

Introducción a R y RStudio

---

# Programación básica

# Índice

Ideas clave	3
4.1. Introducción y objetivos	3
4.2. Operadores en R	3
4.3. Estructuras de control	14
4.4. Funciones	43
4.5. Referencias bibliográficas	58
4.6 Cuaderno de ejercicios	58

## 4.1. Introducción y objetivos

Hasta el momento hemos utilizado funciones programadas previamente en R, pero cabe la posibilidad de que necesitemos una función que haga una tarea tan particular que no se encuentre disponible en ningún paquete existente.

Es por eso por lo que en este tema daremos las bases para poder crear funciones sencillas en R y cómo utilizarlas para resolver problemas. Primero, estudiaremos los principales tipos de operadores que encontramos en R, luego cubriremos las estructuras de control (condicionales e iterativas) y finalmente nos ocuparemos de la creación de funciones y como aplicarlas.

Al finalizar este tema, esperamos alcanzar los siguientes objetivos:

- ▶ Ser capaces de controlar la ejecución de código con la ayuda de las estructuras condicionales.
- ▶ Estar capacitado para ejecutar bloques de código repetidamente mediante estructuras iterativas.
- ▶ Escribir declaraciones condicionales utilizando `if()` y `else()`.
- ▶ Escribir y entender los bucles con `for()`.
- ▶ Definir funciones sencillas que utilicen argumentos.

## 4.2. Operadores en R

Comencemos con un poco de vocabulario: ¿qué es un **operador**? Es simplemente un elemento del lenguaje, como `+`, `-` o, incluso `=`, que son operadores que (seguramente)

ya conoces. Así + se llama operador de suma, - de resta, e = se llama operador de asignación (porque permite asignar valores a variables).

Los operadores son los símbolos que le indican a R que debe realizar una tarea. Combinando datos y operadores es como logramos que R haga su trabajo.

Existen en R varios operadores para trabajar. Esto incluye operadores aritméticos para cálculos matemáticos, operadores lógicos, relacionales o de asignación.

### 4.2.1. Operadores aritméticos

Los operadores aritméticos de R nos permiten realizar operaciones matemáticas, como sumas, divisiones o multiplicaciones, entre otras. Gracias a ellos, podemos utilizar la consola de R tal como si fuera una calculadora. Las operaciones usuales se indican en R con los símbolos que se muestran en la Tabla 1.

OPERACIÓN	Suma	Resta	Producto	División	Potencia	Cociente entero	Resto
Signo	+	-	*	/	^	%%/%	%%

Tabla 1: Las operaciones básicas en R.

En cuanto a la operación radicación, excepto para la raíz cuadrada, no existe ningún símbolo o función especial para su cálculo, pero si recordamos que  $\sqrt[a]{b} = b^{\frac{1}{a}}$ , podremos calcular raíces de cualquier índice haciendo uso de la operación de potenciación.

Veamos algunos ejemplos de uso de estas operaciones:

```
3 * 5 + 10 / 2 # aquí lo único que se divide por 2 es el 10
## [1] 20
(3 * 5 + 10) / 2 # ahora dividimos por 2 al resultado de lo que está
                  # encerrado entre paréntesis
## [1] 12.5
75 / 5^2 # notar el orden en que realiza las operaciones
## [1] 3
(2 + 3) * 75 / 5^2
## [1] 15
```

```

# Si queremos calcular la raíz cuadrada de 16, tenemos dos maneras
sqrt(16)
## [1] 4
16^(1 / 2) # observar el uso de paréntesis para indicar la precedencia
# de las operaciones
## [1] 4
# pero, para obtener la raíz cúbica de 27,
27^(1 / 3)
## [1] 3
27^1 / 3 # ¡No olvidar los paréntesis!
## [1] 9
# ¿Cuántos días completos hay en 12345 horas?
12345 %% 24
## [1] 514
# ¿cuántas horas sobran?
12345 %%% 24
## [1] 9
# Podemos chequear que 12345 = 514*24+9
12345 == 514 * 24 + 9
## [1] TRUE

```

Para tener en cuenta: el **orden de precedencia** en las operaciones que sigue R es:

1. Paréntesis (),
2. Exponentes ^,
3. División / y Multiplicación \*,
4. Suma + y Resta -.

Como lo hemos mencionado en el Tema 3, al hablar de las estructuras de datos, la mayoría de las operaciones en R se vectorizan al aplicarse a vectores, matrices o *arrays*, lo cual nos insta a tener especial cuidado al momento de operar con estas estructuras, para obtener resultados que realmente buscamos. El ejemplo emblemático de esto podría ser la multiplicación de matrices. Ya vimos que aplicar el operador `*` a dos matrices (suponiendo que son ambas del mismo tamaño) nos devuelve otra matriz (del mismo tamaño) cuyos elementos corresponden a los productos, elemento a elemento, de las matrices operadas. Si lo que buscamos es multiplicar las matrices (tal y como lo aprendimos en algún curso de álgebra lineal), tenemos que utilizar el operador `%*%`.

```

A <- matrix (1, nrow = 2, ncol = 2)
B <- matrix (1:4, ncol = 2)
A
##      [,1] [,2]

```

```
## [1,] 1 1
## [2,] 1 1
B
##      [,1] [,2]
## [1,] 1 3
## [2,] 2 4
A * B
##      [,1] [,2]
## [1,] 1 3
## [2,] 2 4
A %*% B
##      [,1] [,2]
## [1,] 3 7
## [2,] 3 7
```

### 4.2.2. Operadores lógicos en R

Los operadores lógicos en R se utilizan para describir relaciones lógicas entre objetos. Estas comparaciones devuelven valores TRUE o FALSE.

Operador	Comparación	Ejemplo	Resultado
<code>x y</code>	Indica a disyunción lógica (O): x o y (o los dos) son verdaderos	TRUE   FALSE TRUE   TRUE FALSE   FALSE	TRUE TRUE FALSE
<code>x&amp;y</code>	Indica la conjunción lógica (Y): x e y son verdaderos	TRUE & FALSE	FALSE
<code>!x</code>	Indica la negación lógica	!TRUE !FALSE	FALSE TRUE
<code>xor()</code>	Indica la disyunción exclusiva: x o y (pero no los dos) son verdaderos	<code>xor(TRUE, TRUE)</code> <code>xor(TRUE, FALSE)</code> <code>xor(FALSE, FALSE)</code>	FALSE TRUE FALSE

Tabla 2: Los operadores lógicos en R.

La Figura 1 muestra el conjunto de operaciones lógicas y sus resultados.

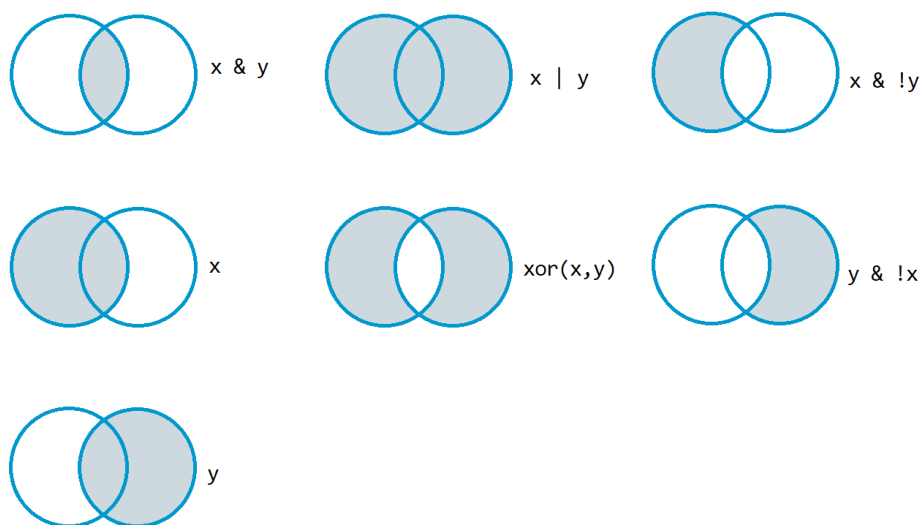


Figura 1: Operaciones lógicas: x es el círculo de la derecha, y el de la izquierda y la región sombreada el resultado de la operación. Fuente: Adaptado de (Wickham, H et al. 2019)

Los operadores comparativos actúan sobre cada elemento de los dos objetos que se están comparando (reciclando los valores de los más pequeños si es necesario), devolviendo un objeto del mismo tamaño. Si queremos comparar un objeto de forma global (es decir, comparar elemento a elemento pero devolver un único resultado final) es necesario usar alguna de las funciones `identical()` (evalúa si dos objetos son exactamente iguales. Devuelve `TRUE` en tal caso, y `FALSE` en caso contrario), `any()` (aplica sobre un vector lógico. Devuelve `TRUE` si al menos una componente del vector es `TRUE`, y `FALSE` en caso contrario), `all()` (aplica sobre un vector lógico. Devuelve `TRUE` si todas las componentes del vector son `TRUE`, y `FALSE` en caso contrario):

```
x <- 1:3; y <- 1:3; z <- c(1, 2, 4)
# comparamos x e y
x == y
## [1] TRUE TRUE TRUE
identical(x, y)
## [1] TRUE
all(x == y)
## [1] TRUE
any(x == y)
## [1] TRUE

# comparamos x y z
```

```
x == z
## [1] TRUE TRUE FALSE
identical(x, z)
## [1] FALSE
all(x == z)
## [1] FALSE
any(x == z)
## [1] TRUE
```

### 4.2.3. Operadores relacionales en R

Los operadores relacionales o de comparación están diseñados para comparar objetos. El resultado de estas comparaciones es de tipo booleano (siempre devuelven como resultado TRUE o FALSE - verdadero o falso, respectivamente -).

Operador	Comparación	Ejemplo	Resultado
<	Menor que	2 < 6 6 < 2 2 < 2	TRUE FALSE FALSE
<=	Menor o igual que	2 <= 6 6 <= 2 2 <= 6	TRUE FALSE TRUE
>	Mayor que	2 > 6 6 > 2 2 > 6	FALSE TRUE FALSE
>=	Mayor o igual que	2 >= 6 6 >= 2 2 >= 2	FALSE TRUE TRUE
==	Exactamente igual a	2 == 2 2 == 6	TRUE FALSE
!=	No es igual a	2 != 2 2 != 6	FALSE TRUE

Tabla 3: Los operadores relacionales de R.

Los operadores lógicos y los relacionales nos serán de utilidad para crear estructuras condicionales, por ejemplo, con la función `if`, como veremos en la próxima sección. Solo para anticiparnos, la idea básica de la construcción `if` es: “Hace lo que sigue si se cumple esta condición”. El “si se cumple” refiere a una comparación lógica o de



relación, que eventualmente será TRUE, y se ejecutará la instrucción, o FALSE y terminará sin ejecutarla.

#### 4.2.4. Operadores de asignación

Aunque ya hemos utilizado la asignación <- (para dar valores a variables), existen otros operadores que podemos utilizar para realizar asignaciones de valores. Los operadores de asignación en R permiten asignar datos a un objeto para almacenarlos y/o crear nuevos objetos.

Operador de asignación en R	Descripción
<-	Asignación izquierda
=	Asignación izquierda (no recomendado) y asignación de argumentos
->	Asignación derecha

Tabla 4: Operadores de asignación

```
# son todas expresiones equivalentes para asignar 10 a x:
x <- 10
x = 10
10 -> x
x
## [1] 10
muestra <- rnorm(n = 10) # utilizamos = para asignar en argumentos
n
Error: object 'n' not found
```

Una vez que hemos realizado la operación de asignación, podemos usar el *nombre de la variable* para realizar operaciones con ella, **como si fuera del tipo de datos que le hemos asignado**. Debemos tener presente que, si asignamos un valor a una variable a la que ya habíamos asignado datos, esta variable conserva el valor más reciente.

```
x <- 10
x <- -2
x
## [1] -2
```

Notemos, además, que esta operación nos permite guardar el resultado de operaciones, de modo que podemos recuperarlos sin necesidad de realizar las operaciones otra vez.

```
x <- 10
y <- 8
suma <- x + y
semi_suma <- suma / 2 # no necesitamos volver a escribir x + y pues
                      # ya lo tenemos almacenado en la variable suma
semi_suma # vemos que se guardó en esta variable
## [1] 9   # que es el resultado de (10 + 8) / 2
```

## Otros operadores

También encontramos en R otros operadores que no entran en las categorías anteriores, pero son igualmente importantes. Estos operadores se utilizan para fines específicos, como acceder a datos, funciones, crear secuencias o especificar la fórmula de un modelo. La siguiente tabla contiene algunos operadores disponibles en R utilizados con mayor frecuencia.

Operador	Descripción	Ejemplo	Resultado
\$	Se utiliza para acceder a una variable de un data frame	df <- data.frame( x = c(7, 9, 2), y = c(5, 9, 5)) df\$x	[1] 7 9 2
:	Genera una sucesión de números.	2:5	[1] 2 3 4 5
::	Se utiliza para acceder a una función de un paquete	car::boxCox(ajuste)	(aplica la función boxCox del paquete car al objeto <i>ajuste</i> )
%in%	Se utiliza para identificar si un elemento se encuentra en un vector.	x <- 2 y <- c(1,2,4) x %in% y	TRUE
~	Formulación de modelos	lm(y~x)	(ajuste la regresión lineal de y en x)

Tabla 5: Operadores misceláneos en R.

Para tener en cuenta:

## La notación científica

Cuando un número es muy grande o pequeño, R utiliza la notación científica para dar el resultado aproximado:

```
# La notación científica
3^30
## [1] 2.058911e+14
3^(-30)
## [1] 4.856936e-15
```

En el ejemplo,  $2.058911e+14$  representa el número  $2.058911 \cdot 10^{14}$ , es decir, 205891100000000 (omitimos la utilización de puntos como separadores de miles para evitar la confusión con la notación que utiliza R, de acuerdo con la convención angloamericana, de utilizar el punto como separador decimal), y  $4.856936e-15$  representa el número  $4.856936 \cdot 10^{-15}$ , es decir, 0.000000000000004856936.

## Otras funciones matemáticas disponibles

R también dispone de muchas funciones matemáticas integradas, entre otras, las que se muestran en la Tabla 2. Para llamar a una función, simplemente escribimos su nombre seguido de paréntesis (). Todo lo que escribas dentro de los paréntesis se llaman argumentos de la función.

FUNCIÓN	$\sqrt{x}$	$e^x$	$\ln(x)$	$\log_{10}(x)$	$\log_a(x)$	$n!$	$\binom{m}{n}$
Signo	sqrt	exp	log	log10	log(,a)	factorial	choose
FUNCIÓN	$\sin(x)$	$\cos(x)$	$\tan(x)$	$\arcsin(x)$	$\arccos(x)$	$\arctan(x)$	$ x $
Signo	sin	cos	tan	asin	acos	atan	abs

Tabla 6: Funciones matemáticas básicas.

Veamos algunos ejemplos de las anteriores funciones.

```
sqrt(16)
## [1] 4
sqrt(16) - 16^(1 / 2) # chequeamos que es lo mismo!
## [1] 0
exp(1) # el número e
## [1] 2.718282
log(exp(2))
## [1] 2
log10(1000)
## [1] 3
log(8, base = 2)
## [1] 3
factorial(4)
## [1] 24
choose(4, 2)
## [1] 6
sin(pi / 2) # notar que el ángulo de estar medido en radianes
## [1] 1
cos(2 * pi)
## [1] 1
tan(pi / 4)
## [1] 1
asin(1) # pi/2, 5pi/2, 9pi/2, ...
## [1] 1.570796
acos(1) # 0, 2pi, 4pi, ...
## [1] 0
atan(1) # pi/4, 9pi/4, ...
## [1] 0.7853982
abs(-1)
## [1] 1
abs(1)
## [1] 1
```

Como lo comentamos, R espera que los argumentos de las funciones trigonométricas (sin, cos y tan) estén dados en radianes. Por lo tanto, si queremos aplicar estas funciones a ángulos medidos en grados, primero deberemos pasar los grados a radianes (multiplicando por  $\pi/180$ ). Por ejemplo,

```
sin(90) # seno del ángulo de 90 radianes
## [1] 0.8939967
sin(90 * pi / 180) # seno del ángulo de 90 grados
## [1] 1
asin(1) # arcoseno de 1 en radianes
## [1] 1.570796
asin(1) * 180 / pi # arcoseno de 1 en grados
## [1] 90
asin(4)
## [1] NaN
```

Observemos que el último resultado es NaN (acrónimo de *Not a Number*). ¿Qué significa esto? Significa que el resultado no existe, pues no existe ningún ángulo cuyo seno sea igual a 4 (sabemos que las funciones seno y coseno están acotadas entre -1 y 1).

Observemos lo siguiente:

```
tan(pi / 2) # esto debería ser infinito, pero...  
## [1] 1.633124e+16  
sqrt(2)^2 - 2 # esto debería ser 0, pero...  
## [1] 4.440892e-16  
sqrt(4)^2 - 4 # esto si da 0!  
## [1] 0
```

¿Qué ocurre? Lo que está pasando es que R opera numéricamente y no formalmente. Luego, resultados que no son exactos los aproxima y por ello al resolver  $(\sqrt{2})^2 - 2$  no nos devuelve el valor exacto que es 0, sino uno aproximado ( $\approx 4.4 \cdot 10^{-16}$ ). En el caso de  $(\sqrt{4})^2 - 4$  si nos devuelve 0 pues al calcular  $\sqrt{4}$  el resultado es exacto y no aproximado. Lo mismo ocurre con el resultado de  $\tan(\pi/2)$ , que sabemos que debería ser infinito, pero al tomar un valor aproximado para  $\pi/2$ , puede calcular la tangente y devolver un resultado -grande, pero finito-.

Otras funciones básicas son que, a menudo, resultan útiles:

`round(x)`: redondea a x. Si queremos redondear a cierto número de decimales, por ejemplo 2 decimales, debemos indicar un segundo argumento `digits = 2`

```
round(2.3465, digits = 2)  
## [1] 2.35
```

`floor(x)`: devuelve el mayor entero menor o igual que x.

```
floor(2.3465)  
## [1] 2  
floor(-2.3465)  
## [1] -3
```

`ceiling(x)`: devuelve el menor entero mayor o igual que x.

```
ceiling(2.3465)  
## [1] 3  
ceiling(-2.3465)  
## [1] -2
```

`trunc(x)`: devuelve la parte entera de `x`, eliminando la parte decimal.

```
trunc(2.3465)
## [1] 2
trunc(-2.3465)
## [1] -2
```

## 4.3. Estructuras de control

Las estructuras de control son construcciones sintácticas que nos permiten controlar la manera en que se ejecuta nuestro código (conocido como *Control Flow*), es decir, dirigen el flujo de operaciones y controlan la ejecución del código en una dirección u otra. Si no existieran las estructuras de control, los programas se ejecutarían linealmente desde el principio hasta el fin, sin la posibilidad de tomar decisiones.

Por lo general, en la mayoría de los lenguajes de programación encontraremos dos tipos de estructuras de control:

- ▶ **condicionales**: un tipo que permite la ejecución condicional de bloques de código,
- ▶ **iterativas**: permiten la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición.

De acuerdo con el manual base de R, las estructuras condicionales corresponden a las palabras reservadas `if` y su variante con `if - else`. Por otro parte, las estructuras para bucles son `for`, `while` y `repeat`, con las cláusulas adicionales `break` y `next`. Para consultar las diferentes estructuras de control disponibles en R, podemos escribir la instrucción `?Control` en la consola de R/RStudio.

### 1.3.1. Estructuras condicionales

Las estructuras condicionales permiten que un programa decida de manera automática entre varias opciones en función de si se cumplen o no determinadas condiciones.

Por ejemplo, cuando estamos frente a un semáforo evaluamos una de dos condiciones: si el semáforo está en verde seguimos nuestro camino y, en caso contrario (si está en amarillo o rojo), reducimos nuestra velocidad hasta detenernos.

## Sentencia if

Una declaración if le dice a R que haga una tarea determinada para un caso determinado. En palabras diríamos algo como: “Si esto es cierto, haz aquello”. En R, decimos:

```
# if
if (condición) {
  realizar una acción
}
```

El objeto `condición` debe ser una prueba lógica o una expresión que R evalúe con un solo TRUE o FALSE. Si `condición` se evalúa como TRUE, R ejecutará todo el código que **aparece entre las llaves** que siguen a la instrucción if (es decir, entre los símbolos { y }). Si `condición` se evalúa como FALSE, R omitirá el código entre llaves sin ejecutarlo.

Es importante que observes la sintaxis. La sentencia comienza con la palabra reservada if, seguida de la expresión lógica encerrada entre paréntesis. Después, encerrada entre llaves ({}), viene el conjunto de instrucciones a ejecutar si el resultado de evaluar la expresión lógica es verdadero.

Por ejemplo, podríamos utilizar una sentencia if para asegurar que algún objeto num sea positivo:

```
if (num < 0) {
  num <- -num
}
```

Si `num < 0` es verdadero (TRUE), R devolverá el opuesto de num (-num), lo que hará num positivo:

```
num <- -2 # asignamos a num el valor -2
if (num < 0) {
  num <- -num
}
num
## [1] 2 # la sentencia if cambió num a 2
```

Si `num < 0` es falso (FALSE), R no hará nada y `num` permanecerá como está: positivo (o cero):

```
num <- 4 # asignamos a num el valor 4
if (num < 0) {
  num <- -num
}
num
## [1] 4
```

## Sentencia `if... else`

La sentencia `else` complementa un `if`, pues indica qué ocurrirá cuando la condición no se cumple, es decir cuando la condición es falsa (FALSE), en lugar de no hacer nada.

En español diríamos: “Si esto es cierto, haz el plan A; de lo contrario haz el plan B.”

En R, decimos:

```
# if ... else
if (condición) {
  realizar una acción
} else { # es decir, si condición es falsa,
  realizar una acción alternativa
}
```

Cuando `condición` se evalúa como TRUE, R ejecutará el código en el primer conjunto de llaves, pero no el código en el segundo. Cuando `condición` se evalúa como FALSE, R ejecutará el código en el segundo conjunto de llaves, pero no en el primero. Podemos utilizar este tipo de construcción para cubrir todos los casos posibles.

Notemos como es la sintaxis en este caso. La parte del `if` es igual que antes. La parte `else` (en otro caso) incluye otro conjunto de instrucciones (encerradas entre llaves) que se ejecutan si la expresión lógica es falsa. Además, notar que la sentencia `else` debe colocarse en la misma línea (y a continuación de) que la llave de cierre que marca el final de la acción que se ejecuta cuando `condición` se evalúa como TRUE.

Como ejemplo, vamos a escribir un código que *redondee* un decimal al entero más cercano.

```
# redondear al entero más cercano
a <- 5.67 # comenzamos con un decimal (el que vamos a redondear)
```



```

a_ent <- trunc(a) # la parte entera de a
dec <- a - a_ent  # la parte decimal
if (dec >= 0.5) {
  a <- a_ent + 1
} else {
  a <- a_ent
}
a
## [1] 6

# Lo probamos con otro número, en este caso, a <- 5.24
a <- 5.24
a_ent <- trunc(a)
dec <- a - a_ent
if (dec >= 0.5) {
  a <- a_ent + 1
} else {
  a <- a_ent
}
a
## [1] 5

```

Utilizamos `if` (solo) cuando deseamos que una operación se ejecute únicamente cuando una condición se cumple, mientras que `if... else` es usado para indicarle a R qué hacer en caso de la condición de un `if` no se cumpla.

Si una situación tiene más de dos casos que mutuamente excluyentes, podemos unir varias declaraciones `if... else` agregando una declaración `if` nueva inmediatamente después de `else`. Por ejemplo:

Ejemplo de sentencia `if` compuesta (con 3 casos mutuamente excluyentes):

```

a <- 1
b <- 1
if (a > b) {
  print("a gana!")
} else if (a < b) {
  print("b gana!")
} else {
  print("Esto es un empate :)")
}
## [1] "Esto es un empate :("

```

Como vemos, la forma de agregar condiciones múltiples es utilizando la sentencia `else if`, tantas veces como número de casos/condiciones mutuamente excluyentes

tengamos menos dos (menos dos porque una condición va con el `if` y otra con el último `else`).

Ejemplo:

Vamos a programar un condicional que nos muestre la condición final de un alumno en un curso según su calificación final.

Primero, comenzamos creando la variable `calificacion <- 8.9` que va a ser el valor que vamos a comparar en nuestras condiciones.

Teniendo en cuenta que la calificación aprobatoria es 5, tenemos las siguientes posibilidades:

- ▶ Si la calificación es mayor o igual a 5 está aprobado,  
`if (calificacion >= 5) {print("Aprobado")}`
- ▶ en caso contrario, el alumno está reprobado.  
`else {print("Reprobado")}`

El código podría ser:

```
calificacion <- 8.9
if (calificacion >= 5) {
  print("Aprobado")
} else {
  print("Reprobado")
}
## [1] "Aprobado"
```

Como podemos observar, para la calificación 8.9 el resultado es Aprobado.

Supongamos que dentro de las personas reprobadas queremos identificar a los estudiantes que tienen calificaciones menores a 3 para darles asesorías personalizadas el siguiente semestre.

En este caso tenemos tres escenarios:

- ▶ Si la calificación es mayor o igual a 5 está aprobado  
`else if (calificacion >= 5){print ("Aprobado")}`
- ▶ Si la calificación está entre 3 y 5 está reprobado:  
`else if (calificacion < 5 & calificacion > 3){print ("Reprobado")}`
- ▶ En caso de que no se cumplan las primeras dos condiciones, el alumno requiere asesoría personalizada:  
`else {print ("Asesoría personalizada")}`

Corroboramos nuestro código corriéndolo para distintos valores de calificación: con `calificacion <- 8.9` obtenemos "Aprobado", con `calificacion <- 4.5` obtenemos "Reprobado" y con `calificacion <- 2.5` obtenemos "Asesoría personalizada".

```
calificacion <- 8.9 # 4.5, 2.5
if (calificacion >= 5) {
```

```

  print("Aprobado")
} else if (calificacion < 5 & calificacion > 3) {
  print("Reprobado")
} else {
  "Asesoría personalizada"
}
## [1] "Aprobado"

```

Repitamos una vez más: para la condición inicial siempre se escribe `if`, para las condiciones de en medio `else if` y para la final `else`.

### Limitaciones de la sentencia `if`

De acuerdo con la documentación, la sentencia `if`, la condición debe ser un vector lógico de **longitud uno** que no sea `NA`. A partir de la versión R 4.2.0, las condiciones de longitud superior a uno son un error (en versiones anteriores no devolvía un error, sino que evaluaba solamente la primera componente del vector y ejecutaba la instrucción `if` con ese valor lógico, adjuntando un mensaje de advertencia).

Ejemplo: condición de longitud mayor que uno.

Supongamos que tenemos un vector de números enteros, y queremos un procedimiento que nos diga si cada elemento del vector es par o impar. Vamos a intentarlo con `if... else`

```

x <- c(5, 3, 2, 8, -4, 1) # el vector de números enteros
if (x %% 2 == 0) {
  "Es par"
} else {
  "Es impar"
}

```

**Error in if (x%%2 == 0) { : the condition has length > 1**

R nos devuelve un mensaje de error, indicando que la condición que le pasamos en la sentencia `if` tiene longitud mayor que 1. En efecto,

```

x %% 2 == 0
# [1] FALSE FALSE TRUE TRUE TRUE FALSE

```

Por supuesto que una posible solución sería hacer la inspección del vector componente a componente (esto es, aplicar la sentencia `if... else` a cada componente de `x`), pero esto implicaría la utilización de algún *bucle de repetición* (que veremos a continuación).

## Sentencia ifelse

R base incluye la función `ifelse()` nos permite vectorizar `if`, `else` (y con ella podemos resolver el ejemplo anterior). En lugar de escribir una línea de código para cada comparación, podemos usar una sola llamada a esta función, que se aplicará a todos los elementos de un vector.

La sintaxis de la función es:

```
ifelse(test, yes, no)
```

Como podemos ver, la función tiene tres parámetros: un vector de valores lógicos (`test`) y dos vectores de expresiones (`yes` y `no`). Si los vectores de expresiones tienen menos elementos que el vector lógico, se reciclan a la longitud del vector lógico.

La función `ifelse()` devuelve como resultado un vector del mismo tamaño que el vector de valores lógicos usado como primer argumento (condición). El funcionamiento es el siguiente: para los índices del vector lógico (primer argumento) con valores `TRUE` se usa el elemento del mismo índice del primer vector de expresiones (`yes`). En caso de que el valor sea `FALSE` se usa el elemento del mismo índice del segundo vector de expresiones (`no`).

Ejemplo: (*continuación*)

Recordemos que tenemos el vector de números enteros

```
x <- c(5, 3, 2, 8, -4, 1)
```

y queremos decir de cada número si es par o impar. Utilizando la función `ifelse()` resulta:

```
ifelse(x %% 2 == 0, 'Es par', 'Es impar')  
## [1] "Es impar" "Es impar" "Es par" "Es par" "Es par" "Es impar"
```

Como vemos en el ejemplo, `ifelse` nos devuelve un valor para cada elemento del vector en el que la condición sea `TRUE`, y otro valor para los elementos en que la condición sea `FALSE`. En este ejemplo como los vectores de expresiones son ambos de longitud 1, se reciclan a la longitud del vector lógico `x %% 2 == 0`.

También podemos evaluar varias condiciones utilizando anidaciones con la función `ifelse`. Debemos tener en cuenta que se pueden anidar hasta 50 condiciones de esta forma. La sintaxis para estas formas anidadas es:

```
ifelse(vector_condicional,
       ifelse(vector_condicional_si_TRUE, value_if_TRUE, valor_si_FALSE),
       ifelse(vector_condicional_si_FALSE, value_if_TRUE, valor_si_FALSE)
)
```

Continuando con el ejemplo, podemos pedirle al código que, en caso de que el número sea par, chequee si es también divisible por 4. En cuyo caso tendríamos las siguientes sentencias `ifelse` anidadas:

```
ifelse(x %% 2 == 0, # condición primera: ver si el número es par
       ifelse(x %% 4 == 0, "es múltiplo de 4", "es múltiplo de 2 pero no de 4"),
       "es impar", # si es par, se fija si también es múltiplo de 4
               # resultado si no es par
       )
## [1] "es impar" "es impar"
## [3] "es múltiplo de 2 pero no de 4" "es múltiplo de 4"
## [5] "es múltiplo de 4" "es impar"
```

Veamos un segundo ejemplo. Supongamos que tenemos una base de datos en la cual, entre otras variables, tenemos el sexo y edad de los individuos. Queremos condensar la información de estas dos variables en una nueva que indique si el individuo es hombre o mujer y si es adulto o menor de edad.

```
set.seed(15)
sexo = sample(c('hombre', 'mujer'), 10, replace = T)
edad = sample(13:45, 10, replace = T)
datos <- data.frame(Sexo = sexo, Edad = edad)
datos
##      Sexo Edad
## 1 hombre  24
## 2 hombre  32
## 3  mujer  13
## 4  mujer  24
## 5 hombre  35
## 6  mujer  14
## 7 hombre  37
## 8 hombre  22
## 9 hombre  38
## 10 hombre  33
```

Ahora vamos a crear la nueva variable, a partir de Sexo y Edad, que clasifique a los individuos en hombre/mujer mayor/menor de edad

```
datos$clasif <- with(datos, ifelse(Sexo == "hombre",  
  ifelse(Edad >= 18, "Hombre adulto", "Hombre menor de edad"),  
  ifelse(Edad >= 18, "Mujer adulta", "Mujer menor de edad")  
))  
datos  
##      Sexo Edad      clasif  
## 1 hombre  24  Hombre adulto  
## 2 hombre  32  Hombre adulto  
## 3  mujer  13 Mujer menor de edad  
## 4  mujer  24  Mujer adulta  
## 5 hombre  35  Hombre adulto  
## 6  mujer  14 Mujer menor de edad  
## 7 hombre  37  Hombre adulto  
## 8 hombre  22  Hombre adulto  
## 9 hombre  38  Hombre adulto  
## 10 hombre  33  Hombre adulto
```

### 1.3.2. Estructuras iterativas

Las estructuras iterativas, también llamadas repetitivas o cíclicas, facilitan la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición. En este apartado estudiaremos las distintas instrucciones de R relacionadas con la iteración.

#### Sentencia for

El bucle for es una estructura iterativa que se ejecuta un número preestablecido de veces, que es controlado por un *contador* o *índice*, incrementado en cada iteración. Es la estructura que nos permite iterar sobre los elementos de un vector. Tiene la siguiente forma básica:

```
for (elemento in vector) realizar_la_acción_con_item
```

Con la expresión anterior le estamos diciendo a R: **Para** cada **elemento en vector**, repite la acción. En cada iteración del lazo for, `realizar_la_acción_con_item` se llama una vez, actualizando el valor de **elemento** al mismo tiempo.

Veamos un ejemplo sencillo, para obtener el doble de los primeros tres naturales.

```
for(i in 1:3) {  
  print(2 * i)  
}  
## [1] 2  
## [1] 4  
## [1] 6
```

En este ejemplo, `elemento` es la variable `i`, `vector` es la secuencia `1:3`, y la acción es multiplicar por 2. Observa que el conjunto de instrucciones se ha ejecutado una vez por cada elemento del vector y que la variable `i` ha ido tomando en cada ejecución uno de los valores del vector, del primero al último.

Es práctica usual, al iterar sobre un vector de índices, usar nombres de variables (índice de iteración) muy cortos como `i`, `j` o `k`.

Observación: la sentencia `for` asigna `elemento` al entorno de trabajo actual, sobrescribiendo cualquier variable que pudiera existir con el mismo nombre.

```
i = 4      # asignamos a la variable i el valor 4  
i  
## [1] 4  
for(i in 1:3) {  
  print(2 * i)  
}  
## [1] 2  
## [1] 4  
## [1] 6  
i  
## [1] 3    # Al finalizar el bucle, i toma el valor 3, último valor  
            # que se le asigna
```

Si nos interesa iterar por los índices de un vector podemos usar la función `seq_along()` para generar sus índices:

```
v <- c(-2, 1, 4, sqrt(55), -11, 2.3)  
seq_along(v)  
## [1] 1 2 3 4 5 6  
for (idx in seq_along(v)) {  
  cat(idx, '--', v[idx], '\n') # cat es para concatenar e imprimir  
}  
## 1 -- -2  
## 2 -- 1  
## 3 -- 4
```

```
## 4 -- 7.416198
## 5 -- -11
```

## Usando el lazo for

Debemos tener en cuenta que, de acuerdo a su documentación, el lazo for retorna NULL. Esto significa que, para ver y/o guardar los resultados obtenidos en cada iteración del lazo, deberemos indicarlo expresamente entre las instrucciones. Veamos un ejemplo: supongamos que queremos obtener los cuadrados de los primeros 10 números naturales.

```
for(i in 1:10) {
  i^2
}
```

Ejecutar estas líneas pareciera no producir ningún efecto o resultado (no se muestra nada en consola), pero podemos chequear que efectivamente el lazo se ejecutó, constatando que, en el entorno de trabajo, se ha creado la variable `i` (cuyo valor actual es 10)

Como solución, si lo único que queremos es mostrar los resultados, podemos utilizar la función `print()`, que imprimirá los resultados de cada iteración en la consola.

Por otro lado, podría interesarnos guardar estos resultados para utilizarlos luego, Supongamos por ejemplo que queremos obtener la suma de los cuadrados de los primeros diez naturales. Aunque hay formas más sencillas de hacerlo, hagamos esta tarea utilizando un lazo for para crear un vector con los cuadrados de los diez primeros naturales y luego sumemos esos valores.

```
cuadrados <- rep(NA,10) # inicializamos un vector vacío de longitud 10
for(i in 1:10) {
  cuadrados[i] <- i^2 # en cada iteración calculamos el cuadrado de i
                    # y lo guardamos en la posición i de cuadrados
}
```



En este caso, aunque no se muestren los valores en consola, se guardaron en el objeto (vector) `cuadrados`.

```
cuadrados
## [1] 1 4 9 16 25 36 49 64 81 100
# hallamos su suma
sum(cuadrados)
## [1] 385
```

## Lazo for y vectorización

Podemos comprobar fácilmente que el resultado en `cuadrados` es exactamente el mismo que obtenemos al ejecutar (simplemente)

```
v <- (1:10)^2
v
## [1] 1 4 9 16 25 36 49 64 81 100
```

Bien, es preciso mencionar que, dado que en R contamos con vectorización de operaciones, que podemos usar las funciones de la familia `apply` (que veremos en próximos temas) en objetos diferentes a vectores y que la manera de recuperar los resultados de un `for` es un tanto laboriosa, este tipo de lazo no es muy popular en R. En R generalmente hay opciones mejores, en cuanto a simplicidad y velocidad de cómputo, que un bucle `for`.

Sin embargo, es conveniente conocer esta estructura de control, pues habrá ocasiones en la que será la mejor herramienta para algunos problemas específicos.

## Sentencia `while` y `repeat`

Los lazos `for` son útiles si conocemos de antemano el conjunto de valores que queremos iterar. Si no es el caso, hay dos herramientas relacionadas con especificaciones más flexibles:

- ▶ `while` (*condición*) acción: realiza acción **mientras** *condición* es verdadera (`TRUE`).
- ▶ `repeat` (acción): se repite acción *para siempre*, a menos que le indiquemos la parada con un `break`.

Podemos reescribir cualquier bucle `for` para usar `while` en su lugar, y podemos reescribir cualquier bucle `while` para usar `repeat`, pero lo contrario no es cierto. Eso significa que `while` es más flexible que `for`, y `repeat` es más flexible que `while`.

Veamos cómo funcionan y utilizamos estas sentencias.

## While

Este es un tipo de bucle que ocurre mientras una condición es verdadera (`TRUE`). Es decir, la acción se realiza hasta que se llega a cumplir un criterio **previamente establecido**.

La sintaxis para este bucle es:

```
while (condición) {  
  operaciones  
}
```

o, en forma compacta,

```
while (condición) acción
```

Con esto le decimos a R: **MIENTRAS** esta condición sea **VERDADERA**, **haz** estas operaciones.

La condición generalmente es expresada como el resultado de una o varias operaciones de comparación, pero también puede ser el resultado de una función.

Al igual que la sentencia `for`, `while` retorna `NULL`. Luego si queremos que se muestre y/o guarde el resultado de cada iteración, debemos indicarlo expresamente.

Ejemplo: tirar un dado hasta obtener un 6.

Vamos a simular el experimento de tirar un dado hasta obtener 6. Notemos que no sabemos cuántas veces será necesario arrojar el dado hasta que salga 6. Luego no escribir nuestra simulación utilizando el lazo `for` (desconocemos la cantidad de iteraciones necesarias). Es un caso en el que tenemos que emplear un lazo `while`.

```
set.seed(67)
```

```

dado <- sample (1:6, 1) # tiramos una vez el dado
dado
## [1] 3
# y repetimos hasta obtener el 6
tiradas <- dado      # acá vamos a guardar los resultados que
                    # van saliendo
while (dado != 6) {
  dado <- sample(1:6, 1) # vuelvo a tirar el dado
  tiradas <- c(tiradas, dado) # agregamos el valor obtenido
}
dado
## [1] 6 # al salir del lazo el valor de dado es 6
tiradas
## [1] 3 4 1 6 # efectivamente el lazo se terminó al primer 6

```

**Ejercicio:** modifica el código anterior para simular el experimento de arrojar un dado hasta obtener el segundo 6. Ayuda: la función `any()` podría serte útil.

Para finalizar, merece la pena remarcar que, al utilizar una sentencia `while`, debemos tener cuidado con crear bucles infinitos. Si ejecutas un `while` con una condición que nunca será `FALSE`, este nunca se detendrá.

Por ejemplo, si corres lo siguiente, presiona la tecla `ESC` para detener la ejecución, de otro modo, correrá por siempre y puede llegar a congelar tu equipo.

```

while (1<2) {
  print("Presiona ESC para detener")
}

```

## Repeat

Este es un bucle que se llevará a cabo el número indeterminado de veces, usando un `break` para detenerse. Esto es, un bucle `repeat` repetirá un bloque de código hasta que le digas que se detenga (pulsando `ESC`) o hasta que encuentre el comando `break`, que detendrá el ciclo.

La estructura de `repeat` es la siguiente:

```

repeat {
  operaciones
  break_para_detenerse
}

```

Si no incluimos un `break`, el bucle se repetirá indefinidamente y sólo lo podremos detener pulsando la tecla `ESC`, así que hay que tener cuidado al usar esta estructura de control.

Como ejemplo, usemos una declaración `repeat` para dividir por 2 a un valor (positivo) hasta que sea menor que 0.5.

```
valor <- 10      # valor inicial
res <- valor     # vamos a guardar el valor en cada paso
repeat{
  valor <- valor / 2
  res <- c(res, valor) # adjuntamos el valor en la iteración al vector
                        # de resultados parciales
  if(valor < 0.5) {
    break
  }
}
valor
## [1] 0.3125
res
## [1] 10.0000  5.0000  2.5000  1.2500  0.6250  0.3125
```

Este tipo de bucle es quizás el menos utilizado de todos, pues en R existen alternativas para obtener los mismos resultados de manera más sencilla y sin el riesgo de crear un bucle infinito.

### **break y next**

`break` y `next` son palabras reservadas en R, no podemos asignarles nuevos valores y realizan una operación específica cuando aparecen en nuestro código.

Ya vimos que, por ejemplo, utilizamos `break` para finalizar un bucle `repeat`.

En general, `break` nos permite interrumpir un bucle, mientras que `next` nos deja avanzar a la siguiente iteración del bucle, “saltándose” la actual. Veamos algunos ejemplos:

```
for (i in 1:10) {
  if (i %% 2 != 0) # esto saltea los impares
    next
}
```

```

print(i)

if (i >= 8)      # esto finaliza el bucle
  break
}
## [1] 2
## [1] 4
## [1] 6
## [1] 8

```

Obviamente que podemos utilizar los dos, uno o ninguno de los comandos `next` y `break`.

```

# Ejemplo con next
for (i in 1:10) {
  if (i %% 2 != 0) # esto saltea los impares
    next

  print(i)
}
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10

# Ejemplo con break
for (i in 1:10) {
  print(i)
  if (i >= 5)      # esto finaliza el bucle
    break
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5

```

Y finalmente, dentro de un bucle `while`:

```

iter <- 0

while (iter <= 10) {
  iter <- iter + 1
  if(iter %% 2 !=0)
    next

  print(iter)
}

```

```
    if (iter >= 8)
      break
  }

## [1] 2
## [1] 4
## [1] 6
## [1] 8
```

Llegados a este punto quizás debemos aclarar algo: la indentación en el código no es estrictamente necesaria:

```
iter <- 0

while (iter <= 10) {
  iter <- iter + 1
  if(iter %% 2 !=0)
    next

  print(iter)

  if (iter >= 8)
    break
}

## [1] 2
## [1] 4
## [1] 6
## [1] 8
```

pero se recomienda fuertemente seguir ciertas guías de estilo (como indentar con dos espacios en blanco) para que el código resulte más fácil de leer por otros y por nosotros mismos. En este [link](#), encontrarás una guía de estilo si te interesa ahondar en ello. Esta guía de estilo te ayudará a decidir dónde usar espacios, cómo indentar el código y cómo usar corchetes `[]` y `{}` llaves, entre otras cosas. Si todo eso te parece demasiado trabajo, puedes instalar el paquete `{styler}` que incluye un complemento (*addins*) de RStudio que permite cambiar automáticamente el estilo del código seleccionado (o archivos y proyectos completos) con un simple clic del mouse. Para más información sobre el paquete `{styler}` mira [aquí](#). Una vez instalado, simplemente seleccionamos el código que queremos cambiar de estilo, hacemos clic en el botón *Addins* en la barra de herramientas general de RStudio y seleccionamos la opción *'Style selection'*.

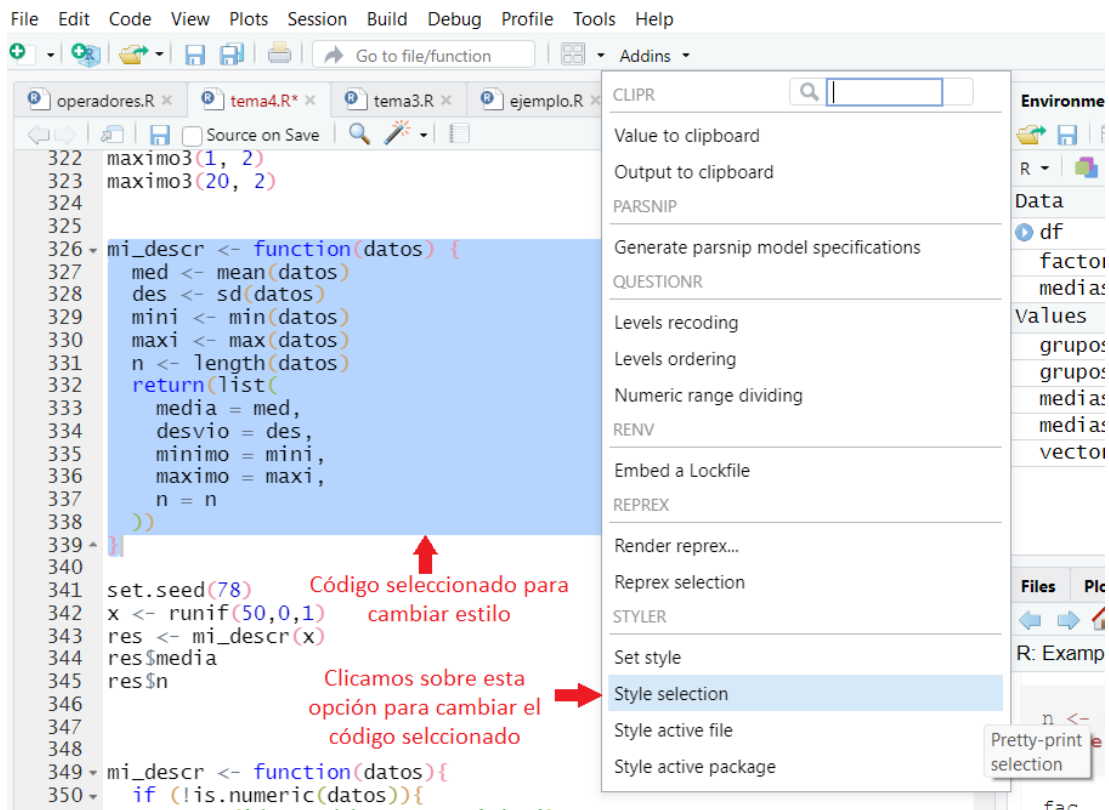


Figura 2: Uso del Addins de styler para cambiar estilo de código.

### 1.3.3. La familia apply de funciones

Incluimos en este apartado una breve referencia a esta familia de funciones que están estrechamente vinculadas con las estructuras iterativas.

La familia de funciones **apply** es usada para **aplicar una función a cada elemento de una estructura de datos**, evitando el uso de bucles **for**. En particular, es usada para aplicar funciones en matrices, *data frames*, *arrays* y listas.

Con esta familia de funciones podemos **automatizar tareas complejas** usando pocas líneas de código y es una de las características distintivas de R como lenguaje de programación.

Cabe señalar que la aplicación de estas funciones no conduce necesariamente a una ejecución más rápida (las diferencias no son enormes); más bien evita la codificación de bucles complicados o pesados, lo que reduce la posibilidad de errores.

Las funciones dentro de la familia son: `apply`, `eapply`, `sapply`, `lapply`, `mapply`, `rapply`, `tapply`, `vapply`.

Es una familia numerosa y esta variedad de funciones se debe a que varias de ellas tienen aplicaciones sumamente específicas.

Todas las funciones de esta familia tienen una característica en común: reciben como argumentos a un objeto y al menos una función.

Hasta ahora, todas las funciones que hemos usado han recibido como argumentos estructuras de datos, sean vectores, *data frames* o de otro tipo. Las funciones de la familia `apply` tienen la particularidad que pueden recibir a otra función como un argumento y es por eso por lo que pertenecen a los llamados '[funcionales](#)' (Wickham, H. (2019)). Lo anterior puede sonar confuso, pero es más bien intuitivo al verlo implementado.

En este apunte, estudiaremos tres de las funciones más generales y de uso común de esta familia: `apply()`, `lapply()` y `tapply()`.

Estas funciones nos permitirán solucionar casi todos los problemas a los que nos encontremos. Además, conociendo su uso, las demás funciones de la familia `apply` serán relativamente fáciles de entender.

### La función `apply()`

La función `apply()`, en su forma de utilización más sencilla, se utiliza para evaluar una función sobre los márgenes (1 = filas ó 2 = columnas) de un *array*, *data frame* o una matriz.

La sintaxis de esta función es:

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```



donde:

- ▶ X: es un *array*, *data frame* o matriz.
- ▶ MARGIN: 1: filas, 2: columnas, c(1, 2): filas y columnas
- ▶ FUN: función a ser aplicada
- ▶ ...: argumentos adicionales para ser pasados a FUN
- ▶ simplify: valor lógico que indica si los deben simplificarse si es posible

Para ejemplificar el uso de esta función, vamos a trabajar con los datos *iris* (que ya utilizamos en el tema anterior, al introducir la estructura *data frame*).

### Aplicar una función a cada columna

Si queremos aplicar una función a cada columna de un *array*, *data frame* o matriz, utilizamos el argumento `MARGIN = 2`.

Ejemplo: aplicar una función a cada columna de un dataset

Supongamos que queremos calcular las sumas marginales de las primeras 4 columnas del *dataset* *iris*. Sabemos que la función `sum()` de R base nos permite sumar los elementos de un vector que le ingresamos como argumento. Ahora queremos aplicar esa función a cada columna de *iris*.

```
apply(X = iris[, -5], MARGIN = 2, FUN = sum)
## Sepal.Length Sepal.Width Petal.Length Petal.Width
##           876.5           458.6           563.7           179.9
```

Como podemos observar, la salida es un vector que contiene la suma correspondiente de cada columna.

Si el objeto *x* sobre el cual aplicamos la función es un *array*, entonces el resultado es un vector cuyas componentes son las sumas de las sumas de las columnas de cada matriz componente.

```
a1 <- array(data=1:24, dim = c(3, 4, 2))
# a1 es una array con 2 matrices componentes de 3 x 4
apply(a1, MARGIN = 2, FUN = sum)
## [1] 48 66 84 102
# aplicando la función a la 1era matriz componente
apply(a1[, , 1], 2, sum)
```

```
## [1] 6 15 24 33
# aplicando la función a la 2da matriz componente
apply(a1[,2], 2, sum)
## [1] 42 51 60 69
```

Podemos comprobar fácilmente que el resultado de `apply(a1, MARGIN = 2, FUN = sum)` es igual a la suma de `apply(a1[,1], 2, sum)` más `apply(a1[,2], 2, sum)`.

Para más detalles sobre la función y valores de salida podemos consultar su documentación ejecutando `?apply` o `help(apply)`.

### Aplicar una función a cada fila

Para aplicar una función a cada fila de un array, *data frame* o matriz, utilizamos el argumento `MARGIN = 1`.

Ejemplo: aplicar una función a cada columna de un *dataset*

Supongamos que queremos calcular las sumas marginales de las primeras 4 columnas del *dataset* iris. Sabemos que la función `sum()` de R base nos permite sumar los elementos de un vector que le ingresamos como argumento. Ahora queremos aplicar esa función a cada columna de iris.

```
suma.filas <- apply(X = iris[, -5], MARGIN = 1, FUN = sum)
class(suma.filas)
## [1] "numeric"
length(suma.filas)
## [1] 150
suma.filas[1:4]
## [1] 10.2 9.5 9.4 9.4
cbind(iris[1:4, -5], suma.filas[1:4])
##   Sepal.Length Sepal.Width Petal.Length Petal.Width suma.filas[1:4]
## 1      5.1      +      3.5      +      1.4      +      0.2      =      10.2
## 2      4.9      3.0      1.4      0.2      9.5
## 3      4.7      3.2      1.3      0.2      9.4
## 4      4.6      3.1      1.5      0.2      9.4
```

Como podemos observar, la salida de nuevo es un vector que contiene la suma correspondiente a cada fila y, por lo tanto, tiene longitud 150 (la cantidad de filas del *dataset*).

## Argumentos adicionales de la función `apply`

Supongamos que la función que queremos aplicar a cada columna (o fila) es la función `mean()` que tiene un argumento adicional (`na.rm`) para especificar si se eliminan los valores de tipo `NA` o no. En tal caso, si necesitamos especificar argumentos de la función que estamos aplicando, podemos pasarlos separados por comas de la siguiente manera:

```
apply(X = iris, MARGIN = 2, FUN = mean, na.rm = TRUE)
```

Ejecutar esta instrucción, aplicará la función `mean()` con el argumento `na.rm = TRUE`.

## La función `tapply()`

`tapply()` aplica una función a un vector en los subvectores que define otro vector factor. Así, permite crear resúmenes de grupos basados en niveles del factor.

La sintaxis de esta función es:

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- ▶ `X`: es un objeto divisible (matriz, *data frame*, ...)
- ▶ `INDEX`: Lista(s) de factores de la misma longitud que `X`
- ▶ `FUN`: función a ser aplicada
- ▶ `...`: argumentos adicionales para ser pasados a `FUN`
- ▶ `default`: Si `simplify = TRUE`, es el valor de inicialización del array
- ▶ `simplify`: valor lógico que indica si los deben simplificarse si es posible

Usualmente se especifican solamente los tres primeros argumentos y es común no especificar el nombre de los argumentos en las funciones de la familia `apply` debido a su simple sintaxis.

### Ejemplo: medias por especies

Volviendo sobre los datos `iris`, supongamos que queremos ahora hacer un resumen numérico de cada variable, pero diferenciando por especie de flor. Calculemos por ejemplo la media de cada variable dentro de cada grupo (grupo = especie de flor)

```
tapply(iris$Petal.Length, iris$Species, mean)
##      setosa versicolor  virginica
##      1.462      4.260      5.552
```

Como vemos, `tapply()` aplica la función `mean()` a cada subvector del vector “longitud del pétalo” determinado por cada especie. De esta manera calcula la longitud de pétalo media por especie.

De forma completamente análoga obtenemos las medias por especie de las demás variables:

```
tapply(iris$Petal.Width, iris$Species, mean)
tapply(iris$Sepal.Length, iris$Species, mean)
tapply(iris$Sepal.Width, iris$Species, mean)
```

Notar que los argumentos de la función `tapply()` deben tener la misma *longitud*. Puedes verificarlo con la función `length`. También debe tenerse en cuenta que la salida predeterminada es de clase *array*.

```
media_PL <- tapply(iris$Petal.Length, iris$Species, mean)
length(iris$Petal.Length)
## [1] 150
length(iris$Species)
## [1] 150
class(media_PL)
## [1] "array"
```

### `tapply()` con múltiples factores

De acuerdo a la documentación de la función `tapply()`, podemos utilizar múltiples factores pasándolos a través de una lista. Veamos un ejemplo.

### Ejemplo: múltiples factores

Para ejemplificar el uso de varios factores para definir los grupos sobre los cuales queremos calcular una función, vamos a crear tres variables: una numérica y dos factores.

```
set.seed(45) # para que sea reproducible
vector <- rnorm(n = 30, mean = 1, sd = 5)
grupos1 <- as.factor(sample(1:5, size = 30, replace = TRUE))
grupos2 <- as.factor(sample(c("A", "B", "C"), size = 30, replace = TRUE))
```

Ahora que ya tenemos nuestros datos, vamos a calcular la cantidad de observaciones en cada grupo. Notar que ahora los grupos están determinados por todas las combinaciones posibles de grupo1 y grupo2

```
tapply(vector, list(grupos1, grupos2), length)
```

```
##      A  B  C
## 1  5  2  2
## 2  4 NA  2
## 3  4  2  2
## 4  2  2 NA
## 5 NA  1  2
```

Observemos que, por ejemplo, la combinación 2-B no tiene elementos y, por defecto, `tapply()` muestra NA. Podemos cambiar este comportamiento predeterminado especificando el argumento `default = 0` (pues eso es justamente lo que está pasando, que hay 0 observaciones en ese grupo)

```
tapply(vector, list(grupos1, grupos2), length, default = 0)
```

```
##      A B C
## 1  5  2  2
## 2  4  0  2
## 3  4  2  2
## 4  2  2  0
## 5  0  1  2
```

Mejor, ¿no?

## La función `lapply()`

La última función de la familia que veremos es la función `lapply()`.

`lapply()` es un caso especial de `apply()`, diseñado para aplicar funciones a todos los elementos de una **lista**. Tiene el beneficio adicional de que generará los resultados como una lista también.

`lapply()` intentará coercionar a una lista el objeto que demos como argumento y después aplicará una función a todos sus elementos. Dado que en R todas las estructuras de datos pueden coercionarse a una lista, `lapply()` puede usarse en un

número más amplio de casos que `apply()`, además de que esto nos permite utilizar funciones que aceptan argumentos que no sean vectores.

La sintaxis de esta función es:

```
lapply(X, FUN, ...)
```

donde:

- ▶ `x`: es una lista o un objeto coercionable a una lista
- ▶ `FUN`: es la función a aplicar
- ▶ `...`: argumentos adicionales para ser pasados a `FUN`

Si nos fijamos, estos argumentos son idénticos a los de `apply()`, excepto que aquí no especificamos `MARGIN`, pues las listas son estructuras con una unidimensionales, que sólo tienen longitud (`length`).

Veamos algunos ejemplos de usos de la función.

### **`lapply()` en vector**

Ejemplo:

Supongamos que queremos calcular el logaritmo natural de cada uno de los elementos del vector `x <- c(5, 10, 23)`.

```
# Definimos el vector x
x <- c(10, 25, 30)
# calculamos el logaritmo natural de cada elemento de x
l1 <- lapply(x, log)
class(l1)
## [1] "list"
v1 <- log(x)
class(v1)
## [1] "numeric"
l1
## [[1]]
## [1] 2.302585
##
## [[2]]
## [1] 3.218876
##
## [[3]]
```

```
## [1] 3.401197
##
v1
## [1] 2.302585 3.218876 3.401197
```

Como podemos ver, dado que la operación `log()` se vectoriza al aplicarla sobre el vector, también podemos calcular el logaritmo de cada elemento de `x` simplemente aplicando la función `log()` al vector `x`. La diferencia es que `lapply()` crea como objeto de salida una lista, mientras que `log(x)` nos devuelve un vector.

### **`lapply()` en un *data frame***

Vamos a crear un *data frame* para ejemplificar el uso de la `lapply()` en estas estructuras de datos.

```
v1 <- c("Juan", "Gloria", "Olivia", "María", "Esteban")
v2 <- factor(c("M", "F", "F", "F", "M"))
v3 <- c(165, 158, 160, 157, 155)
v4 <- c(72, 65, 69, 58, 49)
df <- data.frame(Nombre = v1, Sexo = v2, Altura = v3, Peso = v4)
df
##      Nombre Sexo Altura Peso
## 1    Juan     M    165    72
## 2  Gloria     F    158    65
## 3  Olivia     F    160    69
## 4   María     F    157    58
## 5 Esteban     M    155    49
```

Supongamos que queremos verificar la clase de todas las columnas de un marco de datos. Esto lo podemos hacer fácilmente utilizando `lapply()` en el *data frame*, especificando la función `class` como argumento:

```
lapply(df, class)
## $Nombre
## [1] "character"
##
## $Sexo
## [1] "factor"
##
## $Altura
## [1] "numeric"
##
## $Peso
## [1] "numeric"
```

## lapply() en listas

Para ejemplificar el uso de la función `lapply()` en listas, vamos a crear una lista de la siguiente manera:

```
P <- c(10, 12, 28)
Q <- 1:5
R <- 11:15
lista1 <- list(P, df = data.frame(Q, R))
lista1
## [[1]]
## [1] 10 12 28
##
## $df
##   Q  R
## 1 1 11
## 2 2 12
## 3 3 13
## 4 4 14
## 5 5 15
```

Ahora usamos `lapply()` para obtener la suma de los elementos de cada componente de la lista.

```
lapply(lista1, sum)
## [[1]]
## [1] 50
##
## $df
## [1] 80
```

Notemos que el resultado de nuevo es una lista de longitud 2, cuyos elementos corresponden a la suma de los elementos de `lista1`.

Observación: para finalizar, merece la pena destacar dos comentarios:

1. Si aplicamos la función `lapply()` a una matriz, FUN se aplicará sobre cada celda de la matriz y no sobre cada columna.
2. `lapply()`, a diferencia de `apply()` no puede aplicarse a renglones.

```
m1 <- matrix(1:6, ncol=2)
m1
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

lapply(m1, mean)
```



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

Si queremos calcular las medias por columnas, primero convertimos a *data frame*

```
lapply(as.data.frame(m1), mean)
## $V1
## [1] 2
##
## $V2
## [1] 5
```

Si queremos calcular las medias por filas, primero transponemos la matriz y luego convertimos a *data frame*

```
lapply(as.data.frame(t(m1)), mean)
## $V1
## [1] 2.5
##
## $V2
## [1] 3.5
##
## $V3
## [1] 4.5
```

### Extra gift: la función `sapply()`

Esta función está estrechamente relacionada con la función `lapply()`, de hecho, es una versión simplificada de `lapply`. Es decir, `sapply` aplica la función `lapply` y estudia la salida. Cuando entiende que dicha salida admite una representación menos

aparatosas que una lista, la simplifica. Esto es, por defecto y si es posible, devuelve un vector, una matriz o un *array*, si `simplify = "array"`. Por ejemplo,

```
sapply(airquality[, 1:4], mean, na.rm = TRUE)
##      Ozone      Solar.R      Wind      Temp
## 42.129310 185.931507   9.957516 77.882353
```

En este ejemplo, en lugar de una lista, como hace `lapply`, `sapply` devuelve una representación más natural: un vector de números.

De acuerdo a lo que podemos leer en su documentación, `sapply(X, FUN, simplify = FALSE, USE.NAMES = FALSE)` es exactamente lo mismo que `lapply(X, FUN)`.

```
sapply(X = airquality[, 1:4], FUN = mean, simplify = FALSE, USE.NAMES
= FALSE)
## $Ozone
## [1] 42.12931
##
## $Solar.R
## [1] 185.9315
##
## $Wind
## [1] 9.957516
##
## $Temp
## [1] 77.88235

lapply(X = airquality[, 1:4], FUN = mean)
## $Ozone
## [1] 42.12931
##
## $Solar.R
## [1] 185.9315
##
## $Wind
## [1] 9.957516
##
## $Temp
## [1] 77.88235
```

## 4.4. Funciones

Una función permite escribir un fragmento de código parametrizado. De esta forma, es posible escribir **un bloque de código** y ejecutarlo para **distintos datos**. Una función puede considerarse un subcódigo que resuelve una subtarea. Un motivo para utilizar funciones es que permiten estructurar u organizar el código de un programa.

En temas previos ya hemos utilizado muchas funciones de R, unas simples como la función `+` para añadir números, otras más complejas como `c()` o `data.frame()` que permiten crear un vector o un *data frame*.

En este apartado, aprenderemos cómo crear nuestras propias funciones.

### 4.4.1. ¿Por qué crear funciones?

Deberías considerar escribir una función cuando has copiado y pegado un bloque de código más de dos veces (es decir, ahora tienes tres copias del mismo). Mira, por ejemplo, el siguiente código. ¿Qué es lo que hace?

Imaginemos que tenemos el siguiente *data frame*

```
df <- data.frame(  
  a = rnorm(10),  
  b = rnorm(10, mean = 0, sd = 2),  
  c = rnorm(10, mean = 0, sd = 0.5),  
  d = rnorm(10, mean = -1, sd = 1)  
)
```

y queremos estandarizar los datos, esto es, queremos que cada columna tenga media 0 y varianza 1. Para lograr esto necesitamos transformar cada columna restándole su media y dividiéndola por su desviación típica. Esto es,

```
df$a.new <- (df$a - mean(df$a)) / sd(df$a)  
df$b.new <- (df$b - mean(df$b)) / sd(df$b)  
df$c.new <- (df$c - mean(df$a)) / sd(df$c)  
df$d.new <- (df$d - mean(df$d)) / sd(df$d)
```

¿Pueden detectar el error en el código anterior? Por causa de utilizar el copiar-y-pegar al código para `df$c.new`, nos olvidamos de cambiar a por `c`. Esto podría evitarse si, en lugar de copiar-y-pegar bloques de código, utilizamos una función. Por supuesto que este ejemplo tiene soluciones ya programadas en R (simplemente `scale(df)` ¡hace el trabajo!), pero intenta ilustrar la necesidad de contar con la posibilidad de crear nuestras propias funciones que resuelvan tareas específicas.

#### 4.4.2. ¿Cómo crear funciones en R?

Una función tiene un nombre, argumentos y un cuerpo. La estructura general de las funciones definidas por el usuario es:

```
nombre <- function(argumentos) {  
  cuerpo  
}
```

Para ejemplificar el procedimiento, vamos a crear una función que encuentre el máximo entre dos valores dados.

```
maximo <- function(a, b) {  
  if (a > b) {  
    m <- a  
  } else {  
    m <- b  
  }  
  m  
}  
maximo(a = -7, b = 2)  # [1] 2  
y <- 18  
maximo(a = 20, b = y + 7)  # [1] 25
```

Diagrama de la función `maximo`:

- Nombre:** `maximo`
- Argumentos:** `a, b`
- Cuerpo:** El código entre llaves que define la lógica para encontrar el máximo.

Una llamada a la función: `maximo(a = -7, b = 2)`

Como observamos, para crear una función se usa la palabra reservada `function`. A continuación (y encerrados entre paréntesis), se enumeran los argumentos o parámetros *formales* de la función separados por comas (en el ejemplo los argumentos son `a` y `b`). Después viene el cuerpo de la función. El cuerpo contiene, encerrado entre llaves, el código con todas las operaciones que se ejecutarán cada vez que se llame a la función. Con esto se crea un *objeto* función que se asigna a la

variable `maximo`. Cuando asignamos una función a un nombre decimos que hemos definido una función.

El nombre que asignamos a una función nos permite ejecutarla y hacer referencias a ella. Podemos asignar la misma función a diferentes nombres o cambiar una función a la que ya le hemos asignado un nombre. Como siempre, es recomendable elegir nombres claros, no ambiguos y descriptivos.

En el ejemplo aparecen dos llamadas a la función. En la primera se usan los valores de argumentos o parámetros *reales* `-7` y `2`, y en la segunda `20` y la expresión `y + 7`. Al invocar a una función se evalúan las expresiones que conforman los argumentos y se emparejan con los parámetros formales, es decir, a cada parámetro formal se le asigna el resultado de evaluar la expresión usada como parámetro real. Los parámetros formales son variables que reciben la información de entrada de la función (en el ejemplo, `a` y `b`). Una vez emparejados parámetros reales con formales se ejecutan las instrucciones de la función. Una función devuelve como salida el resultado de evaluar la última instrucción que ejecuta. En el ejemplo, la última instrucción ejecutada es `m`, que es una expresión que produce el máximo de los dos valores de entrada de la función (almacenados en `a` y `b`).

En R, podemos especificar los argumentos *por posición*. El orden de los argumentos se determina cuando creamos una función. En nuestro ejemplo, determinamos que el primer argumento que recibe `maximo` es `a` y el segundo es `b`. Así, podemos escribir lo siguiente y obtener el resultado esperado:

```
maximo(20, y + 7) # el 1er argumento es a = 20 y el 2do b = y + 7
## [1] 25
```

En este ejemplo por supuesto que es indistinto que valor se asigna a `a` y cuál se asigna a `b`, pero existen casos en los cuales esto sí resulta importante y debemos tener cuidado. Por ejemplo, al utilizar la función `rnorm()` (ya implementada en R). De acuerdo con la documentación de la función (`?rnorm`), debemos proveerle los argumentos `n`, `mean` y `sd`. Esto lo podemos hacerlo indicado expresamente cada

argumento por su nombre o bien, por posición. Pero en este último caso, deberemos respetar el orden de dichos argumentos en la definición de la función. Por ejemplo, si utilizamos `rnorm(20, -1, 1)`, R entenderá que queremos una muestra aleatoria de tamaño 20 de una distribución normal con media -1 y desvío estándar 1, mientras que, utilizando el nombre de cada uno de los argumentos, podemos introducirlos en cualquier orden: `rnorm(mean = -1, sd = 1, n = 20)` hará el mismo trabajo (esto es, proporcionar una muestra aleatoria de tamaño 20 de una distribución normal con media -1 y desvío estándar 1).

### Para tener en cuenta

Para conocer los argumentos formales de una función, podemos utilizar la función `formals()`.

```
formals(maximo)
## $a
##
## $b
```

Para conocer el código que conforma el cuerpo de una función, podemos utilizar la función `body()`.

```
body(maximo)
## {
##   if (a > b) {
##     m <- a
##   }
##   else {
##     m <- b
##   }
##   m
## }
```

Si en la consola escribimos el nombre de una función, se mostrará el código de la función. Y esto vale tanto para las funciones creadas por nosotros como para la mayoría de las funciones incorporadas en R (existen funciones en R llamadas primitivas, escritas en lenguaje C, y éstas no se mostrarán).

```

> maximo
function(a, b) {
  if (a > b) {
    m <- a
  } else {
    m <- b
  }
  m
}
<bytecode: 0x000001b50dc2e620>
> colSums
function (x, na.rm = FALSE, dims = 1L)
{
  if (is.data.frame(x))
    x <- as.matrix(x)
  if (!is.array(x) || length(dn <- dim(x)) < 2L)
    stop("'x' must be an array of at least two dimensions")
  if (dims < 1L || dims > length(dn) - 1L)
    stop("invalid 'dims'")
  n <- prod(dn[id <- seq_len(dims)])
  dn <- dn[-id]
  z <- if (is.complex(x))
    .Internal(colSums(Re(x), n, prod(dn), na.rm)) + (0+1i) *
      .Internal(colSums(Im(x), n, prod(dn), na.rm))
  else .Internal(colSums(x, n, prod(dn), na.rm))
  if (length(dn) > 1L) {
    dim(z) <- dn
    dimnames(z) <- dimnames(x)[-id]
  }
  else names(z) <- dimnames(x)[[dims + 1L]]
  z
}
<bytecode: 0x000001b576b28210>
<environment: namespace:base>
> sum
function (... , na.rm = FALSE) .Primitive("sum")
>

```

Código de la función **maximo**

Código de la función **colSums**

No se muestra código, función primitiva

Figura 3: Código de funciones mostrados en consola

Esto es una herramienta poderosísima, pues, además de permitirnos ver ejemplos de definiciones de funciones (que resulta especialmente útil cuando queremos crear una), nos permite comprender los resultados obtenidos al ejecutar una función mediante el estudio de su código.

#### 4.4.3. La función `return()`

Por defecto, las funciones de R devolverán el último objeto evaluado dentro de ella. También podemos hacer uso de la función `return()`, que tiene un solo parámetro y produce el siguiente efecto: termina la ejecución de la función en la que se ejecuta y el valor devuelto por la función es el parámetro con que es invocada. Es una función especialmente importante cuando se quiere devolver un objeto u otro dependiendo

de ciertas condiciones o cuando se quiere ejecutar algún código después del objeto que queremos devolver. Por ejemplo, nuestra función `maximo`, podría escribirse así:

```
maximo3 <- function(a, b) {  
  if (a > b) {  
    return(a)  
  } else {  
    return(b)  
  }  
}  
maximo3(2, 7)  
## [1] 7  
maximo3(9, -6)  
## [1] 9
```

Otra utilidad, quizás mucho más interesante, tiene que ver con la posibilidad de devolver más de un objeto en una función. Con una función podemos devolver todos los tipos de objetos de R, pero solo uno de ellos. Por ello, si queremos que una función devuelva más de un objeto, podemos conseguirlo haciendo que, el objeto devuelto, sean una lista de objetos (una lista por es la única estructura capaz de contener objetos de distinto tipo y tamaño). Veamos un ejemplo:

Ejemplo: Resumen numérico de variables cuantitativas.

Vamos a crear una función que obtenga las principales medidas resumen de variables cuantitativas. Es decir, como argumento le pasaremos un vector de datos, y la función nos devolverá el mínimo, máximo, desviación estándar, media, mediana y cantidad de datos.

```
mi_descr <- function(datos) {  
  med <- mean(datos)  
  des <- sd(datos)  
  mini <- min(datos)  
  maxi <- max(datos)  
  n <- length(datos)  
  return(list(  
    media = med,  
    desvio = des,  
    minimo = mini,  
    maximo = maxi,  
    n = n  
  ))  
}
```

la función devuelve una lista de objetos



Para probar nuestra función, vamos a generar una muestra aleatoria de números del intervalo (0; 1) (que asignamos a la variable `x`) para luego aplicarle `mi_descr` para obtener los principales estadísticos descriptivos.

```
set.seed(78)
x <- runif(50, 0, 1)
res <- mi_descr(x)
res
## $media
## [1] 0.4738317
##
## $desvio
## [1] 0.2903592
##
## $minimo
## [1] 0.006186323
##
## $maximo
## [1] 0.9794657
##
## $n
## [1] 50
```

Podrías comprobar que en el ejemplo anterior utilizar o no la función `return()` no cambia nada. Sin embargo, supongamos que agregar a la función una instancia donde chequee que el vector de datos ingresado es adecuado. Para ello, vamos a testear datos es de tipo `numeric`, devolviendo un mensaje error y deteniendo la ejecución del código en caso contrario.

Ejemplo: Resumen numérico de variables cuantitativas (*continuación*).

```
mi_descr <- function(datos) {
  if (!is.numeric(datos)) {
    return('datos debe ser numérico')
  }
  med <- mean(datos)
  des <- sd(datos)
  mini <- min(datos)
  maxi <- max(datos)
  n <- length(datos)
  return(list(
    media = med,
    desvio = des,
    minimo = mini,
    maximo = maxi,
    n = n
  ))
}
```

```

    ))
  }
  set.seed(78)
  x <- runif(50, 0, 1)
  res <- mi_descr(x)
  names(res)
  ## [1] "media" "desvio" "minimo" "maximo" "n"
  res$media
  ## [1] 0.4738317

  x <- c(x, 'hola')
  res <- mi_descr(x)
  names(res)
  ## NULL
  res$media
  Error in res$media : $ operator is invalid for atomic vectors
  res
  ## [1] "datos debe ser numérico"

```

En el ejemplo último, incluir la instancia de testeo del tipo del objeto datos, interrumpe la ejecución del código si el resultado del testeo es FALSE, evitando de este modo incurrir en errores. Si, por ejemplo, si en lugar de usar la función `return()` hubiésemos usado la función `print()`, si se pasa un vector de datos que no sea numérico se devolverá el texto "datos debe ser numérico" pero también un error, ya que se ejecutará todo el código.

Ejemplo: Resumen numérico de variables cuantitativas (*continuación*).

```

mi_descr2 <- function(datos) {
  if (!is.numeric(datos)) {
    print('datos debe ser numérico')
  }
  med <- mean(datos)
  des <- sd(datos)
  mini <- min(datos)
  maxi <- max(datos)
  n <- length(datos)
  list(
    media = med,
    desvio = des,
    minimo = mini,
    maximo = maxi,
    n = n
  )
}
res2 <- mi_descr2(x)

```

```
## [1] "datos debe ser numérico"
Warning messages:
1: In mean.default(datos) :
  argument is not numeric or logical: returning NA
2: In var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm
= na.rm) :
  NAs introduced by coercion
```

#### 4.4.4. Variables locales y globales

Una función puede realizar cálculos complejos, por lo que a veces necesita usar variables para almacenar los resultados intermedios de sus cálculos. Por ejemplo, la siguiente función calcula la suma de los elementos de un vector:

```
f <- function(v) {
  n <- length(v)
  acum <- rep(NA, n)
  s <- 0
  for(i in seq_along(v)) {
    s <- s + v[i]
    acum[i] <- s
  }
  return(list(
    suma = s,
    acumuladas = acum,
    n = n
  ))
}
# probamos nuestra función
v <- c(0.25, 0.15, 0.10, 0.35, 0.10, 0.05)
f(v)
## $suma
## [1] 1
##
## $acumuladas
## [1] 0.25 0.40 0.50 0.85 0.95 1.00
##
## $n
## [1] 6
```

#### Variables locales

Las variables creadas en una función se llaman **variables locales**. En el ejemplo, la variable `n`, `acum` y `s` son variables locales que se usan para calcular la longitud del vector, guardar las sumas acumuladas de los elementos del vector y calcular la suma

acumulada total de los elementos del vector. Las variables locales, junto con los parámetros formales, se crean al ejecutarse la función y normalmente desaparecen al terminar su ejecución. Las variables locales y parámetros formales son independientes de otras variables que puedan existir fuera de la función con el mismo nombre. Por ejemplo:

```
n = 4
> f(v)
## $suma
## [1] 1
##
## $acumuladas
## [1] 0.25 0.40 0.50 0.85 0.95 1.00
##
## $n
## [1] 6
acum
Error: object 'acum' not found # solo existe en el entorno de la
función
```

## Variables globales

Otra posibilidad es que una función utilice variables que han sido definida fuera de ésta. Por ejemplo:

```
s <- 0 ----▶ variable global
g <- function(v) {
  n <- length(v)
  for(i in seq_along(v)) {
    s <- s + v[i]
  }
  s
}
v <- c(0.25, 0.15, 0.10, 0.35, 0.10, 0.05)
g(v)
## [1] 1
s
## [1] 0 # luego de ejecutar g, s permanece inalterada
```

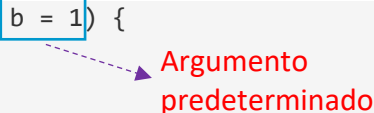
La función `g` usa el valor de la variable `s` que no es una variable local, sino que es **global** a la función (ha sido creada fuera). El valor de la variable `s` en el entorno global de trabajo, no cambia. La función la utiliza, pero no la modifica.

En general, el uso de variables globales no es considerada una buena práctica de programación, por lo que no las usaremos. Pensemos que esta posibilidad de que R busque el valor asociado a un nombre es una “invitación al error” pero, en líneas generales, no causa demasiados problemas (especialmente si reinicias regularmente R para hacer borrón y cuenta nueva).

#### 4.4.5. Argumentos predeterminados

En ocasiones resulta útil incluir valores predeterminados a los parámetros formales de las funciones. En tal caso, al llamar a la función, se utilizaron esos valores predeterminados a menos que se especifiquen otros valores al ejecutar la función. Por ejemplo, reescribamos la función `maximo`, de modo tal que el segundo valor, `b`, sea un argumento predeterminado:

```
maximo2 <- function(a, b = 1) {  
  if (a > b) {  
    m <- a  
  } else {  
    m <- b  
  }  
  m  
}  
  
# al pasar un solo argumento, asigna el valor a a. Esto es equivalente  
# a máximo(3, 1) o maximo2(3, 1)  
maximo2(3)  
## [1] 3  
maximo2(3, 1)  
## [1] 3  
maximo(3, 1)  
## [1] 3  
# pero  
maximo(3)  
Error in maximo(3) : argument "b" is missing, with no default  
  
# al pasarle dos argumentos, asigna el 1ero 3 a a y el 2do a b.  
maximo2(3, 4)  
## [1] 4
```



**Nota:** La mayoría de las funciones en R poseen argumentos predeterminados. Cuando accedemos a la documentación de una función, al indicar su sintaxis, se

especifican los valores predeterminados de los argumentos que los posean. Por ejemplo, al solicitar la ayuda de la función `rnorm()`, obtenemos

```
rnorm(n, mean = 0, sd = 1)
```

Como vemos, todos los parámetros formales, salvo el primero, tienen el formato `x = valor`. Esto quiere decir que el parámetro `x` toma el valor por defecto `valor`, a menos que se especifique otro valor para ese argumento en la invocación a la función. Así, al ejecutar la función, será lo mismo usar `rnorm(n = 20, mean = 0, sd = 1)`, que usar simplemente `rnorm(n = 20)`.

#### 4.4.6. Argumentos adicionales

Al definir una función, R permite que se incluyan argumentos adicionales, indicados por `...` (tres puntos), cuya finalidad es pasar argumentos libremente que se usarán en una subfunción dentro de la función principal. Veamos un ejemplo. Supongamos que queremos crear una función que grafique el diagrama de caja y bigotes correspondiente a un conjunto de datos (que ingresaremos como argumento) que tenga cierto formato predefinido.

```
mi_boxplot <- function(datos, nombre_variable = NULL, ...) {  
  par(bty = "n", mgp = c(0, -3, -4), mar=c(1, 5, 3, 6))  
  boxplot(datos,  
    col = "#2a94c0",  
    border = "#1D6786",  
    col.axis = "#0A335D",  
    cex.axis = 0.8,  
    ...)  
  if (!is.null(nombre_variable)) {  
    titulo <- paste("Boxplot de la variable", nombre_variable)  
    title(titulo, adj = 0.4, line = 1)  
  }  
}
```

Los tres puntos indican que tenemos la posibilidad de ingresar argumentos adicionales al llamar a la función.

Al llamar a la función dentro de nuestra función, incluimos como argumento los tres puntos, que se emparejarán con los tres puntos en la definición de nuestra función

Ahora probemos nuestra función: su funcionamiento por defecto y luego de agregarle argumentos adicionales para `boxplot()`.

```
# Primero simulemos los datos que luego vamos a graficar
set.seed(2097)
muestra <- rnorm(50)

# llamamos a la función especificando solo el argumento datos
mi_boxplot(datos = muestra)
# el gráfico resultante es
```

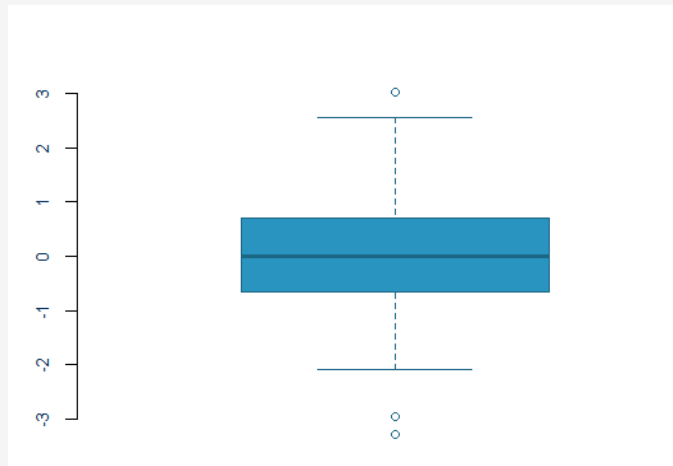


Figura 4: Argumentos adicionales: Gráfico por defecto de mi\_boxplot

```
# Agregamos el nombre de la variable que estamos graficando
mi_boxplot(muestra, 'Altura')
# el gráfico resultante es
```

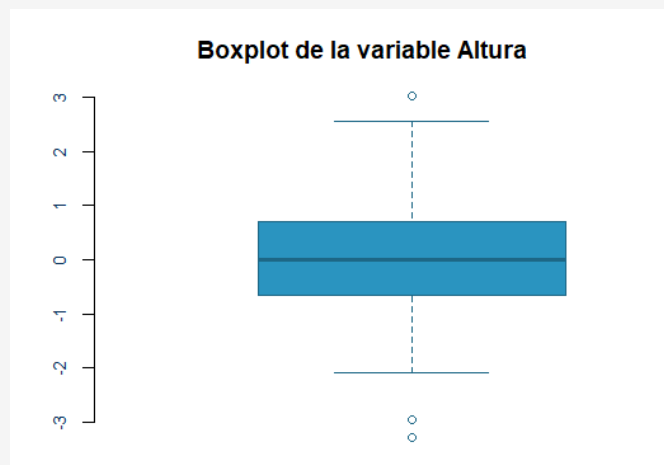


Figura 5: Argumentos adicionales: Gráfico de mi\_boxplot

```
# Incluimos argumentos para el boxplot
mi_boxplot(muestra, 'Altura', horizontal = TRUE)
# el gráfico resultante es
```

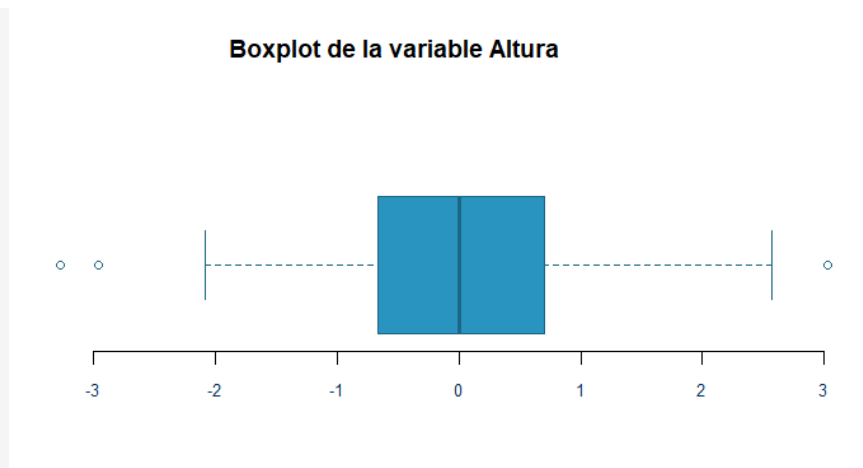


Figura 6: Argumentos adicionales: Gráfico de `mi_boxplot`

```
mi_boxplot(muestra, 'Altura', notch = TRUE, boxwex = 0.5)
# el gráfico resultante es
```

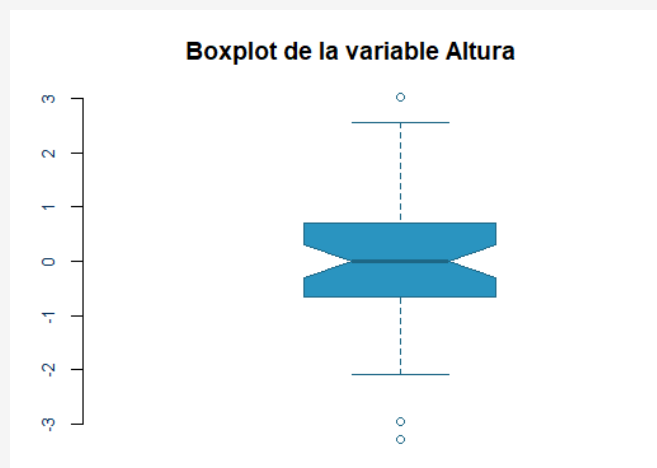


Figura 7: Argumentos adicionales: Gráfico de `mi_boxplot`

#### 4.4.7. Funciones anónimas

Hasta ahora al definir las funciones siempre hemos asignado el objeto función creado a un identificador. Una función anónima es aquella que no tiene un identificador asociado. En R hay ocasiones en que sólo necesitamos una función temporalmente y no vale la pena el esfuerzo de darle un nombre. En tales situaciones, podemos evitarnos el darle un nombre si usamos una función anónima. Un caso de uso frecuente para funciones anónimas está dentro de la familia `apply` de funciones de R base.



### Ejemplo sencillo

Vamos a utilizar una función anónima para sumar 5 a cada elemento de un vector.

```
vector = 1:10
sapply(vector, function(x) (x+5))
## [1] 6 7 8 9 10 11 12 13 14 15
```

Como podemos observar, hemos utilizado la función `sapply()` con el argumento `FUN` utilizando una función anónima. Obviamente podríamos haber definido la función (tal como lo hicimos en la sección anterior) y luego a `FUN` pasarle el nombre de nuestra función. Pero la idea es que, si solamente vamos a usar esta función una vez, no vale la pena invertir esfuerzo en crearla más allá del llamado en `sapply()`.

Veamos un ejemplo más. En este ejemplo queremos calcular la norma de un vector.

### Ejemplo: norma de vectores

Recordemos que la norma de un vector es igual a la raíz cuadrada de la suma de los cuadrados de sus componentes. Esto es, si  $\vec{v} = (v_1, v_2, \dots, v_n)$  es un vector de  $n$  componentes ( $\vec{v} \in \mathbb{R}^n$ ), entonces  $|\vec{v}| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$ . Bien, si queremos calcular la norma de un solo vector (digamos  $v$ ), tomando ventaja de la vectorización de las operaciones en R, bastará con hacer

```
nor_v <- sqrt(sum(v^2))
```

Pero ahora imaginemos que tenemos muchos vectores a los que necesitamos calcular su norma. Si todos los vectores tienen la misma longitud, lo podremos tener almacenados en un *data frame*, sino en una lista. Luego, utilizando `apply()` (para calcular la norma de cada columna del *data frame*) o `lapply()` (para operar sobre cada elemento de la lista) podremos hacer nuestro trabajo. Para ejemplificar, creemos un *data frame* y una lista solamente con tres vectores (pero recordar que esto muestra su mayor utilidad si tenemos que trabajar con una cantidad grande de vectores).

```
# vectores de igual longitud
df <- data.frame(v = 5:9, w = (0:4)^2, s = -1:3)
apply(df, 2, function(x) { sqrt(sum(x^2)) })
##           v           w           s
## 15.968719 18.814888  3.872983

# vectores de longitudes diferentes
ls <- list(v = 5:20, w = (0:5)^2, s = -1:3)
lapply(ls, function(x) { sqrt(sum(x^2)) })
## $v
## [1] 53.29165
```

```
##  
## $w  
## [1] 31.28898  
##  
## $s  
## [1] 3.872983
```

## 4.5. Referencias bibliográficas

Wickham, H. and Golemund, G. (2019). R para Ciencia de Datos.  
<https://es.r4ds.hadley.nz/>

Wickham, H. (2019). *Advanced R* (2nd edition). Chapman and Hall/CRC.  
<https://adv-r.hadley.nz/>

## 4.6 Cuaderno de ejercicios

### Ejercicio 1

Escribe un código que indique qué calificación tiene un alumno dada su nota numérica según la siguiente escala:

Nota numérica	Calificación
[0 ; 5)	Suspenso
[5 , 7)	Aprobado
[7 ; 9)	Notable
[9 ; 10]	Sobresaliente

Prueba el funcionamiento del código para las notas 4, 6 y 9.

## Ejercicio 2

Después de ejecutar el siguiente código, ¿cuál es el valor de x?

```
x <- 3
my_func <- function(y) {
  x <- 5
  y+5
}
```

## Ejercicio 3

Escribe un código que calcule el número inverso del producto de dos números a y b, así como el número inverso de la suma de dichos números. Ten en cuenta que primero el código deberá chequear si el inverso existe.

## Ejercicio 4

1. Escribe una función `cuadrados_n` tal que, para cada n, calcule el valor de la suma  $S_n = 1^2 + 2^2 + \dots + n^2$ .

2. Escribe una función que te permita comprobar la identidad

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

3. Modifica la función `cuadrados_n` para obtener `potencias_n` que, para cada n, calcule el valor de la suma  $1 + 2^2 + \dots + n^n$ .

## Ejercicio 5

1. Define un vector numérico vacío `s_n` de longitud 30 (por ejemplo la función `vector()` o `rep()`) y almacena los resultados de  $S_1, S_2, \dots, S_{30}$  ( $S_n$  definido como en el ejercicio 4.1) usando un lazo `for`.
2. Repite el ítem anterior, pero esta vez usando la función `sapply`.

## Ejercicio 6

Escribe un código que calcule las soluciones de una ecuación de segundo grado de la forma  $ax^2 + bx + c = 0$ , recordando que las soluciones son:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Si las soluciones son complejas no las calcules e indica un mensaje en la pantalla. También debes tener en cuenta que  $a$  puede valer 0, en cuyo caso debes resolver una ecuación lineal. Si  $a$  y  $b$  valen 0 no hay solución a menos que  $c = 0$ .

En el siguiente vídeo podrás acceder a la resolución de los ejercicios 4 y 5 del cuaderno de ejercicios:



Accede al vídeo

---