticas.** Las más conocidas son `runif()` y `rnorm()` que generan secuencias aleatorias de números a través de las distribuciones Uniforme y Normal, respectivamente, pero existen unas cuantas más.

```
set.seed(2)
# 5 valores normales
rnorm(5, mean = 0, sd = 1)
## [1] -0.89691455  0.18484918  1.58784533 -1.13037567 -0.08025176
# 5 Valores uniformes
runif(5, min = 0, max = 1)
## [1] 0.5526741 0.2388948 0.7605133 0.1808201 0.4052822
```

3.3.2. Contar los elementos de un vector

Como lo comentamos al inicio, si queremos saber la cantidad de elementos o cantidad de datos que tiene un vector, la función a utiliza será `length()`.

```
y <- c('Lunes', 'Martes','Miércoles','Jueves','Viernes')
normal <- rnorm(5, mean = 0, sd = 1)
```

```
logico <- c(T,T,F,F)
length(y)
## [1] 5
length(normal)
## [1] 5
length(logico)
## [1] 4
```

En caso de trabajar con vectores de tipo `character`, la función `length()` cuenta la cantidad de *string* o cadenas que hay en el vector (¡como debe ser!). También podría interesarnos contar la cantidad de caracteres en cada *string*. Esto lo conseguimos con la función `nchar()`:

```
nchar(y)
## [1] 5 6 9 6 7
```

lo que significa que la primera cadena tiene 5 caracteres, la segunda 6, y así sucesivamente.

3.3.3. Nombrar elementos de un vector

En algunas ocasiones puede que nos interese nombrar todos o algunos elementos de un vector. Esto no cambia los elementos del vector ni su clase, ni su tipo. Por ejemplo:

```
precioskg <- c(naranja = 2.3, manzana = 3.5, fresa = 8.9, 5.6)
precioskg
## naranja manzana   fresa
##     2.3     3.5     8.9     5.6
class(precioskg)
## [1] "numeric"
```

En caso de que quieras nombrar elementos de un vector ya creado, puedes utilizar la función `setNames()` o `names()`

```
precioskg <- c(2.3, 3.5, 8.9, 5.6)
precioskg
## [1] 2.3 3.5 8.9 5.6
frutas <- c('naranja', 'manzana', 'fresas','')

# Usando setNames()
precioskg_nom <- setNames(precioskg,frutas)
precioskg_nom
## naranja manzana  fresas
##     2.3     3.5     8.9     5.6

# Usando names()
```

```
precioskg <- c(2.3, 3.5, 8.9, 5.6)
precioskg
## [1] 2.3 3.5 8.9 5.6
frutas <- c('naranja', 'manzana', 'fresas','')
names(precioskg) <- frutas
precioskg
## naranja manzana  fresas
##     2.3     3.5     8.9     5.6
```

La diferencia entre usar `names()` y `setNames()` está en que la primera modifica el atributo *names* del vector original, mientras que `setNames()` genera un nuevo objeto, copia del original, con el atributo *names*. Este último comportamiento se llama ***"copiar-al-modificar"*** (del inglés *copy-on-modify*)

3.3.4. Acceder a los elementos de un vector

Acceder a los elementos de un vector implica poder *seleccionar elementos* únicos de un vector, como podría ser el primer o el último elemento, o cualquier subconjunto del vector; pero también reemplazar, cambiar o eliminar algunos elementos. Esto es conocido como *indexing* (del inglés, indexación) y se realiza mediante el uso de los corchetes `[]`. Existen cuatro maneras diferentes de elegir una parte de un vector:

- **Mediante un vector numérico de índices**. Los índices deben ser enteros y todos positivos o todos negativos.

  Seleccionar los elementos con enteros positivos extrae los elementos de las posiciones indicadas:

```
v <- c('uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis')
# Primer elemento
v[1]
## [1] "uno"

# último elemento
n <- length(v)
v[n]
## [1] "seis"

# Tercer y cuarto elemento
v[c(3, 4)]
## [1] "tres" "cuatro"

# Primeros 3 elementos
v[1:3]
```

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

# posiciones pares
v[seq(2, n, 2)]
## [1] "dos" "cuatro" "seis"

# posiciones impares
v[seq(1, n, 2)]
## [1] "uno" "tres" "cinco"
v[-seq(2, n, 2)] # Equivalente
## [1] "uno" "tres" "cinco"
```

Repitiendo una posición, podemos obtener un vector de una longitud más grande que el vector original:

```
v[c(1,1,3,5,5)]
## [1] "uno" "uno" "tres" "cinco" "cinco"
```

Los valores negativos eliminan los elementos en las posiciones especificadas:

```
v[c(-1,-3,-5)]  # que es lo mismo que v[-c(1,3,5)]
## [1] "dos" "cuatro" "seis"
```

Pero no podemos mezclar valores positivos y negativos:

```
v[c(-1,5)]
Error in v[c(-1, 5)] : only 0's may be mixed with negative subscripts
```

- **Por medio de un vector lógico**. En este caso se accederá a todos los valores correspondientes al valor `TRUE`. Este tipo es útil en conjunción con las funciones de comparación. Para ejemplificar, supongamos que tenemos los datos correspondientes a las edades de los empleados de una determinada empresa.

```
set.seed(10)
empleado <- paste0("E",1:25)
edad <- sample(25:70,size=25,replace=TRUE)
# Queremos seleccionar los empleados que tengas más de 60 años
idx <- edad > 60
empleado[idx]

empleado[edad > 60]
## [1] "E1"  "E6"  "E11" "E19" "E21" "E22"
# ¿Que edades tiene?
edad[idx]
## [1] 67 63 66 70 62 63
```

Notar que `edad > 60` es un vector lógico:

```
## [1] TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE
## [23] FALSE FALSE FALSE
```

También podemos combinar más condiciones lógicas:

```
# Queremos seleccionar los empleados que tengan entre 50 y 60 años
idx2 <- edad<=60 & edad>=50
empleado[idx2]
## [1] "E14" "E17" "E23"
# chequeamos
edad[idx2]
## [1] 58 55 51
```

- Si hemos dado nombre a los elementos del vector, podemos seleccionar sus elementos con un vector de tipo `character`:

```
precioskg <- c(naranja = 2.3, manzana = 3.5, fresa = 8.9, 5.6)
precioskg[c('naranja','fresa')]
## naranja   fresa
##     2.3     8.9
```

- Utilizando los corchetes sin especificar ninguna posición, `[]`, obtendremos el vector completo:

```
edad[]
## [1] 67 33 34 36 32 63 43 48 39 39 66 31 34 58 48 37 55 32 70 31 62 63
51 46 42
```

Esta instrucción (corchetes sin argumentos) no es muy útil para acceder a vectores, sin embargo, será de gran utilidad en el acceso a matrices (y cualquier tipo de estructura multidimensional) puesto que nos permite seleccionar todas las filas o columnas. Por ejemplo, si `M` es 2D, `M[1, ]` selecciona la primera fila y todas las columnas, y `M[, -1]` recupera todas las filas y todas las columnas excepto la primera. Volveremos sobre esto cuando estudiemos otras estructuras.

3.3.5. Borrar un vector

Si queremos eliminar un vector en R, podemos utilizar la función `rm()` (*remove*).

```
v <- 1:10
v
## [1]  1  2  3  4  5  6  7  8  9 10
rm(v)
```

```
v
Error: object 'v' not found
```

3.3.6. Vectorización de operaciones

La mayor parte de las operaciones en R están preparados para actuar sobre vectores de datos. Esto significa que, aplicadas a un vector se aplican sobre cada uno de sus elementos y no necesitamos codificar un bucle a fin de aplicar la operación a cada elemento de forma individual. A este proceso le llamamos **vectorización**.

Supongamos que tenemos dos vectores de datos con el peso (en Kg) y la altura (en cm) correspondientes a 100 pacientes de un determinado centro de salud, y necesitamos obtener el IMC mediante la fórmula:

$$IMC = \frac{peso}{altura^2}\frac{[kg]}{[m^2]}$$

Esto significa que precisamos operar con los vectores peso y altura a fin de obtener el IMC (luego de convertir las alturas de cm a m). Veamos (solo como ejemplo ilustrativo) como deberíamos proceder para realizar las operaciones elemento a elemento y como lo hacemos aprovechando la vectorización en R:

```
# Inventemos primero las alturas y los pesos
alturas <- round(rnorm(10000, mean = 160, sd = 5),0)
pesos <- round(alturas/3+rnorm(10000),0)
# ahora debemos halar IMC para cada paciente:
# elemento a elemento:
for(idx in 1:1000){
  IMC[idx] <- pesos[idx] / (alturas[idx]/100)
}
# mediante la vectorización
IMC2 = pesos/(alturas/100)
# comprobamos que los resultados son idénticos
sum(IMC != IMC2)  # cuenta cuántos elementos distintos contienen los vectores
## [1] 0
```

Aunque todavía no hemos estudiado las estructuras iterativas como el bucle `for`, vale la pena introducirlo en este ejemplo para apreciar las ventajas que ofrece la vectorización de las operaciones.

Veamos algunos ejemplos más.

```
x <- 1:5
y <- 9:5
x+y
## [1] 10 10 10 10 10
```

Es decir, la operación se realizó elemento a elemento:

```
# x:    1   2   3   4   5
#       +   +   +   +   +
# y:    9   8   7   6   5
# ------------------------
# x+y: 10  10  10  10  10
```

La multiplicación y división también se realizan de forma vectorizada

```
x*y
## [1]  9 16 21 24 25
x/y
## [1] 0.1111111 0.2500000 0.4285714 0.6666667 1.0000000
```

También la suma de un escalar a un vector o el de un vector por un escalar son operaciones vectorizadas:

```
x+2  # a cada elemento de x le suma 2
## [1] 3 4 5 6 7
x*2   # multiplica por 2 cada elemento de x
## [1]  2  4  6  8 10
```

Al aplicar operaciones relacionales, obtenemos un vector de `TRUE` y `FALSE`, uno para cada elemento comparado:

```
x > 3  # cada elemento de x se compara con 3 y devuelve TRUE si cumple la
condición o FALSE en otro caso.
## [1] FALSE FALSE FALSE TRUE TRUE
# En efecto, miremos x para chequear
x
## [1] 1 2 3 4 5
```

3.4. Factores

Los factores son una clase especial de vectores, con una estructura interna más rica que permite usarlos para clasificar observaciones. En R, los factores se usan para

trabajar con variables categóricas, es decir, variables que tienen un conjunto fijo y conocido de valores posibles. También son útiles cuando queremos mostrar vectores de caracteres en un orden no alfabético.

Podemos pensar en los factores como datos numéricos representados por una etiqueta. ¿Qué significa esto? Supongamos que tenemos un conjunto de datos que representan el sexo de estudiantes de cierta universidad, pero estos se encuentran almacenados (codificados) con los números 1 y 2. El número 1 corresponde a femenino y el 2 a masculino. En R, podemos indicar que se nos muestre, en la consola y para otros análisis, los 1 como femenino y los 2 como masculino. Aunque para nuestra computadora, femenino tiene un valor de 1, pero a nosotros se nos muestra la palabra femenino.

Para ilustrar la diferencia entre vectores y factores, vamos a crear un vector Ciudades con los nombres de algunas ciudades, y a continuación un factor Ciudades.f con el mismo contenido, aplicando a este vector la función `factor()`.

```
Ciudades <- c("Madrid", "Palma", "Madrid", "Madrid", "Barcelona", "Palma",
              "Madrid", "Madrid")
Ciudades
## [1] "Madrid"    "Palma"     "Madrid"    "Madrid"    "Barcelona" "Palma"
## [7] "Madrid"    "Madrid"
Ciudades.f <- factor(Ciudades)
Ciudades.f
#> [1] Madrid    Palma     Madrid    Madrid    Barcelona Palma     Madrid
#> [8] Madrid
#> Levels: Barcelona Madrid Palma
```

Como podemos observar, el factor incluye no solo los valores de la variable categórica correspondiente, sino que también dispone de un atributo especial llamado *levels* (niveles), y cada elemento del factor es igual a un nivel; de esta manera, los niveles clasifican los elementos del factor. Podríamos decir, en resumen, que un factor es una lista formada por copias de etiquetas (los niveles).

Veamos un segundo ejemplo. Supongamos que tenemos datos sobre la cantidad lluvia caída por mes en cierta región. En nuestros datos entonces tendremos al menos

dos variables: una, de tipo numérica, que registra la cantidad de lluvia caída y otra, de tipo cualitativa, que indica el mes de año al que corresponde cada registro. Usar una cadena de caracteres (*strings* en inglés) para guardar esta variable tiene dos problemas principales:

1. Solo hay doce meses posibles y no hay nada que te resguarde de errores de tipeo: Esto quiere decir que, si entre nuestros datos ingresamos:

```
mes <- c("Enero", "Abril", "Juno", "Diciembre", "Junio")
```

no va a saltar ninguna alarma ni R nos enviará ningún mensaje previniéndonos de tal error en `"Juno"` (y, eventualmente, tendremos 13 en lugar de 12 meses!).

2. No se ordena de una forma útil:

```
sort(mes)
[1] "Abril"     "Diciembre" "Enero"     "Junio"     "Juno"
```

Con los factores podemos solucionar ambos problemas. Para crearlo, empezamos definiendo una lista con los niveles válidos:

```
niveles_meses <- c( "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre")
```

Y ahora sí, usando la función `factor()`, creamos factor mes con esos niveles:

```
mes1 <- factor(mes, levels = niveles_meses)
mes1
## [1] Enero     Abril     <NA>      Diciembre Junio
## 12 Levels: Enero Febrero Marzo Abril Mayo Junio Julio ... Diciembre
sort(mes1)
## [1] Enero     Abril     Junio     Diciembre
## 12 Levels: Enero Febrero Marzo Abril Mayo Junio Julio ... Diciembre
```

Cualquier valor no fijado en el conjunto de niveles será convertido a `NA` de forma automática (ver por ejemplo el dato `"Juno"` en el ejemplo anterior).

Si al crear el factor no indicamos cuales son los niveles, se van a definir a partir de los datos en orden alfabético (y se establecerán los valores no repetidos de los datos como los niveles):

```
mes2 <- factor(mes)
mes2
## [1] Enero     Abril     Juno      Diciembre Junio
## Levels: Abril Diciembre Enero Junio Juno
sort(mes2)
## [1] Abril     Diciembre Enero     Junio     Juno
## Levels: Abril Diciembre Enero Junio Juno
```

3.4.1 Crear factores

Como lo hicimos en los ejemplos, para crear un factor, usamos la función `factor()`. Por lo general, comenzamos definiendo un vector clásico (carácter, numérico o lógico) y luego lo transformamos en un factor.

Otra opción es utilizar la función `as.factor()`. La diferencia entre estas funciones es que `as.factor()` convierte el vector en un factor, y toma como sus niveles los diferentes valores que aparecen en el vector, mientras que `factor()` define un factor a partir del vector, y dispone de algunos parámetros que permiten modificar el factor que se crea, tales como:

- *levels*, que permite especificar los niveles e incluso añadir niveles que no aparecen en el vector.
- *labels*, que permite cambiar los nombres de los niveles.

De esta manera, con `as.factor()` o con `factor()` sin especificar `levels`, el factor tendrá como niveles los diferentes valores que toman las entradas del vector, y además aparecerán en su lista de niveles, `Levels`, ordenados en orden alfabético. Si especificamos el parámetro `levels` en la función `factor()`, los niveles aparecerán en dicha lista en el orden en el que los entremos en él.

Volvamos sobre el ejemplo de los meses.

```
mes <- c("Enero", "Abril", "Diciembre", "Junio")
mes2 <- factor(mes)
mes3 <- as.factor(mes)  # esto definirá el mismo factor
mes2
## [1] Enero     Abril     Diciembre Junio
## Levels: Abril Diciembre Enero Junio
mes3
## [1] Enero     Abril     Diciembre Junio
## Levels: Abril Diciembre Enero Junio
```

Ya vimos, en el ejemplo anterior, como incluir niveles que no están incluido en `mes`, utilizando el argumento `levels`:

```
niveles_meses <- c( "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre")
mes1 <- factor(mes, levels = niveles_meses)
```

Supongamos ahora que queremos cambiar los nombres de los niveles. Esto podemos hacerlo de dos maneras:

- Usando el argumento `labels`:

```
labels_meses <- c( "Ene", "Feb", "Mar", "Abr", "May", "Jun",   "Jul",
"Ago", "Sep", "Oct", "Nov", "Dic")
mes4 <- factor(mes, levels = niveles_meses,labels = labels_meses)
mes4
## [1] Ene Abr Dic Jun
## Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
```

- Usando función `levels`: con esta función podemos obtener los niveles de un factor y también permite cambiar los nombres de los niveles de un factor.

```
# obtener los niveles de un factor
levels(mes1)
## [1] "Enero"      "Febrero"    "Marzo"      "Abril"      "Mayo"
## [6] "Junio"      "Julio"      "Agosto"     "Septiembre" "Octubre"
## [11] "Noviembre"  "Diciembre"
levels(mes4)
## [1] "Ene" "Feb" "Mar" "Abr" "May" "Jun" "Jul" "Ago" "Sep" "Oct" "Nov"
## [12] "Dic"

# para cambiar los nombres de los niveles:
levels(mes1) <- c("Ene", "Feb", "Mar", "Abr", "May", "Jun",   "Jul",
"Ago", "Sep", "Oct", "Nov", "Dic")
mes1

# si solo quisiéramos cambiar el nombre de algunos niveles, por ejemplo,
al primer nivel:
levels(mes1)[1] <- "ENE"
mes1
## [1] ENE  Abr  <NA> Dic  Jun
## Levels: ENE Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic

# más de un nivel, ponemos los índice de las posiciones como vector
levels(mes1)[c(2,3)] <- c("FEB", "MAR")
mes1
## [1] ENE  Abr  <NA> Dic  Jun
## Levels: ENE FEB MAR Abr May Jun Jul Ago Sep Oct Nov Dic
```

Debemos prestar especial atención al orden cuando utilizamos la función `levels()` para cambiar los nombres de los niveles. En el ejemplo anterior, la expresión `levels(mes1) <- c("Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic")` significa que se le pide a R que reemplace el primer nivel de `mes1` con `"Ene"`, el segundo con `"Feb"` y así sucesivamente; es decir, `"Enero"` se convierte en `"Ene"`, `"Febrero"` en `"Feb"`, `etc`. Si, en cambio, lo hiciéramos `levels(mes1) <- c("Dic", "Nov",...,` estaríamos mezclando/invirtiendo los meses.

También podemos utilizar la función `levels()` para agregar niveles a un factor existente. Por ejemplo:

```
mes2
## [1] Enero     Abril     Diciembre Junio
## Levels: Abril Diciembre Enero Junio
levels(mes2) <- c(levels(mes2),'Febrero','Marzo')
mes2
## [1] Enero     Abril     Diciembre Junio
## Levels: Abril Diciembre Enero Junio Febrero Marzo
```

Finalmente, con podemos `levels()` borrar o agrupar niveles:

```
mes4
## [1] Ene Abr Dic Jun
## Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
levels(mes4) <- c('I','I','I','P','P','P','V','V','V','O','O','O')
mes4
## [1] I P O P
## Levels: I P V O
```

Hemos hablado sobre el orden de los niveles. En realidad, hay dos tipos de factores: **nominales** y **ordinales**. Hasta ahora sólo hemos considerado los factores nominales, en los que el orden de los niveles realmente no es importante, y si lo modificamos es sólo por razones estéticas o de comprensión de los datos en este caso; la manera más sencilla de hacerlo es redefiniendo el factor con `factor()` y modificando con el argumento `levels` el orden de los niveles. Pero **si el orden de los niveles es relevante** para analizar los datos, entonces es conveniente definir el factor como ordenado. Esto se lleva a cabo con la función `ordered()`, que dispone de los mismos argumentos que `factor()`. Para ejemplificar, supongamos que tenemos como datos las calificaciones, en letras, de 15 estudiantes. Al crear el factor `Notasf`, nos gustaría que

sus niveles estén ordenados, sabiendo que: Suspenso < Aprobado < Notable < Sobresaliente. Esto lo podemos hacer de la siguiente manera:

```
Notas <- c("Sobresaliente", "Suspenso", "Sobresaliente", "Sobresaliente",
"Aprobado",    "Sobresaliente",    "Aprobado",    "Suspenso",    "Suspenso",
"Sobresaliente",   "Sobresaliente",   "Notable",     "Suspenso",   "Notable",
"Sobresaliente")
Notasf <- ordered(Notas, levels = c('Suspenso', 'Aprobado', 'Notable',
'Sobresaliente'))  # creamos el factor ordenado
Notasf
## [1] Sobresaliente Suspenso      Sobresaliente Sobresaliente Aprobado
## [6] Sobresaliente Aprobado      Suspenso      Suspenso      Sobresaliente
## [11] Sobresaliente Notable      Suspenso      Notable       Sobresaliente
## Levels: Suspenso < Aprobado < Notable < Sobresaliente
```

Como podemos observar, R indica el orden de los niveles de un factor ordenado mediante el signo <.

3.5. Matrices

Podemos pensar en las matrices como vectores **multidimensionales**. Al igual que un vector, únicamente pueden contener **datos de un sólo tipo**.

Las matrices son *arreglos rectangulares* de filas (`row`) y columnas (`col`) con información numérica, caracteres o lógica. En R existen varias formas para construir una matriz.

- A partir de la función **`matrix()`**, cuya sintaxis es

  ```
  matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
  ```

  donde

  - **`data`** es el vector que contiene los elementos que formaran parte de la matriz,
  - **`nrow`** es la cantidad de filas y
  - **`ncol`** es la cantidad de columnas de la matriz que se está creando,
  - **`byrow`** es un valor lógico. Si es `TRUE` el vector que pasamos será ordenado por filas, y
  - **`dimnames`** nombres asignado a filas y columnas.

Por ejemplo, para crear una matriz de 3 filas y 5 columnas (es decir, de dimensión 3×5) con los primeros 15 números positivos podemos hacerlo con la siguiente instrucción:

```
M <- matrix(data=1:15, nrow=3, ncol=5, byrow=FALSE) # por columnas
M
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15

M <- matrix(data=1:15, nrow=3, ncol=5, byrow = TRUE)  # por filas
M
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

Si multiplicamos el número de renglones por el número de columnas, obtendremos el número de celdas de la matriz. En el ejemplo anterior, el número de celdas coincide con número de elementos que queremos acomodar en la matriz, así que la operación ocurre sin problemas.

Cuando intentamos acomodar un número diferente de elementos y celdas, ocurren dos cosas diferentes.

- Si el número de elementos que intentamos acomodar es mayor al número de celdas, se acomodarán todos los datos que sean posibles y los demás se omitirán y R emitirá un mensaje avisando de ello.

```
matrix(1:15,nrow=2, ncol = 5) # 10 celdas pero 15 elementos para
acomodar
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
Warning message:
In matrix(1:15, nrow = 2, ncol = 5) :
  data length [15] is not a sub-multiple or multiple of the number of
rows [2]
```

- Cuando el número de celdas es mayor a la cantidad de elementos que queremos acomodar en la matriz, estos se **reciclaran**, es decir, se empezarán a repetir los elementos a partir del primero de ellos. También aquí R nos devolverá una advertencia.

```
matrix(1:7,nrow=2, ncol = 5,byrow=TRUE)   # 10 celdas pero solo 7
elementos para acomodar.
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    1    2    3    # Se reciclan los elementos hasta
                                    # completar la matriz
Warning message:
In matrix(1:7, nrow = 2, ncol = 5, byrow = TRUE) :
  data length [7] is not a sub-multiple or multiple of the number of
rows [2]
```

Veamos un ejemplo en que hagamos uso del argumento `dimnames` para dar nombre a las filas y columnas:

```
M <- matrix(1:15,
  nrow = 3,
  ncol = 5,
  byrow = TRUE,
  dimnames = list(
    c("Supermercado", "Tienda", "Verduleria"),
    c("Naranjas", "Bananas", "Melon", "Fresas", "Sandia")
  )
)
M
##              Naranjas Bananas Melon Fresas Sandia
## Supermercado        1       2     3      4      5
## Tienda              6       7     8      9     10
## Verduleria         11      12    13     14     15
```

- Otro procedimiento para crear matrices es la unión vectores con las siguientes funciones `cbind()` para concatenar columnas o `rbind()` para concatenar filas.

```
x <- 1:5
y <- 6:10
z <- 11:15
rbind(x,y,z)  # concatena filas (es decir, considera que cada vector
              # x, y, z es una fila y organiza una fila debajo de la otra)

##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
## z   11   12   13   14   15

cbind(x,y,z)  # concatena columnas (es decir, considera que cada vector
              # x, y, z es una columna que organiza una al lado de la
otra)
##      x  y  z
## [1,] 1  6 11
## [2,] 2  7 12
## [3,] 3  8 13
```

```
## [4,] 4  9 14
## [5,] 5 10 15
```

Al igual que con `matrix()`, los elementos de los vectores son reciclados para formar una estructura rectangular y se nos muestra un mensaje de advertencia.

```
v1 <- 1:4
v2 <- 1:3
v3 <- 1:5
cbind(v1, v2, v3)
##      v1 v2 v3
## [1,]  1  1  1
## [2,]  2  2  2
## [3,]  3  3  3
## [4,]  4  1  4
## [5,]  1  2  5
Warning message:
In cbind(v1, v2, v3) :
  number of rows of result is not a multiple of vector length (arg 1)

rbind(v1, v2, v3)
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    1
## v2    1    2    3    1    2
## v3    1    2    3    4    5
Warning message:
In rbind(v1, v2, v3) :
  number of columns of result is not a multiple of vector length (arg 1)
```

- Simplemente a partir de un vector al que le añadimos el atributo dimensión:

```
N <- 1:15
N
## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
dim(N) <- c(3,5)  # añadimos el atributo dim para transformar el vector
                  # en matriz
N
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
class(N)   # chequeamos que el objeto ahora es una matriz
## [1] "matrix" "array"
```

Debemos notar que esta forma de obtener una matriz es más restrictiva en cuanto a que no podemos especificar si los elementos se deben acomodar por filas o por columnas, pero que resulta recomendable acostumbrarse a utilizar las formas anteriores.

Observación: Sólo porque resulta más fácil generar vectores numéricos (ya sea aleatorios o mediante las funciones `seq()` o `rep()` o el operador `:`) la mayoría (sino todos) los ejemplos que presentemos corresponderán a matrices cuyos elementos serán números. Pero debería quedar claro que las matrices pueden contener cualquier tipo de dato, siempre y cuando sean todos del mismo tipo. Por ejemplo:

```
# matrices de caracteres
L <- matrix(letters[1:10], nrow = 2, ncol = 5, byrow = TRUE)
L
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "b"  "c"  "d"  "e"
## [2,] "f"  "g"  "h"  "i"  "j"
class(L)
## [1] "matrix" "array"
typeof(L)
## [1] "character"

# matrices de elementos lógicos
TF <- matrix(sample(c(T, F), 12, replace=TRUE),
+              nrow = 3, ncol = 4)
TF
##       [,1]  [,2]  [,3]  [,4]
## [1,] FALSE FALSE FALSE FALSE
## [2,]  TRUE FALSE  TRUE FALSE
## [3,] FALSE  TRUE FALSE  TRUE
class(TF)
## [1] "matrix" "array"
typeof(TF)
## [1] "logical"
```

3.5.1. Dimensión: número de filas y columnas

Aplicada a una matriz, la función `dim()` nos devuelve las dimensiones de la misma. Esto es, un vector de enteros de longitud 2, cuya primera componente indica el número de filas y la segunda el número de columnas.

Volviendo sobre el ejemplo de la matriz M que creamos arriba, tenemos:

```
dim(M)
## [1] 3 5
```

Además con las funciones `nrow()` y `ncol()` podemos conocer el número de filas y columnas, respectivamente:

```
nrow(M)
## [1] 3
ncol(M)
## [1] 5
```

La función `length()` que utilizamos en la sección anterior para obtener la longitud de un vector, también podemos aplicarla a matrices, en cuyo caso nos devuelve la cantidad de elementos en la matriz (que será igual, como ya comentamos, al producto del número de filas por el número de columnas)

```
length(M)
## [1] 15   # 15 = 3*5, es la cantidad de elementos en la matriz
```

3.5.2. Nombre de filas y columnas

Vimos que al crear una matriz podemos indicar, usando el argumento `dimnames`, dar nombres a las filas y columnas. Pues bien, esto también podemos hacerlo si ya tenemos creada la matriz. Para ello debemos utilizar las funciones `colnames()` y `rownames()`.

```
M = matrix(1:15, nrow = 3, ncol = 5, byrow = TRUE)
M
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15

# Añadimos los nombres
rownames(M) <- c('Supermercado', 'Tienda','Verduleria')
colnames(M) <- c('Naranjas','Bananas', 'Melon','Fresas', 'Sandia')
M
##              Naranjas Bananas Melon Fresas Sandia
## Supermercado        1       2     3      4      5
## Tienda              6       7     8      9     10
## Verduleria         11      12    13     14     15
```

Podemos utilizar la función `dimnames()` para obtener una *lista* que contiene dos vectores con los atributos `rownames` y `colnames`:

```
dimnames(M)
## [[1]]
## [1] "Supermercado" "Tienda"       "Verduleria"
## [[2]]
## [1] "Naranjas" "Bananas"  "Melon"    "Fresas"   "Sandia"
```

En ocasiones puede que necesitemos eliminar los nombres de la matriz (o sólo los de las filas o sólo los de las columnas). Para ello bastará con asignar el valor NULL a los nombres de las filas o columnas o ambos, o usando la función `unname()` para borrar todos los nombres.

```
M1 = M2 = M   # creamos copias idénticas de M
# Borrar nombres de las columnas
colnames(M1) <- NULL
M1
##              [,1] [,2] [,3] [,4] [,5]
## Supermercado    1    2    3    4    5
## Tienda          6    7    8    9   10
## Verduleria     11   12   13   14   15

# Borrar nombres de las filas
rownames(M2) <- NULL
M2
##      Naranjas Bananas Melon Fresas Sandia
## [1,]        1       2     3      4      5
## [2,]        6       7     8      9     10
## [3,]       11      12    13     14     15

# Eliminar nombres de filas y columnas a la vez
unname(M)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15

# o bien
dimnames(M) <- NULL
M
##       [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

La diferencia entre usar `unname(M)` y `dimnames(M) <- NULL` es la misma que señalamos para el caso de `names()` y `setNames()`. Es decir, `unname()` crea un nuevo objeto sin los nombres pero sin modificar la matriz original, mientras que con

`dimnames(M) <- NULL` estamos modificando el atributo de nombres de la matriz M, con lo cual, el modificarlo, los perdemos.

3.5.3. Seleccionar elementos de una matriz

La forma más común de crear subconjuntos de matrices es una generalización simple de los subconjuntos en una dimensión (vectores): simplemente usando los corchetes `[,]`, pero proporcionando un índice para cada dimensión (el número de fila(s) y el número de columna(s) que nos interesan), separados por una coma: `[1era_dim, 2da_dim]`. Aplicar el operador `[,]` en blanco (sin indicar alguno de los índices) ahora es útil porque nos permite seleccionar todas las filas o todas las columnas.

Siguiendo con el ejemplo anterior, si quisiéramos seleccionar la entrada que corresponde a `"Naranjas"` en `"Tienda"`, tenemos dos opciones:

- Escribiendo el nombre de la matriz y entre corchetes los nombres de la fila y columnas entre comillas y separados por una coma:

```
M["Tienda","Naranjas"]
## [1] 6
```

- Alternativamente, podemos utilizar los índices correspondientes a la fila y la columna deseada

```
M[2,1]
## [1] 6
```

También podemos seleccionar toda una fila o toda una columna:

```
# seleccionamos la primera fila
M[1,]
## Naranjas  Bananas    Melon   Fresas   Sandia
##        1        2        3        4        5
# o la tercer columna
M[,3]
## Supermercado       Tienda   Verduleria
##            3            8           13

# o los elementos en la primer fila y las columnas 2 y 4
M[1,c(2,4)]
## Bananas  Fresas
##       2       4
```

3.5.4. Añadir y quitar filas y/o columnas de una matriz

Como vimos antes, las funciones `cbind()` y `rbind()` pueden usarse para crear una matriz. Sin embargo, el uso principal de estas funciones es agregar columnas y filas, respectivamente, a las estructuras de datos.

```
# agregar filas o columnas
A <- matrix(1:9,nrow=3,ncol=3)
A
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

# agregamos una columna a A (que debe ser un vector de longitud 3)
A <- cbind(A,c(-1,2,0))
A
##       [,1] [,2] [,3] [,4]
## [1,]    1    4    7   -1
## [2,]    2    5    8    2
## [3,]    3    6    9    0

# agregamos una fila a A (que debe ser un vector de longitud 4)
A <- rbind(A,-2:-5)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   -1
## [2,]    2    5    8    2
## [3,]    3    6    9    0
## [4,]   -2   -3   -4   -5
```

A lo mejor es preciso notar que, el orden en que se agregan las filas o columnas es importante:

```
# importa el orden
A <- matrix(1:9,nrow=3,ncol=3)
# agregamos una columna al final
A2 <- cbind(A,c(-1,2,0))
# agregamos una columna al principio
A3 <- cbind(c(-1,2,0),A)
A2
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   -1
## [2,]    2    5    8    2
## [3,]    3    6    9    0

A3
     [,1] [,2] [,3] [,4]
```

```
[1,]   -1    1    4    7
[2,]    2    2    5    8
[3,]    0    3    6    9
```

Para eliminar columnas o filas, utilizamos los corchetes (operador de indexación) indicando el índice de la columna o de la fila con un -, tal como lo vimos para vectores.

```
# elimino la cuarta columna de A2 y volvemos a tener A
A2[,-4]
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

# o también puedo eliminar las filas 1 y 3 de A3
A3[-c(1,3),]
## [1] 2 2 5 8
```

3.5.5. Operaciones con Matrices

Función `diag()`: Aplicada a una matriz, extrae o reemplaza su diagonal principal. En caso de que la matriz a la que se aplique no sea cuadrada, considera la mayor submatriz cuadrada de la misma y a esa le extrae si diagonal principal.

```
A <- matrix(1:9,nrow = 3,byrow = T)  # matriz cuadrada
A
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
diag(A)
## [1] 1 5 9
B <- matrix(1:12,nrow=4, byrow = T)  # matriz rectangular
B
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
diag(B)
## [1] 1 5 9

diag(B) <- rep(0,3)    # reemplazamos la diagonal principal de B
B
##      [,1] [,2] [,3]
## [1,]    0    2    3
```

```
## [2,]    4    0    6
## [3,]    7    8    0
## [4,]   10   11   12
```

Pero la función `diag()` también puede utilizarse para crear matrices diagonales. Cuando el único argumento que le pasamos es un vector (que puede ser de cualquier tipo -numérico, lógico, carácter-) crea una matriz diagonal[i] cuadrada de tamaño igual a la longitud del vector:

```
diag(c(1,4,2,5))
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    4    0    0
[3,]    0    0    2    0
[4,]    0    0    0    5
diag(TRUE, 3)      # logical
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE FALSE
[2,] FALSE  TRUE FALSE
[3,] FALSE FALSE  TRUE

diag(TRUE, nrow = 3, ncol = 4)     # logical
      [,1]  [,2]  [,3]  [,4]
[1,]  TRUE FALSE FALSE FALSE
[2,] FALSE  TRUE FALSE FALSE
[3,] FALSE FALSE  TRUE FALSE
```

También podemos utilizar `diag()` para crear una matriz identidad[ii]:

```
diag(4)   # matriz identidad de orden 4
[,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
diag(1,nrow=4)    # equivalente
[,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Traza de una matriz cuadrada: En R base no existe ninguna función especial para calcular la traza de una matriz, pero recordando que la traza es la suma de los

elementos en la diagonal principal, podemos hacer uso de las funciones `diag()` y `sum()` para obtener el resultado buscado:

```
A <- matrix(1:16,ncol=4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
sum(diag(A))
## [1] 34   # es igual a  1 + 6 + 11 + 16
```

Matriz traspuesta: Para trasponer un matriz, le aplicamos la función `t()`:

```
A
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15

t(A)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
## [5,]   13   14   15
```

Determinante de una matriz: Podemos calcular el determinante de una matriz usando la función `det()`:

```
A <- matrix(c(100,80,-20,81),nrow=2,byrow=T)
A
##      [,1] [,2]
## [1,]  100   80
## [2,]  -20   81
det(A)
## [1] 9700
```

Inversa de una matriz: En R, la función `solve()` con la matriz como único argumento, calcula su inversa:

```
solve(A)
##         [,1]         [,2]
## [1,] 0.0084 -0.0082
## [2,] 0.0021  0.0103
```

Además, con la función `solve()` podemos resolver sistemas de ecuaciones lineales. Por ejemplo, si disponemos del siguiente sistema de ecuaciones:

$$\begin{aligned} 3x - 2y &= 6 \\ -x + 5y &= -2 \end{aligned}$$

que en forma matricial podríamos expresar de la forma

$$\underbrace{\begin{pmatrix} 3 & -2 \\ -1 & 5 \end{pmatrix}}_{A} \begin{pmatrix} x \\ y \end{pmatrix} = \underbrace{\begin{pmatrix} 6 \\ -2 \end{pmatrix}}_{b}$$

podemos resolverlo en R del siguiente modo:

```
A <- matrix(c(3, -2, -1, 5), ncol = 2, byrow = TRUE)
b <- c(6, -2)
solve(A, b)
## [1]  2 0
```

Operaciones vectorizadas: Tal como lo vimos con vectores, la mayoría de las operaciones en R se vectorizan también al aplicarlas a matrices. Por ello es por lo que, a la hora de operar con matrices, debemos tener mucho cuidado en que es lo que realmente necesitamos hacer.

```
A <- matrix(1:4,nrow=2,byrow=T)
B <- matrix(5:8,nrow=2,byrow=T)
A+2  # a cada elemento de A le suma el escalar 2
##      [,1] [,2]
## [1,]    3    4
## [2,]    5    6
A*2  # a cada elemento de A lo multiplica por el escalar 2
##      [,1] [,2]
## [1,]    2    4
## [2,]    6    8
A+B  # suma las matrices elemento a elemento
##      [,1] [,2]
## [1,]    6    8
## [2,]   10   12
```

Una fuente de posibles errores en el cálculo matricial, cuando se utilizan matrices de la misma dimensión, es utilizar los operadores `*` y `/` ya que multiplican (o dividen) las

matrices término a término (pensar que la división de matrices ni siquiera es una operación que exista en el sentido matemático):

```
A*B  # Cuidado! Multiplica elemento a elemento
##      [,1] [,2]
## [1,]    5   12
## [2,]   21   32

A/A
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
```

Si queremos realizar la multiplicación de matrices (operación en el sentido matemático, siempre y cuando las matrices sean compatibles) debemos utilizar el operador `%*%`

```
A%*%B
##      [,1] [,2]
## [1,]   19   22
## [2,]   43   50

A^2    # eleva cada elemento de A al cuadrado (potencia vectorizada)
##      [,1] [,2]
## [1,]    9    4
## [2,]    1   25
```

Para el caso de la potencia de una matriz, no existe una función en R base para calcularla. Podríamos usar el operador anterior, aplicado sucesivas veces. Pero eso sería ineficiente. Veamos dos alternativas diferentes

Por una parte, puedes hacer uso del operador `%^%` del paquete `{expm}` como sigue:

```
# install.packages("expm")     # si fuera necesario, deberás instalarlo
library(expm)
A2 <- A %^% 2    # esto si es el cuadrado de la matriz A
A2
##      [,1] [,2]
## [1,]   11  -16
## [2,]   -8   27
```

Entonces si queremos halla la matriz A al cubo, la instrucción sería así.

```
A3 <- A %^% 3
A3
##     [,1] [,2]
## [1,]   49 -102
## [2,]  -51  151
```

Por otra parte el paquete `{matrixcalc}` proporciona la función `matrix.power()`:

```
# install.packages("matrixcalc") # si fuera necesario, deberás instalarlo
library(matrixcalc)
matrix.power(A, 2)
##      [,1] [,2]
## [1,]   11  -16
## [2,]   -8   27
```

Podemos comprobar que los resultados anteriores son correctos con el siguiente código:

```
A%*%A
##      [,1] [,2]
## [1,]   11  -16
## [2,]   -8   27
```

3.6. Arrays

Un *array* es una extensión de un vector a más de dos dimensiones o, dicho de otro modo, una matriz de varias dimensiones.

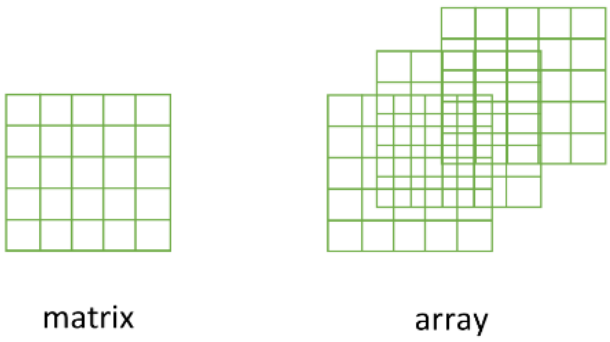

Figura 1: Relación entre una matriz y un *array*. Fuente: Douglas, A. et al. (2022)

Los *arrays* se emplean para representar **datos multidimensionales de un único tipo**. Su uso no es muy común en R y por ello no profundizaremos en su estudio, aunque a veces es deseable contar con objetos n-dimensionales para manipular datos. Como los vectores y matrices, los *arrays* tienen la restricción de que todos sus datos deben ser del mismo tipo, no importando en cuántas dimensiones se encuentren, esto limita sus usos prácticos. En general, es preferible usar listas en lugar de *arrays*, una estructura de datos que además tienen ciertas ventajas que veremos más adelante.

3.6.1. Crear un *array*

Para construir un *array* se usa la función `array()`. Por ejemplo, para crear un arreglo de dimensiones 3 x 4 x 2 con las primeras 24 letras minúsculas del alfabeto se escribe el siguiente código:

```
a1 <- array(data=letters[1:24], dim=c(3, 4, 2))
> a1
, , 1

     [,1] [,2] [,3] [,4]
[1,] "a"  "d"  "g"  "j"
[2,] "b"  "e"  "h"  "k"
[3,] "c"  "f"  "i"  "l"

, , 2

     [,1] [,2] [,3] [,4]
[1,] "m"  "p"  "s"  "v"
[2,] "n"  "q"  "t"  "w"
[3,] "o"  "r"  "u"  "x"
```

Observemos que como resultado obtenemos un arreglo formado por dos matrices de 3 filas y 4 columnas cada una.

Veamos un segundo ejemplo:

```
# creando un arreglo de 'cubos' (2x3x3x2)
a2 <- array(data=1:16,dim = c(2,3,3,2))
a2
## , , 1, 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2, 1
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3, 1
##
##      [,1] [,2] [,3]
## [1,]   13   15    1
## [2,]   14   16    2
```

```
##
## , , 1, 2
##
##      [,1] [,2] [,3]
## [1,]    3    5    7
## [2,]    4    6    8
##
## , , 2, 2
##
##      [,1] [,2] [,3]
## [1,]    9   11   13
## [2,]   10   12   14
##
## , , 3, 2
##
##      [,1] [,2] [,3]
## [1,]   15    1    3
## [2,]   16    2    4
```

Notar que en este ejemplo el resultado es un arreglo de dos cubos, cada uno de los cuales está formado por 3 matrices de 2 filas y tres columnas cada una.

Para obtener la dimensión de un *array*, utilizamos la función `dim()`:

```
dim(a1)
## [1] 3 4 2
dim(a2)
## [1] 2 3 3 2
```

Tal como vimos con matrices, la función `length()` nos devuelve la cantidad de elementos del *array* (que es igual al producto de sus dimensiones):

```
length(a1)
## [1] 24
length(a2)
## [1] 36
```

3.6.2. Acceder a elementos de un *array*

Para acceder a elementos almacenados en un *array*, como vimos con matrices, también utilizamos los corchetes, y dentro de los corchetes, indicaremos las coordenadas del objeto de interés.

```
a1[1, 3, 2]  # El orden es importante. En este caso estamos extrayendo
             # el elemento en la primera fila, tercera columna,
             # de la segunda matriz.
```

```
## [1] "s"
a1[2, 3, 1]  # 2da fila, 3era columna, 1er matriz.
## [1] "h"
a1[2, 1, 3]  # no existen 3 matrices
Error in a1[2, 1, 3] : subscript out of bounds
```

De forma análoga a lo que vimos con matrices, utilizar el operador [, ,] "en blanco" (sin indicación de índices/posiciones a seleccionar) nos permite seleccionar toda una dimensión:

```
a1[, , ]   # selecciona el array completo
a1[, 1, ]  # selecciona la primera columna de cada matriz
a1[, , 2]  # selecciones la 2da matriz
a1[1, , 1] # selecciona la fila 1 de la primera matriz
a1[, 1, 1] # selecciona la columna 1 de la primera matriz
```

3.7. Data frames

Un *data frame* es una estructura de datos bidimensional que permite almacenar datos de diferentes tipos (por ejemplo, numérico, lógico, carácter) y, por lo tanto, son estructuras heterogéneas. Esta estructura de datos es la más usada para realizar análisis de datos y seguro te resultará familiar si has trabajado con otros paquetes estadísticos.

Podemos entender a los *data frames* como una versión más flexible de una matriz. Mientras que en una matriz todas las celdas deben contener datos del mismo tipo, los renglones de un *data frame* admiten datos de distintos tipos, pero sus columnas conservan la restricción de contener datos de un sólo tipo. Así, podemos pensar en un *data frame* como en un conjunto de vectores columnas, eventualmente de distinto tipo, pero todos de la misma longitud.

Conceptualmente, podemos pensar en un *data frame* como una tabla con filas que representan observaciones y con columnas que representan las diferentes variables recopiladas para cada observación.

La instalación básica de R lleva predefinidos algunos conjuntos de datos. Podemos visualizarlos ejecutando la instrucción `data()`, que abrirá una ventana con la lista de los *datasets* a los que tenemos acceso en la sesión actual de R (los que lleva la instalación básica de R y los que aportan los paquetes que tengamos cargados). Uno de los *datasets* más populares que lleva R es el llamado `iris data set`, que contiene las medidas en centímetros de las variables largo y ancho del sépalo y largo y ancho del pétalo, y la especie de 150 flores iris (Fisher, R., 1936). En R, este conjunto de datos de flores iris está recogido en el *data frame* `iris`. Veamos como lucen las primeras seis filas de este *data frame*.

```
head(iris, 4)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
?iris   # para acceder a la información sobre los datos
class(iris)
## [1] "data.frame"
```

Como podemos observar en el ejemplo, cada observación, esto es, cada fila del conjunto de datos tiene ciertas características que son representadas en las cinco columnas restantes. Cada columna consiste en valores del mismo tipo, puesto que se corresponden a vectores: las primeras cuatro columnas contienen valores numéricos (de tipo *double*), mientras que la última columna, *Species*, contienen caracteres que son almacenados como factor.

Los *data frames* también tienen similitud con las listas, puesto que son básicamente *colecciones de elementos*. Sin embargo, el *data frame* es una lista que únicamente contiene vectores de la misma longitud. Por lo tanto, podemos considerarlo un tipo especial de lista y en el podemos acceder a sus elementos del mismo modo que lo hacemos en las matrices o las listas.

Generalmente los *data frames* los creamos al cargar/leer una base de datos (veremos más adelante), pero ahora vamos a crear un *data frame* para ver su estructura.

Para ello, comenzaremos creando los vectores que constituirán las columnas de nuestro futuro *data frame*.

```
altura <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170) # en cm
peso <- c(86, 74, 83, 50, 78, 66, 66, 51, 50, 55) # en Kg
fuma <- c("No", "Si", "No", "Si", "No", "No", "Si", "Si", "Si", "No") #
habito de fumador
sexo <- c("M", "M", "F", "M", "M", "M", "F", "M", "F", "F")
```

Ahora simplemente usamos la función `data.frame()` para construir nuestro data frame, que vamos a asignar a la variable `df`:

```
df <- data.frame(altura, peso, habito = fuma, sexo)
df
##    altura peso habito sexo X1.10
## 1     167   86     No    M     1
## 2     192   74     Si    M     2
## 3     173   83     No    F     3
## 4     174   50     Si    M     4
## 5     172   78     No    M     5
## 6     167   66     No    M     6
## 7     171   66     Si    F     7
## 8     185   51     Si    M     8
## 9     163   50     Si    F     9
## 10    170   55     No    F    10
```

nombre de columnas/variables

nombre/enumeración de las filas

vectores columnas con los datos

Las columnas de un *data frame* siempre tienen nombre. Heredan automáticamente los nombres de los vectores proporcionados (ver por ejemplo columnas 1 y 2 de `df`). Pero también podemos asignar otros nombres (ver columna 3 de `df`). Como todos los nombres, es recomendable que este sea claro, no ambiguo y descriptivo. En el caso de que un vector no tenga nombre y no le asignemos uno manualmente, R le asigna un nombre (ver la 5ta columna de `df`). No es recomendable dejar que R haga esto, ya que a menudo da como resultado nombres irrelevantes.

Las filas se nombran/enumeran automáticamente en orden creciente (1 = primera fila, 2 = segunda fila, etc.). En la práctica, rara vez se necesita cambiar estos nombres.

Algo que debemos tener en cuenta es que, si los vectores que usamos para construir el *data frame* no son del mismo largo, los datos no se reciclarán. Se nos devolverá un error.

```
data.frame(
+   "altura" = c(167,192,173,174),
+   "peso" = c(86,74,83,50),
+   "habito" = c('No','Si','No')    # notar que solo tiene 3 elementos
+ )
Error in data.frame(altura = c(167, 192, 173, 174), peso = c(86, 74, 83,  :
  arguments imply differing number of rows: 4, 3
```

Debemos tener presente que un *data frame* aunque se muestre en forma de matriz, en realidad (internamente) es una lista particular cuyos elementos solo pueden ser vectores de la misma longitud. Podemos ver esto, en parte, al observar la estructura y las características de nuestro objeto `df`:

```
str(df)   # para ver la estructura del objeto
'data.frame':       10 obs. of  5 variables:
 $ altura: num  167 192 173 174 172 167 171 185 163 170
 $ peso  : num  86 74 83 50 78 66 66 51 50 55
 $ habito: chr  "No" "Si" "No" "Si" ...
 $ sexo  : chr  "M" "M" "F" "M" ...
 $ X1.10 : int  1 2 3 4 5 6 7 8 9 10
dim(df)
## [1] 10  5
length(df)   # cantidad de elementos en la lista = cantidad de columnas
## [1] 5
typeof(df)
## [1] "list"
attributes(df)
## $names
## [1] "altura" "peso"   "habito" "sexo"   "X1.10"

## $class
## [1] "data.frame"

## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Como vemos en la salida anterior, los *data frames* tienen atributos de nombres: `names()` nos da los nombres de las columnas/variables y `rownames()` los nombres de las filas/observaciones.

```
names(df)
## [1] "altura" "peso"   "habito" "sexo"   "X1.10"
colnames(df)   # también sirve para obtener los nombres de las columnas
## [1] "altura" "peso"   "habito" "sexo"   "X1.10"
rownames(df)
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

En RStudio podemos ver el *data frame* de forma interactiva, para lo cual tenemos varias opciones:

- ejecutando en la consola la instrucción `View(df)`
- haciendo clic en el icono de la tabla pequeña que se encuentra a la derecha en la fila del objeto `df` en la pestaña *Environment*.

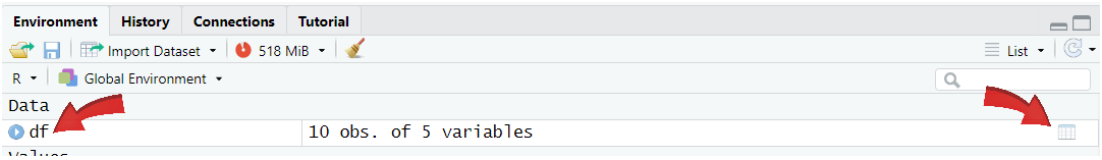

Figura 2: Visualización del data frame desde *Environment.*

- haciendo clic sobre el objeto `df` en la pestaña *Environment.* (ver Figura 1)

Cualquiera de estas tres formas abrirá una ventana en el panel de scripts, como se muestra en la Figura 2.

ProyectoT2 - RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function | Addins

operadores.R | tema3.R* | Untitled1 | df

Filter

	altura	peso	habito	sexo	X1.10
1	167	86	No	M	1
2	192	74	Si	M	2
3	173	83	No	F	3
4	174	50	Si	M	4
5	172	78	No	M	5
6	167	66	No	M	6
7	171	66	Si	F	7
8	185	51	Si	M	8
9	163	50	Si	F	9
10	170	55	No	F	10

Figura 3: Visualización interactiva del *data frame.*

Es posible ordenar los datos según una variable haciendo clic en el nombre de esta última. También hay un campo de búsqueda y un botón *Filter* que da acceso a las opciones de filtrado de los datos.

3.7.1. Acceso a elementos de un data frame

Los *data frames* tienen las características tanto de listas como de matrices:

- Si utilizamos el operador `[]` con un solo índice, se comportan como listas e indexan las columnas, por lo que `df[1:2]`selecciona las dos primeras columnas.
- Si utilizamos el operador `[]` con dos índices, se comportan como matrices, por lo que `df[1:2, ]` selecciona las dos primeras filas (y todas las columnas).

Además de las técnicas vistas anteriormente para matrices, podemos usar aquí una nueva función muy útil llamada `subset()`. Recuerda que siempre puedes consultar la ayuda para a función (`help(subset)` o `?subset`).

En caso de lo que queramos hacer es acceder a una **columna** de un *data frame*, también será posible hacerlo mediante el operador `$` y el nombre de la columna deseada. Veamos algunos ejemplos.

- Seleccionar columnas
  - Por posición

```
df[c(2, 3)]                    # columnas 2 y 3
df[2:3]                        # idem
df[, c(2, 3)]                  # idem
subset(df,select = c(2, 3))    # idem
##    peso habito
## 1    86     No
## 2    74     Si
## 3    83     No
## 4    50     Si
## 5    78     No
## 6    66     No
## 7    66     Si
## 8    51     Si
## 9    50     Si
## 10   55     No

df[-c(2, 3)]                    # todas las columnas excepto la 2 y 3
df[, -c(2, 3)]                  # idem
subset(df,select = -c(2, 3))    # idem
##    altura sexo X1.10
## 1     167    M     1
## 2     192    M     2
```

```
## 3     173    F     3
## 4     174    M     4
## 5     172    M     5
## 6     167    M     6
## 7     171    F     7
## 8     185    M     8
## 9     163    F     9
## 10    170    F    10
```

- Por nombre

```
df$peso                       # columna "peso" (resultado: un vector)
## [1] 86 74 83 50 78 66 66 51 50 55
df["peso"]                    # columna "peso" (resultado: data.frame)
subset(df, select = peso)     # idem
subset(df, select = "peso")   # idem
## peso
## 1    86
## 2    74
## 3    83
## 4    50
## 5    78
## 6    66
## 7    66
## 8    51
## 9    50
## 10   55

subset(df, select = -c(peso, altura)) # todas las columnas excepto
                                      # peso y altura
##    habito sexo X1.10
## 1      No    M     1
## 2      Si    M     2
## 3      No    F     3
## 4      Si    M     4
## 5      No    M     5
## 6      No    M     6
## 7      Si    F     7
## 8      Si    M     8
## 9      Si    F     9
## 10     No    F    10
```

- Seleccionar filas
  - Por posición
  - Por condición lógica

Al igual que vimos con vectores, puede que nos interese extraer un subconjunto de observaciones (filas) que satisfacen una determinada condición. Veamos algunos ejemplos.

```
df[df$sexo == "M", ]             # seleccionamos solo los hombres
subset(df, subset = sexo == "M") # idem
subset(df, sexo == "M")          # idem
##   altura peso habito sexo X1.10
## 1    167   86     No    M     1
## 2    192   74     Si    M     2
## 4    174   50     Si    M     4
## 5    172   78     No    M     5
## 6    167   66     No    M     6
## 8    185   51     Si    M     8
>
# los hombres cuyo peso es inferior al peso promedio
subset(df, sexo == "M" & peso < mean(peso))
##   altura peso habito sexo X1.10
## 4    174   50     Si    M     4
## 8    185   51     Si    M     8

# idem, pero además conservamos solo las columnas altura y peso
subset(df, sexo == "M" & peso < mean(peso), select = c(altura, peso))
##   altura peso
## 4    174   50
## 8    185   51

# peso de las personas cuya altura es mayor a 170
subset(df,subset = altura>175, select = peso)
##   peso
## 2   74
## 8   51
```

Observación 1: *Comportamiento que puede ser no deseado de* `df$x`*: coincidencia parcial de nombres.*

Debemos tener cuidado cuando queremos extraer una columna de un *data frame* utilizando el operador `$`. Imaginemos que queremos extraer una columna utilizando `df$x`. Si no hay ninguna columna llamada `x`, `df$x` seleccionará cualquier variable que comience con `x`. Si ninguna variable comienza con `x`, `df$x` regresará `NULL`. Esto favorece seleccionar la variable incorrecta o seleccionar una variable que no existe.

Observación 2: *Preservar la dimensionalidad.*

De forma predeterminada, al crear subconjuntos de una matriz o *data frame* con un solo índice, un solo nombre o un vector lógico que contenga un solo TRUE, simplificará la salida devuelta, es decir, devolverá un objeto con menor dimensionalidad. Para conservar la dimensionalidad original, debe utilizar el argumento `drop = FALSE`.

```
A <- matrix(1:4, nrow = 2)
A
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
class(A[1, ])
## [1] "integer"    # es un vector
class(A)
## [1] "matrix" "array"   # pero A es una matriz!
class(A[1, , drop = FALSE])
## [1] "matrix" "array"   # esto si es una matriz
A[1, ]
## [1] 1 3
A[1, , drop = FALSE]
##      [,1] [,2]
## [1,]    1    3

# volviendo al df de ejemplo
class(df[, "peso"])
## [1] "numeric"        # es un vector
class(df[, "peso", drop = FALSE])
## [1] "data.frame"     # es un data.frame
df[, "peso"]
## [1] 86 74 83 50 78 66 66 51 50 55
df[, "peso", drop = FALSE]
##    peso
## 1    86
## 2    74
## 3    83
## 4    50
## 5    78
## 6    66
## 7    66
## 8    51
## 9    50
## 10   55
```

Puede que a esto ahora no le encontremos mucho sentido, pero el comportamiento predeterminado `drop = TRUE` es una fuente común de errores en las funciones: si la

hemos definido la función para aplicarla, por ejemplo, a un *data frame* y necesitamos utilizarla con un subconjunto de un *data frame*, podría ocurrir un error si al seleccionar dicho subconjunto no se preserva la estructura de *data frame*. Al escribir funciones, deberíamos acostumbrarnos a usar siempre `drop = FALSE` al crear subconjuntos de un objeto 2D.

Acceso directo utilizando la función `attach()`

Si no quieres escribir el nombre del *data frame* una y otra vez, simplemente podemos hacer uso directo de las variables aplicando función `attach()` al *data frame*. Básicamente la idea es "adjuntar" el contenido del *data frame* al entorno de trabajo de R, de modo que R las pueda encontrar directamente por su nombre.

```
peso   # no podemos acceder directamente a las variables(columnas) de df
Error: object 'peso' not found

attach(df)
The following objects are masked _by_ .GlobalEnv:
    altura, peso, sexo
peso  # ahora podemos acceder directamente a las variables(columnas)
## [1] 86 74 83 50 78 66 66 51 50 55
```

Cuando hayamos terminado de usar nuestras columnas, lo mejor es revertir el `attach()`. Es una buena práctica para evitar problemas. Para deshabilitar el acceso directo, solo hay que usar la función `detach()`:

```
detach(df)  # deshabilitamos el acceso directamente
peso
Error: object 'peso' not found
```

¿Por qué es importante utilizar `detach()`? Porque puede haber confusión si, en el espacio de trabajo, hay una variable (o creamos una) que tiene el mismo nombre que una columna del *data frame* que fue adjuntado, estaríamos sobreescribiendo el objeto prexistente. Veamos un ejemplo.

```
attach(df)
peso
## [1] 86 74 83 50 78 66 66 51 50 55
peso <- 90  # creamos una variable con el mismo nombre que una columna
peso
## [1] 90  # como resultado se sobreescribe la variable peso
```

3.7.2. Coerción a tipo data frame

Podemos coercionar una estructura de matriz o vector a un *data frame*. Esto puede resultar útil ligado a lo que vimos en la sección anterior de preservar la dimensionalidad. Incluso también para crear objetos de tipo `data.frame`.

```
matriz <- matrix(1:12, ncol = 4)
df2 <- as.data.frame(matriz)
class(df2)
## [1] "data.frame"
df2
##   V1 V2 V3 V4
## 1  1  4  7 10
## 2  2  5  8 11
## 3  3  6  9 12

c1 <- df2[,1]
c1
## [1] 1 2 3
class(c1)
## [1] "integer"
c12 <- as.data.frame(c1)
c12
##   c1
## 1  1
## 2  2
## 3  3
class(c12)
## [1] "data.frame"
```

3.7.3. Añadir y transformar columnas y filas

Añadir columnas

Para añadir filas o columnas a un *data frame* podemos utilizar las mismas funciones que vimos para matrices: `rbind()` o `cbind()`, respectivamente. También nos servirá el operador `$` en el caso de añadir nuevas columnas.

Además de estas técnicas que ya las hemos presentado anteriormente, podemos usar aquí una nueva función muy útil llamada `transform()` (consulta la ayuda para esta función usando `help(trasnform)` o `?transform`)

Supongamos que quisiéramos agregar las siguientes dos variables a nuestro *data frame* `df`:

```
nom <- c('Hugo', 'Ernesto','Emma','Lucas','Mateo','Leo','Begoña',
         'Alejandro','Carmen','Dolores')
edad <- c(46,54,27,17,32,19,44,33,21,38)
```

Para esto podemos usar uno de los siguientes comandos (son todos equivalentes):

```
df$nombre <- nom ; df$edad <- edad                # forma 1
df <- cbind(df, nombre = nom, edad = edad)        # forma 2
df <- transform(df, nombre = nom, edad = edad)    # forma 3
df
##    altura peso habito sexo X1.10    nombre edad
## 1     167   86     No    M     1      Hugo   46
## 2     192   74     Si    M     2   Ernesto   54
## 3     173   83     No    F     3      Emma   27
## 4     174   50     Si    M     4     Lucas   17
## 5     172   78     No    M     5     Mateo   32
## 6     167   66     No    M     6       Leo   19
## 7     171   66     Si    F     7    Begoña   44
## 8     185   51     Si    M     8 Alejandro   33
## 9     163   50     Si    F     9    Carmen   21
## 10    170   55     No    F    10   Dolores   38
```

Añadir filas

Si tenemos ahora uno o más nuevo/s individuo/s que agregar al *data frame*, necesitamos agregar una nueva fila. Lo haremos mediante la función `rbind()`. Pero primero tendremos que crear un *data frame* con la/s fila/s que se desea agregar.

```
#creamos un data.frame con los nuevos datos
new_F <- data.frame(altura = 178,
+                     peso = 72,
+                     habito = 'Si',
+                     sexo = 'F',
+                     X1.10 = 8,
+                     nombre = 'Lucia',
+                     edad = 25)
```

Es sumamente importante que al crear el *data frame* con las nuevas observaciones respetemos exactamente el nombre de las columnas del *data frame* al que las queremos agregar. De otro modo nos dará error.

Una vez creado el *data frame* con las filas a agregar, simplemente usando `rbind()` las añadimos al `df`:

```
df <- rbind(df,new_F)
dim(df)
## [1] 11  7   # efectivamente tenemos una fila más
df[10:11,]
##    altura peso habito sexo X1.10  nombre edad
## 10    170   55     No    F    10 Dolores   38
## 11    178   72     Si    F     8   Lucia   25  # la fila que agregamos
```

Transformar variables existentes

Supongamos ahora que queremos transformar las variables `sexo` y `habito` a factor. Esto podemos hacerlo así

```
str(df)
## 'data.frame':   11 obs. of  7 variables:
##  $ altura: num  167 192 173 174 172 167 171 185 163 170 ...
##  $ peso  : num  86 74 83 50 78 66 66 51 50 55 ...
##  $ habito: chr  "No" "Si" "No" "Si" ...   # son de tipo carácter
##  $ sexo  : chr  "M" "M" "F" "M" ...
##  $ X1.10 : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ nombre: chr  "Hugo" "Ernesto" "Emma" "Lucas" ...
##  $ edad  : num  46 54 27 17 32 19 44 33 21 38 ...
# las convertimos a factor
df$sexof <- factor(df$sexo); df$habito <- factor(df$habito)
df <- transform(df, sexo = factor(sexo), habito = factor(habito)) # idem
## 'data.frame':   11 obs. of  7 variables:
##  $ altura: num  167 192 173 174 172 167 171 185 163 170 ...
##  $ peso  : num  86 74 83 50 78 66 66 51 50 55 ...
##  $ habito: Factor w/ 2 levels "No","Si": 1 2 1 2 1 1 2 2 2 1 ...
##  $ sexo  : Factor w/ 2 levels "F","M": 2 2 1 2 2 2 1 2 1 1 ...
##  $ X1.10 : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ nombre: chr  "Hugo" "Ernesto" "Emma" "Lucas" ...
##  $ edad  : num  46 54 27 17 32 19 44 33 21 38 ...
```

Esto es importante porque, como vimos anteriormente, ciertas funciones reservan un tratamiento especial para los factores. Por ejemplo, podemos ver que es lo que da ahora la función genérica `summary()` (atendiendo a las columnas `"sexo"`,`"sexof"` , `"habito"` particularmente).

Observación: para que podamos ver el efecto de la transformación, en el caso de la variable "sexo", en lugar de transformar la variable original, agregamos una nueva columna llamada "sexof".

```
summary(df)
     altura           peso       habito     sexo
 Min.   :163.0   Min.   :50.00   No:5   Length:11
 1st Qu.:168.5   1st Qu.:53.00   Si:6   Class :character
 Median :172.0   Median :66.00          Mode  :character
 Mean   :173.8   Mean   :66.45
 3rd Qu.:176.0   3rd Qu.:76.00
 Max.   :192.0   Max.   :86.00
     X1.10           nombre               edad       sexof
 Min.   : 1.000   Length:11          Min.   :17.00   F:5
 1st Qu.: 3.500   Class :character   1st Qu.:23.00   M:6
 Median : 6.000   Mode  :character   Median :32.00
 Mean   : 5.727                      Mean   :32.36
 3rd Qu.: 8.000                      3rd Qu.:41.00
 Max.   :10.000                      Max.   :54.00
```

La mayoría de las veces, la función `transform()` se usa para agregar nuevas variables a partir de las existentes. Como ejemplo, consideremos el caso en el que queremos:

1. crear la variable altura.m: la altura en metros (m),
2. transformar el peso a gramos (g),
3. eliminar la variable altura,
4. crear la variable bmi (índice de masa corporal): peso (Kg) /altura (en m)^2

```
# forma 1
df2 <- transform(df, altura.m = 0.01*altura, peso = peso*1000,
+                    altura = NULL, imc = peso/(0.01*altura)^2)

# forma 2
df3 <- transform(df, altura.m = 0.01*altura, altura = NULL) |>
+   transform(imc = peso/altura.m^2, peso = peso*1000)

df2   # o df3, son idénticos
##     peso habito sexo X1.10    nombre edad sexof altura.m      imc
## 1  86000     No    M     1      Hugo   46     M     1.67 30.83653
## 2  74000     Si    M     2   Ernesto   54     M     1.92 20.07378
## 3  83000     No    F     3      Emma   27     F     1.73 27.73230
...
```

Como podemos observar, hemos quitado la columna correspondiente a la variable `"altura"` e incluimos `"altura.m"`. Además, agregamos la variable `"bmi"` y a `"peso"` la pasamos a gramos. El resto del *data frame* permanece igual.

Para comprender la sintaxis que hemos utilizado, debemos saber que una variable recién creada, a través de la función `transform()`, no se puede usar de inmediato. De hecho, la función `transform()` solo reconoce las variables/columnas originales (y sus valores) tal como están almacenadas en el *data frame* original/proporcionado. Esto significa, por ejemplo, que el siguiente código:

```
df3 <- transform(df, altura.m = 0.01*altura, altura = NULL, imc =
peso/altura.m^2, peso = peso*1000)
Error in eval(substitute(list(...)), `_data`, parent.frame()) :
  object 'altura.m' not found
```

es incorrecto y devuelve un error, pues no reconoce la variable `"altura.m"` que está siendo creada al mismo tiempo que la queremos usar para definir la variable `"bmi"`.

De este último ejemplo debemos notar además el uso del *pipe* `|>`. Lo veremos en detalle cuando estudiemos en profundidad operadores, pero vale la pena introducirlo aquí por la claridad y simplificación de código que supone. Dicho rápidamente, este operador se utiliza para encadenar instrucciones. La idea básica es reemplazar la expresión `f(a)`, donde `f` es cualquier función de R y `a` es un objeto, por

```
a |> f
```

que podemos leer como "tomar el objeto `a` y aplicarle la función `f` ". Por ejemplo:

```
20 |> sqrt()   # sqrt(20)
[1] 4.472136
```

Esta forma de escribir código R permite encadenar varias operaciones con una sintaxis más clara y fácil de leer. Por ejemplo, en lugar de

```
log(sqrt(abs(sin(23))),10)
## [1] -0.03625825
```

usando el operador |>, podemos escribir:

```
23 |> sin() |> abs() |> sqrt() |> log(base = 10)
[1] -0.03625825
```

lo cual vuelve mucho más clara la sintaxis. Para facilitar aún más la lectura del código, se recomienda realizar un salto de línea después de cada |>. Como, por ejemplo,

```
23 |>
  sin() |>
  abs() |>
  sqrt() |>
  log(base = 10)
```

Observación: |> es el operador nativo que viene de serie con R (básico). Hay otros operadores del mismo tipo, como el %>% del paquete {magrittr}, que forma parte de la colección de paquetes {tidyverse} que veremos más adelante.

3.7.4. Funciones útiles de manipulación de *data frames*

Descripción general/resumen

```
head(df, n = 3)  # muestra las primeras n filas (3 en este ejemplo)
##   altura peso habito sexo X1.10  nombre edad sexof
## 1    167   86     No    M     1    Hugo   46     M
## 2    192   74     Si    M     2 Ernesto   54     M
## 3    173   83     No    F     3    Emma   27     F

tail(df, n = 3)  # muestra las últimas n filas (3 en este ejemplo)
##    altura peso habito sexo X1.10  nombre edad sexof
## 9     163   50     Si    F     9  Carmen   21     F
## 10    170   55     No    F    10 Dolores   38     F
## 11    178   72     Si    F     8   Lucia   25     F

summary(df)   # proporciona un resumen de cada variable/columna
              # de acuerdo a su tipo
##      altura           peso       habito     sexo
##  Min.   :163.0   Min.   :50.00   No:5   Length:11
##  1st Qu.:168.5   1st Qu.:53.00   Si:6   Class :character
##  Median :172.0   Median :66.00          Mode  :character
##  Mean   :173.8   Mean   :66.45
##  3rd Qu.:176.0   3rd Qu.:76.00
##  Max.   :192.0   Max.   :86.00
##      X1.10           nombre               edad       sexof
##  Min.   : 1.000   Length:11          Min.   :17.00   F:5
```

```
##  1st Qu.: 3.500   Class :character   1st Qu.:23.00   M:6
##  Median : 6.000   Mode  :character   Median :32.00
##  Mean   : 5.727                      Mean   :32.36
##  3rd Qu.: 8.000                      3rd Qu.:41.00
##  Max.   :10.000                      Max.   :54.00
```

Estadísticas calculadas por grupo

A menudo sucede que queremos calcular estadísticas descriptivas resumen (como acabamos de ver con `summary()`), pero sólo para un subconjunto de individuos en un conjunto de datos. Esto es posible gracias a la función `aggregate()` cuya sintaxis (básica) viene dada por

```
aggregate(formula, data, FUN, ...)
```

donde

- `formula` es una fórmula de tipo `y ~ x`, siendo `y` la variable de interés y `x` el factor de agrupación,
- `data` es el conjunto de datos (data frame) al que pertenecen `x` e `y`,
- `FUN` el nombre de la función que se utilizará para los cálculos.

El argumento `...` indica que se pueden pasar argumentos adicionales a `FUN`. Es altamente recomendable, en este caso, nombrar explícitamente estos argumentos. Veamos algunos ejemplos:

```
# función aggregate
# altura promedio por sexo
aggregate(altura ~ sexo, data = df, FUN = mean)
##   sexo   altura
## 1    F 171.0000
## 2    M 176.1667

# cuartiles de altura por sexo et habito
aggregate(altura ~ sexo + habito, data = df, FUN = quantile)
##   sexo habito altura.0% altura.25% altura.50% altura.75% altura.100%
## 1    F     No    170.00     170.75     171.50     172.25      173.00
## 2    M     No    167.00     167.00     167.00     169.50      172.00
## 3    F     Si    163.00     167.00     171.00     174.50      178.00
## 4    M     Si    174.00     179.50     185.00     188.50      192.00

# mediana de altura por sexo
aggregate(altura ~ sexo, data = df, FUN = quantile, probs = 0.5)
# notar que probs es una argumento de quantile
aggregate(altura ~ sexo, data = df, FUN = median)   # idem
```

```
##   sexo altura
## 1    F    171
## 2    M    173

# mediana de altura y peso por sexo y habito
aggregate(cbind(altura, peso) ~ sexo + habito, data = df, FUN = quantile,
probs = 0.5)
##   sexo habito altura peso
## 1    F     No  171.5   69
## 2    M     No  167.0   78
## 3    F     Si  171.0   66
## 4    M     Si  185.0   51
```

Contar individuos en cada grupo

Supongamos que ahora queremos saber cuántos individuos o caso pertenecen a cada uno de los grupos determinados por uno o más factores. Esto lo podemos hacer con la función `table()`:

```
# función table
# ¿Cuantas mujeres hay?
table(df$sexo)
## F M
## 5 6     # hay 5 mujeres en el dataset

# Cantidad de mujeres que fuman
table(habito, sexo)
##       sexo
## habito F M
##     No 2 3
##     Si 2 3   # dos mujeres fuman
```

También podemos utilizar la función `xtab()`, cuyos principales argumentos son `formula` y `data`

```
# función xtab
# candidad de varones y mujeres
xtabs(~sexo, data = df)
## sexo
## F M
## 5 6

#  candidad de varones y mujeres según habito
xtabs(~sexo+habito, data = df)
##     habito
## sexo No Si
##    F  2  3
```

```
##    M  3  3
```

Cambiar nombre de filas y columnas

Si necesitamos cambiar el nombre de las filas o columnas de un *data frame*, utilizamos las funciones `names()` (para las columnas) y `row.names()` para las filas.

```
names(df)[5] <- 'numero'  # cambiamos "X1.10" a "numero"
rownames(df) <- paste0(1:11,'th')
head(df, n = 3)
##        altura peso habito sexo numero  nombre edad sexof
## fila_1    167   86     No    M      1    Hugo   46     M
## fila_2    192   74     Si    M      2 Ernesto   54     M
## fila_3    173   83     No    F      3    Emma   27     F
```

Ordenar las filas según valores crecientes o decrecientes una variable cuantitativa

```
# función order
# ordenar los individuos según la altura (creciente)
df[order(df$altura), ]
##         altura peso habito sexo numero    nombre edad sexof
## fila_9     163   50     Si    F      9    Carmen   21     F
## fila_1     167   86     No    M      1      Hugo   46     M
## fila_6     167   66     No    M      6       Leo   19     M
...

# ordenar los individuos según la altura (decreciente)
df[order(df$altura,decreasing = TRUE), ]
df[order(-df$altura), ]                # idem

##         altura peso habito sexo numero    nombre edad sexof
## fila_2     192   74     Si    M      2   Ernesto   54     M
## fila_8     185   51     Si    M      8 Alejandro   33     M
## fila_11    178   72     Si    F      8     Lucia   25     F
...

# ordenar por altura y peso
df[order(df$altura, df$peso), ]
##         altura peso habito sexo numero    nombre edad sexof
## fila_9     163   50     Si    F      9    Carmen   21     F
## fila_6     167   66     No    M      6       Leo   19     M
## fila_1     167   86     No    M      1      Hugo   46     M
...
```

Apilar columnas

Frecuentemente nos encontraremos con la necesidad de apilar las columnas de un *data frame*, principalmente para preparar los datos antes de graficarlos.

Supongamos que contamos con datos correspondientes a resultados de dos tipos diferentes de pruebas diagnósticas, aplicadas sobre tres individuos:

```
df2 <- data.frame(
+   individuo = c("A", "A", "B", "B", "C", "C"),
+   prueba = c(1, 2, 1, 2, 1, 2),
+   res_1 = c(4, 2, 7, 3, 5, 6),
+   res_2 = c(8, 5, 0, 7, 7, 8),
+   res_3 = c(0, 5, 6, 1, 3, 4))
df2
##   individuo prueba res_1 res_2 res_3
## 1         A      1     4     8     0
## 2         A      2     2     5     5
## 3         B      1     7     0     6
## 4         B      2     3     7     1
## 5         C      1     5     7     3
## 6         C      2     6     8     4
```

Si quisiéramos graficar, o calcular estadísticas resumen por cada tipo de prueba por ejemplo, sería simple de hacer (función `aggregate()` vista anteriormente). Para esto sería más simple contar con una variable que indique el tipo de prueba, otra que indique individuos y otra que nos de el valor propiamente dicho. Para lograr esto podemos utilizar la función `stack()`.

Veamos primero el resultado de aplicar la función al *data frame* completo:

```
stack(df2)
   values       ind
1       A individuo
2       A individuo
3       B individuo
4       B individuo
5       C individuo
6       C individuo
7       1    prueba
8       2    prueba
9       1    prueba
10      2    prueba
11      1    prueba
```

```
12      2    prueba
13      4     res_1
14      2     res_1
15      7     res_1
16      3     res_1
17      5     res_1
18      6     res_1
19      8     res_2
20      5     res_2
21      0     res_2
22      7     res_2
23      7     res_2
24      8     res_2
25      0     res_3
26      5     res_3
27      6     res_3
28      1     res_3
29      3     res_3
30      4     res_3
```

Como vemos, la función devuelve un *data frame* con dos columnas: la primera llamada `value` con los valores correspondientes a las columnas del *data frame* `df2` apiladas y la segunda, llamada `ind`, con los nombre de la variable que corresponde a cada valor.

Pero esto no es lo que queremos. Nos gustaría tener apilados los resultados de las pruebas, asignado a cada valor el individuo y tipo de prueba correspondiente. En nuestro *data frame*, sería apilar solo las últimas tres columnas.

```
# apilar ultimas tres columnas
cbind (df2[1: 2], stack(df2[3:5]))
##    individuo prueba values   ind
## 1          A      1      4 res_1
## 2          A      2      2 res_1
## 3          B      1      7 res_1
## 4          B      2      3 res_1
## 5          C      1      5 res_1
## 6          C      2      6 res_1
## 7          A      1      8 res_2
## 8          A      2      5 res_2
## 9          B      1      0 res_2
## 10         B      2      7 res_2
## 11         C      1      7 res_2
## 12         C      2      8 res_2
## 13         A      1      0 res_3
## 14         A      2      5 res_3
## 15         B      1      6 res_3
```

```
## 16          B       2       1 res_3
## 17          C       1       3 res_3
## 18          C       2       4 res_3
```

Otra posibilidad para hacer este es utilizar la función `melt()` del paquete `{reshape2}`. Como siempre, para saber más sobre esta función, consulta la ayuda disponible utilizando `help(melt)` o `?melt`.

```
library(reshape2)
# apilar columnas 'measure.vars' del data frame data
melt(data = df2,
     id.vars = c ('individuo', 'prueba'),
     measure.vars = c('res_1','res_2','res_3'),
     value.name = 'resultados')
```

3.7.5. Una variante de los *data frames*: los *tibbles*

Con el tiempo, se han desarrollado variantes "más modernas" de los *data frames*. Entre las estructuras más conocidas/usadas se encuentran los *data table* (del paquete `{data.table}`) y los *tibbles* (del paquete `{tidyverse}`), cada una tiene sus ventajas y desventajas.

Una de las mayores ventajas de los *data table* es que ocupan menos espacio que los *data frames*. Otra ventaja es la mejora en los tiempos de ejecución para leer y escribir un csv, para hacer un `group_by`, para ordenar una tabla, entre otras funciones.

Los *data table* pueden ser utilizados como *data frame*. En efecto, si se pedimos a R que nos diga la clase de un objeto de este tipo, usando `class(Tabla)` donde `Tabla` es un `data.table` se imprime una lista con dos valores: `"data.table"  "data.frame"`, indicando que los objetos *data table* también pueden ser considerados como *data frame* y las funciones que utilizan *data frames* no tendrían problemas al usar *data tables*.

Los *tibbles* forman parte del paquete del `{tibble}` y son la estructura de datos propia de la colección de paquetes `{tidyverse}`. Para poder crear o manipular *tibbles*, debes

instalar el paquete `{tibble}` (o el `{tidyverse}`, que instalará todos los paquetes que forman parte de este *universo*). Daremos una pequeña introducción al `{tidyverse}` en el Tema 5: Manejo de datos.

Crear un *tibble* es similar a crear un *data frame*.

```
# altura medida en cm
altura <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170)
# peso en Kg
peso <- c(86, 74, 83, 50, 78, 66, 66, 51, 50, 55)
# habito de fumador
fuma <- c("No", "Si", "No", "Si", "No", "No", "Si", "Si", "Si", "No")
sexo <- c("M", "M", "F", "M", "M", "M", "F", "M", "F", "F")

library(tibble)
tbl <- tibble(Altura = altura, Peso = peso, Habito = fuma, Sexo = sexo)
tbl
# A tibble: 10 × 4
##    Altura  Peso Habito Sexo
##     <dbl> <dbl> <chr>  <chr>
##  1    167    86 No     M
##  2    192    74 Si     M
##  3    173    83 No     F
##  4    174    50 Si     M
##  5    172    78 No     M
##  6    167    66 No     M
##  7    171    66 Si     F
##  8    185    51 Si     M
##  9    163    50 Si     F
## 10    170    55 No     F
```

Notemos que, al mostrar un *tibble* evaluando su nombre en la consola:

- R muestra un comentario que indica la naturaleza del objeto (*tibble*) y sus dimensiones (en el ejemplo, 10 x 4, 10 filas y 4 columnas)
- R solo muestra las primeras 10-20 filas y todas las columnas que pueden caber (adecuadamente) en la pantalla (en el ejemplo no lo notamos porque el *tibble* que creamos es chico);

  Se muestra el tipo de cada variable/columna, en forma abreviada, justo debajo de los nombres de las columnas.
- Además, las filas no están, estrictamente hablando, nombradas, simplemente están numeradas.

A pesar de estas diferencias, los *tibbles* y los *data frames* se manejan exactamente de la misma manera. Por lo tanto, podemos aplicar a un *tibble* todo lo que hemos aprendido con los *data frames*. En efecto, si pedimos a R que nos muestre la clase de un objeto *tibble*, nos devuelve una lista con tres valores:

```
class(tbl)
[1] "tbl_df"     "tbl"        "data.frame"
```

identificando también a los *tibbles* como `data.frame`.

Además, es fácil convertir un *tibble* en un *data frame* y viceversa utilizando las funciones de coerción `as.tibble()` y `as.data.frame()` respectivamente.

```
mi_df <- as.data.frame(tbl)
str(mi_df)
## 'data.frame':    10 obs. of  4 variables:
##  $ Altura: num  167 192 173 174 172 167 171 185 163 170
##  $ Peso  : num  86 74 83 50 78 66 66 51 50 55
##  $ Habito: chr  "No" "Si" "No" "Si" ...
##  $ Sexo  : chr  "M" "M" "F" "M" ...
mi_tbl <- as.tibble(mi_df)
str(mi_tbl)
## tibble [10 × 4] (S3: tbl_df/tbl/data.frame)
##  $ Altura: num [1:10] 167 192 173 174 172 167 171 185 163 170
##  $ Peso  : num [1:10] 86 74 83 50 78 66 66 51 50 55
##  $ Habito: chr [1:10] "No" "Si" "No" "Si" ...
##  $ Sexo  : chr [1:10] "M" "M" "F" "M" ...
```

3.8. Listas

Las listas son las estructuras menos rígidas y más versátiles del lenguaje R. Al igual que los vectores, son estructuras de datos **unidimensionales**, *sólo tienen largo* o *longitud*, pero a diferencia de los vectores cada uno de *sus elementos puede ser de diferente tipo o incluso de diferente clase*, por lo que son estructuras **heterogéneas**. Podemos tener listas que contengan datos atómicos, vectores, matrices, *arrays*, *data frames* u otras listas. Esta última característica es la razón por la que una lista puede ser considerada un vector recursivo, pues es un objeto que puede contener objetos de su misma clase.

3.8.1. Crear listas

Para crear una lista usamos la función `list()`, escribiendo los elementos que deseamos incluir en nuestra lista entre paréntesis y separados por comas. Para esta estructura, no importan las dimensiones o largo de los elementos que queramos incluir en ella.

```
# Primero creamos los elementos que conformarán nuestra lista
v <- runif(10,2,10)
m <- matrix(1:4, ncol = 2)
df <- data.frame("Nombres" = c('Lucia', 'Bianca', 'Fernando','Carlos'),
"edad" = c(19,20,34,56))
# creamos la lista
lista1 <- list(v,m,df)
lista1
## [[1]]
##  [1] 8.022847 5.386355 7.779090 5.575576 3.523567 5.237170 3.365984
##  [8] 4.728168 8.822085 9.355052

## [[2]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

## [[3]]
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56
```

También podemos nombrar cada uno de los objetos en la lista, de la misma forma que nombramos los elementos de un vector, lo que permite una mejor lectura y en ocasiones un acceso más práctico a los datos.

```
# forma 1: nombrar los elementos al crear la lista
lista2 <- list("Un_vector" = v, "Una_matriz" = m, "Un_df" = df)
# forma 2: añadir el atributo de nombres a la lista ya creada
names(lista1) <- c("Un_vector", "Una_matriz", "Un_df")
lista2   # o lista 1, ahora son idénticas
## $Un_vector
##  [1] 7.159325 2.468880 8.581208 9.508102 8.163141 2.849730 9.879412
##  [8] 5.071620 3.123112 9.352333
##
## $Una_matriz
##      [,1] [,2]
## [1,]    1    3
```

```
## [2,]    2    4
##
## $Un_df
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56
```

Como lo mencionamos al comienzo de esta sección, las listas a veces se denominan **vectores recursivos** porque una lista puede contener otras listas. Esto los hace fundamentalmente diferentes de los vectores atómicos. Por ejemplo:

```
lista3 <- list(list(list(1)))
lista3
## [[1]]
## [[1]][[1]]
## [[1]][[1]][[1]]
## [1] 1
```

Para conocer la estructura interna de una lista, es decir, los nombres de los objetos que la forman y su naturaleza, podemos usar la función `str()`. Si sólo queremos saber sus nombres, podemos usar la función `names()`.

```
str(lista2)
## List of 3  # longitud de la lista (cantidad de elementos que la forman)
##  $ Un_vector : num [1:10] 8.02 5.39 7.78 5.58 3.52 ... # 1er elemento
##  $ Una_matriz: int [1:2, 1:2] 1 2 3 4                  # 2do elemento
##  $ Un_df     :'data.frame':  4 obs. of  2 variables:   # 3er elemento
##   ..$ Nombres: chr [1:4] "Lucia" "Bianca" "Fernando" "Carlos"
##   ..$ edad   : num [1:4] 19 20 34 56

# solo los nombres
names(lista2)
## [1] "Un_vector"  "Una_matriz" "Un_df"
```

3.8.2. Propiedades de una lista

Una lista es unidimensional, sólo tiene longitud. La longitud de una lista es igual al número de elementos que contiene, sin importar de qué tipo o clase sean.

```
length(lista1)
## [1] 3
length(lista3)
## [1] 1
```

Dado que una lista siempre tiene una sola dimensión, la función `dim()` nos devuelve `NULL`

```
dim(lista1)
## NULL
```

Las listas tienen clase `list`, sin importar qué elementos contienen.

```
class(lista1)
## [1] "list"
class(lista3)
## [1] "list"
```

Pruebas y coerción

Las listas tienen tipo `list` (igual que su clase). Para comprobarlo, utilizamos la función `typeof()`. Podemos chequear si un objeto es una lista con `is.list()`, y obligar a un objeto al tipo `list` con `as.list()`.

```
list(1:3)     # crea una lista con único elemento el vector c(1,2,3)
## [[1]]
## [1] 1 2 3
##
as.list(1:3)   # fuerza al vector c(1,2,3) a lista
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3

is.list(lista1)
## [1] TRUE
is.list(1:3)
## [1] FALSE
```

A diferencia de los vectores atómicos, no es posible vectorizar operaciones aritméticas usando listas. Al intentarlo nos es devuelto un error.

```
lista1^2
Error in lista1^2 : non-numeric argument to binary operator
lista1*2
Error in lista1 * 2 : non-numeric argument to binary operator
```

Para aplicar una función a cada elemento de una lista, usamos `lapply()`, como veremos en temas siguientes.

Finalmente, podemos convertir una lista en un vector atómico empleando la función `unlist()`, aunque es preciso mencionar que las reglas para el tipo resultante son complejas, no están bien documentadas y no siempre son equivalentes a lo que se obtendría con `c()`.

3.8.3. Acceder a los elementos de una lista

Podemos acceder a un elemento de una lista de varias maneras diferentes.

```
# acceder a los elementos de una lista
# por la posición, utilizando corchetes dobles
lista1[[3]]
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56

# por el nombre, utilizando corchetes dobles
lista1[["Un_df"]]
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56

# por el nombre, utilizando el operador $ (lo más frecuente)
lista1$Un_df
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56
```

También puede construir una sublista a partir de una lista, usando corchetes simples para esto.

```
# creamos una sublista con los elementos 1 y 2 de lista1
sublista1 <- lista1[c(1, 2)]    # notar que usamos corchetes simples
sublista1
## $Un_vector
##  [1] 8.022847 5.386355 7.779090 5.575576 3.523567 5.237170 3.365984
```

```
##  [8] 4.728168 8.822085 9.355052
##
## $Una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

3.8.4. Editar una lista

Al igual que con los vectores, podemos modificar los elementos de una lista existente o agregar/eliminar elementos de ella.

```
# editar elementos de una lista
lista1[[1]] <- 0      # modificamos el 1er elemento
lista1
## $Un_vector    # notar que cambia el contenido, pero no el nombre
## [1] 0
##
## $Una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $Un_df
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56

lista1$Un_vector <- 1:10  # lo volvemos a modificar utilizando el nombre
lista1
## $Un_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $Una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $Un_df
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56

lista1$Un_vector <- NULL  # quitamos Un_vector de la lista
lista1
```

```
## $Una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $Un_df
##    Nombres edad
## 1    Lucia   19
## 2   Bianca   20
## 3 Fernando   34
## 4   Carlos   56
```

3.8.5. La función `c()` con listas

La función `c()` que utilizamos para crear vectores, también puede utilizarse para concatenar dos o más listas. De esta forma, los elementos de las diferentes listas se irán enlazando uno tras otro para construir una única lista grande. Si se le da una combinación de vectores atómicos y listas, `c()` forzará a los vectores a listas (coerción de tipo) antes de combinarlos. Veamos algunos ejemplos:

```
lista4 <- list(list(1, 2), c(3, 4))  # creamos una lista de 2 elementos
lista5 <- c(list(1, 2), c(3, 4))     # concatenamos 2 elementos
str(lista4)
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ : num [1:2] 3 4
str(lista5)
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

Como podemos observar, al utilizar la función `c()` aplicada a una lista y un vector, se produce automáticamente la coerción del vector a lista, para formar una lista única con 4 elementos. Por el contrario, emplear la función `list()` a los mismo elementos (una lista y un vector), crea una lista de dos elementos (la lista y el vector).

3.9. Referencias bibliográficas

Douglas, A., Roos,D., Mancini, F., Couto,A. and Lusseau, D. (2022). *An Introduction to R.* [Una introducción a R]. https://intro2r.com/

Fisher, R. A. (1936). "The use of multiple measurements in taxonomic problems". *Annals of Eugenics. 7* (2): 179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

Paradis, E. (2003). *R para Principiantes*. (Trad. Ahumada, J.) https://cran.r-project.org/doc/contrib/rdebuts_es.pdf

Soage, J. C. (s.f.). Estructuras de datos. En *R Coder*. Recuperado el 8 de agosto de 2022 de: https://r-coder.com/curso-r/

Wickham, H. (2019). *Advanced R* (2nd edition). Chapman & Hall's R Series. Disponible en https://adv-r.hadley.nz/index.html

3.10. Cuaderno de ejercicios

Ejercicio 1

A partir de las funciones estudiadas en el capítulo:

1. Construye un vector con las primeras 15 letras MAYÚSCULAS usando el vector constante LETTERS de R.
2. Construye una matriz de 10 x 10 con los primero 100 números pares positivos.
3. Construye una matriz identidad de orden 5.
4. Construye una lista con los objetos creados en los ítems anteriores.

5. Construye un *data frame* con las respuestas de 3 personas a las preguntas: (a) ¿Cuál es su edad en años? (b) ¿Tipo de música que más le gusta? (c) ¿Practica algún deporte regularmente?

Ejercicio 2

1. ¿Cuál es el error en el siguiente código? ¿Por qué ocurre?

```
edad <- c(15, 19, 13, NA, 20)
deporte <- c('SI', 'NO', 'SI', 'SI', 'NO')
ciudad <- c(NA, 'Madrid', 'Quito', 'Santa Fe', 'Medellin')
matrix(edad, deporte, ciudad)
```

2. Arregla el código anterior para obtener una matriz cuyas columnas sean los vectores `edad`, `deporte` y `cuidad`.

Ejercicio 3

Pon a prueba tu conocimiento de las reglas de coerción de vectores prediciendo el resultado de los siguientes usos de la función `c()`:

1. `c(1,'dos')`
2. `c(1,FALSE)`
3. `c(TRUE,1L)`
4. `c(TRUE,'false')`

Ejercicio 4

En el siguiente ejercicio deberás trabajar con el *datasets* `mtcars`, contenido en el paquete `{datasets}` que trae R incorporado con su instalación básica. Corrige cada uno de los siguientes errores frecuentes en la creación de subconjuntos de data frames:

1. mtcars[mtcars$cyl = 4,]

2. mtcars[-1:4,]
3. mtcars[mtcars$cyl <= 5]
4. mtcars[mtcars$cyl == 4 | 6,]

Ejercicio 5

En cada una de las siguientes expresiones, explica el resultado que devuelve R:

1. 1 == "1" es verdadero (TRUE)
2. -1 < FALSE es verdadero (TRUE)
3. "one" < 2 es falso (FALSE)

Ejercicio 6

Si `df` es un *data frame*, ¿qué puedes decir acerca `t(df)` y de `t(t(df))`? Realiza algunos experimentos, asegurándote de considerar diferentes tipos de columnas en tu *data frame*. Debería prestar especial atención al tipo de estructura del resultado, el tipo de cada columna (original y del resultado), las dimensiones (del *data frame* original y del resultado).

[i] Matriz diagonal es una matriz cuyos elementos fuera de su diagonal principal son todos 0.
[ii] Matriz cuadrada y diagonal cuyos elementos sobre la diagonal principal son todos iguales a 1.