Introducción a R y RSTUDIO

Juan F. Olivares-Pacheco*

Índice

1 Introducción **2**
- 1.1 R y RStudio . . . 2
- 1.2 R: Un lenguaje para la computación estadística . . . 2
- 1.3 RStudio: Un entorno de desarrollo integrado para R . . . 3
- 1.4 Instalación de R y RStudio . . . 3
- 1.5 Primeros pasos en R y RStudio . . . 3
- 1.6 Comunidad y recursos de aprendizaje . . . 4

2 Cálculos básicos **4**
- 2.1 Sumas, restas, multiplicación y división . . . 4
- 2.2 Exponentes . . . 4
- 2.3 Constantes matemáticas . . . 5
- 2.4 Logaritmos . . . 5
- 2.5 Trigonometría . . . 5
- 2.6 Obtener ayuda . . . 5
- 2.7 Instalación de paquetes . . . 6

3 Datos **6**
- 3.1 Tipos de datos . . . 6
- 3.2 Estructura de datos . . . 7
  - 3.2.1 Vectores . . . 7
  - 3.2.2 Vectorización . . . 11
  - 3.2.3 Operadores lógicos . . . 11
  - 3.2.4 Más sobre vectorización . . . 13
  - 3.2.5 Matrices . . . 16
  - 3.2.6 Calculos con vectores y matrices . . . 22

*jfolivar@uda.cl

3.2.7 Listas 25
3.2.8 Data frames 28

4 Conceptos básicos de programación 29
4.1 Instruccion `if-else` 29
4.2 Instrucción `for` 30
4.3 Instrucción `while` 31
4.4 Instrucción `repeat` y `break` 31
4.5 Funciones 32

1 Introducción

1.1 `R` y `RStudio`

- `R` es un lenguaje de programación pensado para **computación estadística**, que se distribuye de forma libre y gratuita bajo licencia GNU, y que corre en distintas plataformas y sistemas operativos.
- `RStudio` es una aplicación, que provee un **entorno gráfico de desarrollo** (IDE) que facilita varias cosas, como la edición de código y ayuda para el autocompletado, organizar fácilmente los archivos, y visualizar objetos en memoria, recurrir a la ayuda, etc.

1.2 `R`: Un lenguaje para la computación estadística

- **Orientación estadística:** Incluye un conjunto de funciones estadísticas, desde estadística descriptiva hasta técnicas avanzadas.
- **Capacidad gráfica y de visualización:** Proporciona librerías integradas y paquetes adicionales para la creación de gráficos de alta calidad.
- **Arquitectura extensible mediante paquetes:** Se organiza en torno a paquetes distribuídos principalmente a través de CRAN (Comprehensive R Archive Network).
- **Compatibilidad multiplataforma:** Es compatible con Windows, macOS y Linux.
- **Integración con otros lenguajes y entornos:** Puede integrarse con `C`, `C++`, `Fortran` y `Python`, así como con sistemas distribuidos y herramientas de big data.

- **Lenguaje vectorizado:** Facilita la manipulación de datos gracias a su naturaleza vectorizada, permitiendo ejecutar operaciones sobre vectores y matrices de manera eficiente.

1.3 `RStudio`: Un entorno de desarrollo integrado para `R`

- **Editor de código avanzado:** Con resaltado de sintaxis y autocompletado, el editor facilita la construcción de scripts complejos.
- **Consola interactiva:** Permite ejecutar instrucciones de `R` en tiempo real, ideal para depurar, probar y explorar datos rápidamente.
- **Herramientas de proyecto y organización:** Facilita la gestión de proyectos, directorios y archivos, permitiendo mantener el flujo de trabajo organizado.
- **Visores integrados de datos y gráficos:** Presenta paneles dedicados para visualizar el entorno, para explorar datos y gráficos sin salir del IDE.
- **Integración con control de versiones (Git, SVN):** Permite trabajar colaborativamente y asegurar la reproducibilidad.
- **Documentación y ayuda en línea:** Acceso rápido a documentación oficial, paquetes y funciones mediante atajos de teclado y menús contextuales.

1.4 Instalación de `R` y `RStudio`

- **Instalar `R` primero:** Visitar el sitio oficial de `R` (`https://www.r-project.org/`) y seleccionar un mirror de CRAN. Descargar el instalador adecuado a su sistema operativo (Windows, macOS, Linux) y seguir las instrucciones predeterminadas.
- **Instalar RStudio posteriormente:** Visitar el sitio oficial de `RStudio` (`https://posit.co/`) y descargar la versión deseada. La versión gratuita es suficiente para la mayoría de los análisis. Tras la instalación, `RStudio` detectará automáticamente `R`.

1.5 Primeros pasos en `R` y `RStudio`

- **Creación de un script:** Desde el menú "Archivo" > "Nuevo archivo" > "Script R". Esto abrirá un editor donde se puede escribir y guardar el código.

- **Ejecución de código:** Seleccionar las líneas de código y pulsar `Ctrl` + `Enter` (Windows) o `Cmd` + `Enter` (macOS) para ejecutarlas en la consola.
- **Instalación de paquetes:** Para instalar y utilizar paquetes especializados, se emplea la función `install.packages()` y, posteriormente, `library()` para cargarlos en la sesión actual.

1.6 Comunidad y recursos de aprendizaje

1. **Documentación oficial y CRAN:** Además de los manuales estándar, CRAN ofrece viñetas y tutoriales para los paquetes más populares.
2. **Foros y listas de correo:** Stack Overflow y las listas de correo de R son entornos activos donde se discuten problemas, se resuelven dudas y se comparten buenas prácticas.
3. **Cursos, libros y materiales de aprendizaje:** Existe una amplia gama de cursos gratuitos, libros especializados, tutoriales en línea y vídeos. La comunidad académica y profesional es muy activa, facilitando el aprendizaje continuo.
4. **Grupos locales de usuarios y conferencias:** Los `R` User Groups (RUGs) y eventos como la conferencia UseR! fomentan el intercambio de conocimiento y la puesta al día con las últimas tendencias en análisis estadístico e ingeniería de datos.

2 Cálculos básicos

2.1 Sumas, restas, multiplicación y división

Matemática	`R`	Resultado
$3 + 2$	`3 + 2`	5
$3 - 2$	`3 - 2`	1
$3 \cdot 2$	`3 * 2`	6
$3/2$	`3 / 2`	1.5

2.2 Exponentes

Matemática	R	Resultado
3^2	`3^2`	9
$2^{(-3)}$	`2^(-3)`	0.125
$100^{1/2}$	`100^(1/2)`	10
$\sqrt{100}$	`sqrt(100)`	10

2.3 Constantes matemáticas

Matemática	R	Resultado
π	`pi`	3.1415927
e	`exp(1)`	2.7182818

2.4 Logaritmos

Matemática	R	Resultado
$\log(e)$	`log(exp(1))`	1
$\log_{10}(1000)$	`log10(1000)`	3
$\log_2(8)$	`log2(8)`	3
$\log_4(16)$	`log(16, base = 4)`	2

2.5 Trigonometría

Matemática	R	Resultado
$\sin(\pi/2)$	`sin(pi/2)`	1
$\cos(0)$	`cos(0)`	1

2.6 Obtener ayuda

Para obtener documentación sobre una función R, simplemente coloca un signo de interrogación antes del nombre de la función, y RStudio mostrará la documentación.

```
?log
?sin
?paste
?lm
```

2.7 Instalación de paquetes

`R` incluye varias funciones y conjuntos de datos integrados, pero una de sus principales fortalezas como proyecto de código abierto es su sistema de paquetes. Los paquetes añaden funciones y datos adicionales. A menudo, si necesitas hacer algo en `R` y no está disponible por defecto, es muy probable que exista un paquete que cubra esa necesidad.

Para instalar un paquete, se utiliza la función `install.packages()`.

```
install.packages("ggplot2")
```

Una vez instalado un paquete, debe ser cargado en la sesión actual de `R` antes de poder usarse.

```
library(ggplot2)
```

Una vez que se cierra `R`, todos los paquetes se cierran. La próxima vez que se abra `R`, no es necesario volver a instalar el paquete, pero sí debes cargar cualquier paquete se que planee usar mediante la función `library()`.

3 Datos

3.1 Tipos de datos

`R` tiene un número de **tipos de datos** básicos.

- **Numérico:** También conocido como **Double**. Es el tipo predeterminado cuando se trabaja con números.
  - Ejemplos: `1`, `1.0`, `42.5`
- **Entero:** Se especifica agregando `L` al número.
  - Ejemplos: `1L`, `2L`, `42L`
- **Complejo:** Se utiliza para números complejos.
  - Ejemplos: `4 + 2i`
- **Lógico:** Tiene dos valores posibles: `TRUE` y `FALSE`.
  - `NA` también se considera un valor lógico.
- **Carácter:** Se utiliza para representar texto.
  - Ejemplos: `"a"`, `"Estadística"`, `"1 más 2"`

3.2 Estructura de datos

R cuenta con varias estructuras de datos básicas. Una estructura da datos puede ser:

- **Homogénea:** Todos los elementos son del mismo tipo de datos.
- **Heterogénea:** Los elementos pueden ser de más de un tipo de datos.

Dimensión	Homogénea	Heterogénea
1	Vector	Lista
2	Matriz	Data frame
3+	Arreglo	

3.2.1 Vectores

Muchas operaciones en R hacen un uso intensivo de los vectores. En R, los vectores están indexados comenzando en 1. El `[1]` en la salida indica que el primer elemento de la fila mostrada es el primer elemento del vector. Para vectores más grandes, las filas adicionales comenzarán con `[*]`, donde `*` es el índice del primer elemento de esa fila.

Una de las formas más comunes de crear un vector en R es utilizando la función `c()`, que es una abreviatura de **combinar**. Como su nombre lo indica, esta función combina una lista de elementos separados por coma.

```
c(1, 3, 5, 7, 9)
```

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

Aquí R simplemente muestra este vector.

Si queremos almacenar un vector en una variable, podemos hacerlo utilizando el operador asignación `<-`. En este caso, la variable `x` ahora contiene el vector que acabamos de crear, y podemos acceder a él escribiendo `x`.

```
x <- c(1, 3, 5, 7, 9)
x
```

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

Dado que los vectores en R deben contener elementos del mismo tipo, R automáticamente convertirá todos los elementos a un solo tipo cuando se intenta crear un vector que combine múltiples tipos de datos.

```r
c(42, "Estadística", TRUE)
```

```
## [1] "42"          "Estadística" "TRUE"
```

```r
c(42, TRUE)
```

```
## [1] 42  1
```

Frecuentemente, es posible que se desee crear un vector basado en un secuencia de números. La forma más rápida y sencilla de hacerlo es utilizando el operador `:`, que genera una secuencia de enteros entre dos números especificados.

```r
y <- 1:100
y
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13
##  [14]  14  15  16  17  18  19  20  21  22  23  24  25  26
##  [27]  27  28  29  30  31  32  33  34  35  36  37  38  39
##  [40]  40  41  42  43  44  45  46  47  48  49  50  51  52
##  [53]  53  54  55  56  57  58  59  60  61  62  63  64  65
##  [66]  66  67  68  69  70  71  72  73  74  75  76  77  78
##  [79]  79  80  81  82  83  84  85  86  87  88  89  90  91
##  [92]  92  93  94  95  96  97  98  99 100
```

Es importante notar que en `R` no existen los escalares. Un escalar simplemente se trata como un vector de longitud 1.

```r
2
```

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

Para crear un secuencia que no esté limitada a enteros consecutivos, se puede usar la función `seq()`, que permite definir una secuencia especificando su inicio, fin e incremento.

```r
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8
## [15] 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

Más adelante hablaremos en detalle sobre las funciones, pero por ahora, es importante notar que las etiquetas de entrada `from`, `to` y `by` en la función `seq()` son opcionales.

```r
seq(1.5, 4.2, 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8
## [15] 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

Otra operación común para crear un vector es la función `rep()`, que permite repetir un único valor varias veces.

```r
rep("A", times = 10)
```

```
##  [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"
```

La función `rep()` también se puede usar para repetir un **vector** un determinado número de veces.

```r
rep(x, times = 3)
```

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

Ahora hemos visto cuatro formas diferentes de crear vectores en `R`:

- `c()`
- `:`
- `seq()`
- `rep()`

Hasta ahora, hemos utilizado estas funciones principalmente de forma aislada, pero a menudo se combinan entre sí para crear y manipular vectores de manera más flexible y eficiente.

```r
c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
```

```
##  [1]  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9  1  3  5
## [19]  7  9  1  2  3 42  2  3  4
```

La longitud de un vector se puede obtener con la función `length()`.

```r
length(x)
```

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

```r
length(y)
```

```
## [1] 100
```

Para obtener un subconjunto de un vector, usamos "`[]`".

```
x
```

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

```
x[1]
```

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

```
x[3]
```

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

Podemos notar que `x[1]` devuelve el primer elemento y `x[3]` el tercer elemento.

```
x[-2]
```

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

Podemos también excluir ciertos indices, en este caso el segundo elemento.

```
x[1:3]
```

```
## [1] 1 3 5
```

```
x[c(1, 3, 4)]
```

```
## [1] 1 5 7
```

Por último, vemos que podemos extraer subconjuntos de un vector utilizando un vector de índices.

Todo lo anterior consiste en extraer subconjuntos de un vector utilizando un vector de índices. Recordemos que un solo número sigue siendo un vector en `R`.

En lugar de usar índices numéricos, también podemos utilizar un vector de valores lógicos (`TRUE` o `FALSE`) para seleccionar elementos de un vector.

```
z <- c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE)
z
```

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

```
x[z]
```

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

3.2.2 Vectorización

Una de la mayores fortalezas de R es su uso de **operaciones vectorizadas**. A menudo, la falta de compresión de este concepto lleva a la creencia errónea de que R es lento. Si bien no es el lenguaje más rápido, su reputación de ser más lento de lo que realmente es se debe en gran parte a un uso ineficiente de la vectorización.

```
x <- 1:10
x + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2^x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
##  [7] 2.645751 2.828427 3.000000 3.162278
```

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
##  [6] 1.7917595 1.9459101 2.0794415 2.1972246 2.3025851
```

Vemos que cuando se llama a una función como `log()` sobre un vector `x`, se devuelve un vector en el que la función se ha aplicado a cada elemento de `x`.

3.2.3 Operadores lógicos

Operador	Descripción	Ejemplo	Resultado
`x < y`	x es menor que y	`3 < 42`	TRUE
`x < y`	x es mayor que y	`3 > 42`	FALSE
`x <= y`	x es menor o igual que y	`3 <= 42`	TRUE
`x >= y`	x es mayor o igual que y	`3 >= 42`	FALSE
`x == y`	x es igual a y	`3 == 42`	FALSE
`x != y`	x no es igual a y	`3 != 42`	TRUE
`!x`	no x	`!(3 > 42)`	TRUE
`x | y`	x o y	`(3 > 42) | TRUE`	TRUE
`x & y`	x y y	`(3 < 4) & (42 > 13)`	TRUE

Operador	Descripción	Ejemplo	Resultado

En R, las operaciones lógicas también pueden ser vectorizadas.

```
x <- c(1, 3, 5, 7, 9, 11)
```

```
x > 3
```

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

```
x < 3
```

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

```
x == 3
```

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

```
x != 3
```

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

```
x == 3 & x != 3
```

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

```
x ==3 | x != 3
```

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

Esto es extremadamente útil para la selección de subconjuntos en un vector.

```
x[x < 3]
```

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

```
x[x != 3]
```

```
## [1]  1  5  7  9 11
```

```
sum(x > 3)
```

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

```
as.numeric(x > 3)
```

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

Aquí podemos ver que al usar la función `sum()` es un vector lógico de valores `TRUE` y `FALSE`, que es el resultado de `x > 3`, se obtiene un resultado numérico.

`R` convierte automáticamente los valores lógicos a número, donde:

- `TRUE` se convierte en `1`
- `FALSE` se convierte en `0`

Este conversión de lógico a numérico ocurre en la mayoría de las operaciones matemáticas en `R`.

```
which(x > 3)
```

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

```
x[which(x > 3)]
```

```
## [1]  5  7  9 11
```

```
max(x)
```

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

```
which(x == max(x))
```

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

```
which.max(x)
```

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

3.2.4 Más sobre vectorización

```
x <- c(1, 3, 5, 7, 8, 9)
y <- 1:100
```

```
x + 2
```

```
## [1]  3  5  7  9 10 11
```

```
x + rep(2, 6)
```

```
## [1]  3  5  7  9 10 11
```

```
x > 3
```

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

```
x > rep(3, 6)
```

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

```
x + y
```

```
## Warning in x + y: longer object length is not a multiple of
## shorter object length
```

```
##   [1]   2   5   8  11  13  15   8  11  14  17  19  21  14
##  [14]  17  20  23  25  27  20  23  26  29  31  33  26  29
##  [27]  32  35  37  39  32  35  38  41  43  45  38  41  44
##  [40]  47  49  51  44  47  50  53  55  57  50  53  56  59
##  [53]  61  63  56  59  62  65  67  69  62  65  68  71  73
##  [66]  75  68  71  74  77  79  81  74  77  80  83  85  87
##  [79]  80  83  86  89  91  93  86  89  92  95  97  99  92
##  [92]  95  98 101 103 105  98 101 104 107
```

```
length(x)
```

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

```
length(y)
```

```
## [1] 100
```

```
length(y) / length(x)
```

```
## [1] 16.66667
```

```
(x + y) - y
```

```
## Warning in x + y: longer object length is not a multiple of
## shorter object length
```

```
##   [1] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5
##  [28] 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9
##  [55] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5
##  [82] 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7
```

```
y <- 1:60
x + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27
## [19] 20 23 26 29 31 33 26 29 32 35 37 39 32 35 38 41 43 45
```

```
## [37] 38 41 44 47 49 51 44 47 50 53 55 57 50 53 56 59 61 63
## [55] 56 59 62 65 67 69
```

```
length(y) / length(x)
```

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

```
rep(x, 10) + y
```

```
##  [1]  2  5  8 11 13 15  8 11 14 17 19 21 14 17 20 23 25 27
## [19] 20 23 26 29 31 33 26 29 32 35 37 39 32 35 38 41 43 45
## [37] 38 41 44 47 49 51 44 47 50 53 55 57 50 53 56 59 61 63
## [55] 56 59 62 65 67 69
```

```
all(x + y == rep(x, 10) + y)
```

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

```
identical(x + y, rep(x, 10) + y)
```

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

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

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

```
all(x == y)
```

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

```
any(x == y)
```

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

La función `all()` devuelve `TRUE` solo cuando todos sus argumentos son `TRUE`, mientras la función `any()` devuelve `TRUE` cuando al menos uno de sus argumentos es `TRUE`.

```
x <- 10^(-8)
x
```

```
## [1] 1e-08
```

```
y <- 10^(-9)
y
```

```
## [1] 1e-09
```

```
all(x == y)
```

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

```
all.equal(x, y)
```

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

La función `all.equal()` en `R` se utiliza para verificar la "igualdad aproximada" entre valores numéricos, con una tolerancia predeterminada de aproximadamente `1.5e-8`. Devuelve `TRUE` si todas las diferencias entre los valores comparados son menores que la tolerancia establecida.

3.2.5 Matrices

`R` también se puede utilizar para cálculos con **matrices**. Las matrices tienen **filas y columnas** que contienen un único tipo de datos. En una matriz, el orden de las filas y columnas es importante.

Las matrices se pueden crear utilizando la función `matrix()`.

```
x <- 1:9
x
```

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

```
X <- matrix(x, nrow = 3, ncol = 3)
X
```

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

Aquí notamos que estamos utilizando dos variables diferentes:

- `x` en minúsculas, que almacena un vector.
- `X` en mayúsculas, que almacena una matriz.

Esto sigue la convención matemática habitual. Podemos hacer esto porque `R` distingue entre mayúsculas y minúsculas, es **case sensitive**.

Por defecto, la función `matrix()` organiza un vector en columnas, pero también podemos indicarle a `R` que lo ordene por filas usando el argumento `byrow = TRUE`.

```
Y <- matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

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

También podemos crear una matriz don dimensiones específicas donde todos los elementos sean iguales, por ejemplo, ceros.

```
Z <- matrix(0, 2, 4)
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

Al igual que los vectores, las matrices pueden ser extraídas en subconjuntos utilizando corchetes `[]`. Sin embargo, dado que las matrices son bidimensionales, es necesario especificar tanto la fila como la columna al hacer la selección.

```
X
```

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

```
X[1, 2]
```

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

Aquí accedemos al elemento en la primera fila y la segunda columna. También podemos extraer una fila completa o una columna completa de la matriz.

```
X[1, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

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

También podemos usar vectores para extraer más de una fila o columna a la vez. Aquí seleccionamos la primera y tercera columna de la segunda fila.

```
X[2, c(1, 3)]
```

```
## [1] 2 8
```

Las matrices también pueden crearse combinando vectores como columnas usando `cbind()`, o vectores usando filas usando `rbind()`.

```
x <- 1:9
rev(x)
```

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

```
rep(1, 9)
```

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

```
rbind(x, rev(x), rep(1, 9))
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## x    1    2    3    4    5    6    7    8    9
##      9    8    7    6    5    4    3    2    1
##      1    1    1    1    1    1    1    1    1
```

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 9))
```

```
##       col_1 col_2 col_3
##  [1,]     1     9     1
##  [2,]     2     8     1
##  [3,]     3     7     1
##  [4,]     4     6     1
##  [5,]     5     5     1
##  [6,]     6     4     1
##  [7,]     7     3     1
##  [8,]     8     2     1
##  [9,]     9     1     1
```

Al usar `rbind()` y `cbind()`, puedes especificar nombres de argumentos, que se utilizarán como nombres de columna.

Además, `R` permite realizar cálculos matriciales, como sumas, productos y operaciones algebraicas con matrices.

```r
x <- 1:9
y <- 9:1
X <- matrix(x, 3, 3)
Y <- matrix(y, 3, 3)
```

```r
X
```

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

```r
Y
```

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

```r
X + Y
```

```
##      [,1] [,2] [,3]
## [1,]   10   10   10
## [2,]   10   10   10
## [3,]   10   10   10
```

```r
X - Y
```

```
##      [,1] [,2] [,3]
## [1,]   -8   -2    4
## [2,]   -6    0    6
## [3,]   -4    2    8
```

```r
X * Y
```

```
##      [,1] [,2] [,3]
## [1,]    9   24   21
## [2,]   16   25   16
## [3,]   21   24    9
```

```r
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
```

```
## [3,] 0.4285714 1.5000000 9.000000
```

Notemos que, `X * Y` no es una multiplicación matricial, sino una multiplicación elemento por elemento. Lo mismo ocurre con `X / Y`. Para realizar multiplicación de matrices, se debe usar el operador `%*%`. Otras funciones útiles para matrices incluyen:

- `t()`: Devuelve la transpuesta de una matriz.
- `solve()`: Devuelve la inversa de una matriz cuadrada (si es invertible).

```
X %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   90   54   18
## [2,]  114   69   24
## [3,]  138   84   30
```

```
t(X)
```

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

```
Z <- matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

Para verificar que `solve(Z)` devuelve la matriz inversa, la multiplicamos por `Z`. Se esperaría que el resultado fuera la matriz identidad, pero debido a problemas de precisión computacional, esto no siempre ocurre exactamente.

Para manejar estas diferencias, `R` proporciona:

- `all.equal()`: Comprueba la igualdad con una pequeña tolerancia para corregir errores numéricos.
- `identical()`: Comprueba la igualdad exacta sin tolerancias.

```
solve(Z) %*% Z
```

```
##               [,1]          [,2]          [,3]
## [1,] 1.000000e+00 -6.245005e-17 0.000000e+00
## [2,] 8.326673e-17  1.000000e+00 5.551115e-17
## [3,] 2.775558e-17  0.000000e+00 1.000000e+00
```

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
all.equal(solve(Z) %*% Z, diag(3))
```

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

R tiene varias funciones específicas para matrices que permiten obtener información sobre sus dimensiones y resumen estadístico.

```
X <- matrix(1:6, 2, 3)
```

```
dim(X)
```

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

```
rowSums(X)
```

```
## [1]  9 12
```

```
colSums(X)
```

```
## [1]  3  7 11
```

```
rowMeans(X)
```

```
## [1] 3 4
```

```
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

La función `diag()` en `R` se puede usar de varias maneras. Una de ellas es extraer la diagonal de una matriz.

```
diag(Z)
```

```
## [1]  9  4 16
```

O crear una matriz con elementos específicos en la diagonal, mientras que los elementos fuera de la diagonal serán ceros.

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

Por último, la función `diag()` se puede utilizar para crear una matriz identidad, que es una matriz cuadrada con `1` en la diagonal y `0` en los elementos fuera de la diagonal.

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

3.2.6 Calculos con vectores y matrices

Ciertas operaciones en `R`, por ejemplo `%*%`, tiene un comportamiento diferente en vectores y matrices. Para ilustrar esto, primero crearemos dos vectores.

```
a_vec <- c(1, 2, 3)
b_vec <- c(2, 2, 2)
```

Note que estos son, de hecho, vectores. No son matrices.

```
c(is.vector(a_vec), is.vector(b_vec))
```

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

```r
c(is.matrix(a_vec), is.matrix(b_vec))
```

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

Cuando este es el caso, el operador `%*%` se utiliza para calcular el producto punto, también conocido como el producto interno de los dos vectores.

El producto punto de los vectores $a = [a_1, a_2, \ldots, a_n]$ y $b = [b_1, b_2, \ldots, b_n]$ se define como:

$$a \cdot b = \sum_{i=1}^{n} a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

```r
a_vec %*% b_vec
```

```
##      [,1]
## [1,]   12
```

```r
a_vec %o% b_vec
```

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

El operador `%o%` se utiliza para calcular el producto exterior de los dos vectores.

Cuando los vectores son forzados a convertirse en matrices, se convierten en vectores columna. Así que un vector de longitud n se convierte en una matriz de $n \times 1$ después de la conversión.

```r
as.matrix(a_vec)
```

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

Si usamos el operador `%*%` en matrices, `%*%` nuevamente realiza la multiplicación de matrices esperadas. Así que podrá esperar que lo siguiente produzca un error, porque las dimensiones son incorrectas.

```r
as.matrix(a_vec) %*% b_vec
```

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

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

A primera vista, esto es una matriz 3×1, multiplicada por una matriz de 3×1. Sin embargo, cuando `b_vec` es automáticamente convertido a una matriz, `R` decide convertirlo en un **vector fila**, una matriz 1×3, para que la multiplicación tenga dimensiones compatibles.

Si hubiéramos convertido ambos explícitamente, entonces `R` producirá un error.

```
as.matrix(a_vec) %*% as.matrix(b_vec)
```

```
## Error in as.matrix(a_vec) %*% as.matrix(b_vec): non-conformable arguments
```

Otra forma de calcular un producto punto es con la función `crossprod()`. Dados dos vectores, la función `crossprod()` calcula el producto punto. El nombre de la función es bastante engañoso.

```
crossprod(a_vec, b_vec)
```

```
##      [,1]
## [1,]   12
```

```
tcrossprod(a_vec, b_vec)
```

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

Estas funciones podrían ser muy útiles más adelante. Cuando se usan con matrices X e Y como argumentos, calculan:

$$X^\top Y$$

Al trabajar con modelos lineales, el cálculo $X^\top X$ se utiliza repetidamente.

```
C_mat <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
D_mat <- matrix(c(2, 2, 2, 2, 2, 2), 2, 3)
```

Esto es útil tanto como un atajo para un cálculo frecuente como una implementación más eficiente que usar `t()` y `%*%`.

```r
crossprod(C_mat, D_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```r
t(C_mat) %*% D_mat
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```r
all.equal(crossprod(C_mat, D_mat), t(C_mat) %*% D_mat)
```

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

```r
crossprod(C_mat, C_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```r
t(C_mat) %*% C_mat
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```r
all.equal(crossprod(C_mat, C_mat), t(C_mat) %*% C_mat)
```

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

3.2.7 Listas

Una lista es una estructura de datos unidimensionales heterogénea. Por tanto, se indexa como un vector con un solo valor entero, pero cada elemento puede contener un elemento de cualquier tipo.

```r
list(42, "Hola", TRUE)
```

```
## [[1]]
## [1] 42
##
## [[2]]
## [1] "Hola"
##
## [[3]]
## [1] TRUE
```

```
lista_ejemplo <- list(a = c(1, 2, 3, 4),
                      b = TRUE,
                      c = "Hola!",
                      d = function(arg = 42) print("Hola Mundo!"),
                      e = diag(5))
```

Las listas pueden ser subconjuntos usando dos sintaxis, el operador `$` y los corchetes `[]`. El operador `$` devuelve un elemento nombrado de una lista. La sintaxis `[]` devuelve una lista, mientas que `[[]]` devuelve un elemento de una lista.

- `lista_ejemplo[1]`: devuelve una lista que contiene el primer elemento.
- `lista_ejemplo[[1]]`: devuelve el primer elemento de la lista, en este caso, un vector.

```
lista_ejemplo$e
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
lista_ejemplo[1:2]
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
```

```
lista_ejemplo[1]
```

```
## $a
```

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

```
lista_ejemplo[[1]]
```

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

```
lista_ejemplo[c("e", "a")]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
##
## $a
## [1] 1 2 3 4
```

```
lista_ejemplo["e"]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
lista_ejemplo[["e"]]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
lista_ejemplo$d
```

```
## function (arg = 42)
## print("Hola Mundo!")
```

```
lista_ejemplo$d(arg = 1)
```

```
## [1] "Hola Mundo!"
```

3.2.8 Data frames

Anteriormente hemos visto vectores y matrices para almacenar datos al introducir R. Ahora introduciremos un marco de datos (data frame), que será la forma más común en que almacenaremos e interactuaremos con los datos.

```
datos_ejemplo <- data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                            y = c(rep("Hola", 9), "Adios"),
                            z = rep(c(TRUE, FALSE), 5))
```

A diferencia de una matriz, que puede considerarse como un vector reorganizado en filas y columnas, un data frame no requiere tener el mismo tipo de datos para cada elemento. Un data frame es una lista de vectores. Por lo tanto, cada vector debe contener el mismo tipo de datos, pero los diferentes vectores pueden almacenar diferentes tipos de datos.

```
datos_ejemplo
```

```
##    x     y     z
## 1  1  Hola  TRUE
## 2  3  Hola FALSE
## 3  5  Hola  TRUE
## 4  7  Hola FALSE
## 5  9  Hola  TRUE
## 6  1  Hola FALSE
## 7  3  Hola  TRUE
## 8  5  Hola FALSE
## 9  7  Hola  TRUE
## 10 9 Adios FALSE
```

A diferencia de una lista, que tiene más flexibilidad, los elementos de un data frame debe ser todos vectores y tener la misma longitud.

```
datos_ejemplo$x
```

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

```
all.equal(length(datos_ejemplo$x),
          length(datos_ejemplo$y),
```

```
          length(datos_ejemplo$z))
```

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

```
str(datos_ejemplo)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ x: num  1 3 5 7 9 1 3 5 7 9
##  $ y: chr  "Hola" "Hola" "Hola" "Hola" ...
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

```
nrow(datos_ejemplo)
```

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

```
ncol(datos_ejemplo)
```

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

```
dim(datos_ejemplo)
```

```
## [1] 10  3
```

4 Conceptos básicos de programación

4.1 Instruccion `if-else`

En `R`, la sintaxis `if-else` es:

```
if (condición) {
  # Instrucciones si la condición es TRUE
} else {
  # Instrucciones si la condición es FALSE
}
```

Por ejemplo:

```
x <- 1
y <- 3
if (x > y) {
  z <- x * y
  print("x es mayor que y")
} else {
```

```
  z <- x + 5 * y
  print("x es menor o igual que y")
}
```

```
## [1] "x es menor o igual que y"
```

```
z
```

```
## [1] 16
```

R también tiene una función especial `ifelse()` que es muy útil. Devuelve uno de dos valores especificados basándose en una declaración condicional.

```
ifelse(4 > 3, 1, 0)
```

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

El verdadero poder `ifelse()` proviene de su capacidad para ser aplicada a vectores.

```
numeros <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
ifelse(numeros %% 2 == 0, "Par", "Impar")
```

```
##  [1] "Impar" "Par"   "Impar" "Par"   "Impar" "Par"   "Impar"
##  [8] "Par"   "Impar" "Par"
```

4.2 Instrucción `for`

En R, la sintaxis de la instrucción `for` es:

```
for (i in secuencia) {
  # Instrucciones a ejecutar para cada elemento i
}
```

Por ejemplo:

```
resultados <- numeric(10)
for (i in 1:10) {
  x <- sample(1:6, size = 1)
  resultados[i] <- x
}
```

```
resultados
```

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

4.3 Instrucción `while`

La sintaxis de la instrucción `while` es:

```
while (condición) {
  # Instrucciones a ejecutar mientras la condición es TRUE
}
```

Por ejemplo:

```
numero_lanzamientos <- numero_caras <- 0
resultados <- c("Cara", "Sello")
historial <- NULL
while (numero_caras < 2) {
  resultado_lanzamiento <- sample(x = resultados, size = 1)
  numero_lanzamientos <- numero_lanzamientos + 1
  historial[numero_lanzamientos] <- resultado_lanzamiento
  if (resultado_lanzamiento == "Cara") {
    numero_caras <- numero_caras + 1
  }
}
```

```
historial
```

```
## [1] "Sello" "Sello" "Cara"  "Cara"
```

4.4 Instrucción `repeat` y `break`

La sintaxis de la instrucción `repeat` es:

```
repeat {
  # Instrucciones
  if (condición de salida) {
    break
  }
}
```

Por ejemplo:

```
x <- 1
repeat {
  print(x)
  x <- x + 1
  if (x == 5) {
```

```
    break
  }
}
```

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

4.5 Funciones

Hasta ahora hemos estado usando funciones, pero en realidad no hemos discutido algunos de sus detalles.

```
nombre_funcion <- function(arg1, arg2, ...) {
  # Instrucciones
  ...
  return(resultado)
}
```

Para usar una función, simplemente escribes su nombre, seguido de un paréntesis abierto, luego especificas los valores de sus argumentos, y terminas con un paréntesis cerrado.

Un argumento es una variable que se utiliza en el cuerpo de la función. Especificar los valores de los argumentos es esencialmente proporcionar las entradas a la función.

También podemos escribir nuestras propias funciones en `R`. Por ejemplo, a menudo nos gusta "estandarizar" variables, es decir, restar la media muestral y dividir por la desviación estándar muestral.

$$\frac{x - \overline{x}}{s}$$

En `R`, escribiríamos una función para hacer esto. Al escribir una función, hay tres cosas que hacer:

- Darle un nombre a la función. Preferiblemente algo que sea corto pero descriptivo.
- Especificar los argumentos usando `function()`.

- Escribir el cuerpo de la función dentro de llaves, `{}`.

```
estandarizar <- function(x) {
  m <- mean(x)
  s <- sd(x)
  resultado <- (x - m) / s
  return(resultado)
}
```

Aquí, el nombre de la función es `estandarizar`, y la función tiene un solo argumento `x` que se utiliza en el cuerpo de la función. Ten en cuenta que la salida de la última línea del cuerpo es lo que se devuelve por la función. En este caso, la función devuelve el vector almacenado en la variable `resultado`.

```
datos_ejemplo <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
estandarizar(x = datos_ejemplo)
```

```
##  [1] -1.4863011 -1.1560120 -0.8257228 -0.4954337 -0.1651446
##  [6]  0.1651446  0.4954337  0.8257228  1.1560120  1.4863011
```

Esta función podría escribirse de forma mucho más sucinta, simplemente realizando todas las operaciones en una sola línea y devolviendo inmediatamente el resultado, sin almacenar ninguno de los resultados intermedios.

```
estandarizar <- function(x) {
  (x - mean(x)) / sd(x)
}
```

Al especificar argumentos, puedes proporcionar argumentos predeterminados.

```
potencia_de_numeros <- function(numero, potencia = 2) {
  numero^potencia
}
```

Veamos varias maneras en las que podríamos ejecutar esta función para realizar la operación 10^2.

```
potencia_de_numeros(10)
```

```
## [1] 100
```

```
potencia_de_numeros(10, 2)
```

```
## [1] 100
```

```r
potencia_de_numeros(numero = 10, potencia = 2)
```

```
## [1] 100
```

```r
potencia_de_numeros(potencia = 2, numero = 10)
```

```
## [1] 100
```

Note que, sin usar los nombres de los argumentos, el orden importa. El siguiente código no se evaluará al mismo resultado que el ejemplo anterior.

```r
potencia_de_numeros(2, 10)
```

```
## [1] 1024
```

Además, la siguiente línea de código producirá un error, ya que los argumentos sin un valor predeterminados deben ser especificados.

```r
potencia_de_numeros(potencia = 2)
```

```
## Error in potencia_de_numeros(potencia = 2): argument "numero" is missing
```

Para ilustrar aún más una función con un argumento predeterminado, escribiremos una función que calcule la varianza muestral de dos maneras.

Por defecto, calculará la estimación insesgada de σ^2, que llamaremos s^2.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

También tendrá la capacidad de devolver la estimación sesgada (basada en la máxima verosimilitud) que llamaremos $\hat{\sigma}^2$.

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

```r
calculo_varianza <- function(x, sesgo = FALSE) {
  n <- length(x) - 1 * !sesgo
  (1 / n) * sum((x - mean(x))^2)
}
```

```r
calculo_varianza(datos_ejemplo)
```

```
## [1] 9.166667
```

```r
calculo_varianza(datos_ejemplo, sesgo = FALSE)
```

```
## [1] 9.166667
```

```r
var(datos_ejemplo)
```

```
## [1] 9.166667
```

Vemos que la función está funcionando como se esperaba, y cuando devuelve la estimación insesgada, coincide con la función incorporada de `R`, `var()`. Finalmente, examinaremos la estimación sesgada de σ^2.

```r
calculo_varianza(datos_ejemplo, sesgo = TRUE)
```

```
## [1] 8.25
```