Curso básico de análisis de datos con R

Alfredo Sánchez Alberca

2022-01-07

Contents

Prefacio

Propósito de este manual

Este manual proporciona una introducción amigable al lenguaje de programación R para aquellas personas interesadas en utilizar este lenguaje para el análisis de datos.

El manual empieza con los conceptos básicos del lenguaje de programación R pero enseguida aborda su uso para la visualización y el análisis estadístico de datos, haciendo un recorrido por los test estadísticos más comunes.

Lo más interesante de este manual es la multitud de ejemplos que ilustran el uso de las técnicas estadísticas presentadas, así como los problemas propuestos.

El manual no aborda los fundamentos matemáticos de los análisis estadísticos presentados, aunque si explica brevemente cuándo deben usarse y cuándo no, así como las interpretaciones de los resultados obtenidos en los ejemplos. Si alguien está interesado en profundizar en los detalles matemáticos, puede visitar esta página.

No es un curso de programación en R, sino de uso de sus funciones predefinidas y de los paquetes más habituales para el análisis de datos.

Para cualquier comentario o sugerencia sobre este manual escriba al autor (as alber@ceu.es).

Licencia

Chapter 1

Introducción a R

La gran potencia de cómputo alcanzada por los ordenadores ha convertido a los mismos en poderosas herramientas al servicio de todas aquellas disciplinas que, como la Estadística, requieren manejar un gran volumen de datos. Actualmente, prácticamente nadie se plantea hacer un estudio estadístico serio sin la ayuda de un buen programa de análisis de datos.

R es un potente lenguaje de programación que incluye multitud de funciones para la representación y el análisis de datos. Fue desarrollado por Robert Gentleman y Ross Ihaka en la Universidad de Auckland en Nueva Zelanda, aunque actualmente es mantenido por una enorme comunidad científica en todo el mundo.

Figure 1.1: Logotipo de R

Las ventajas de R frente a otros programas habituales de análisis de datos, como pueden ser SPSS, SAS o Matlab, son múltiples:

- Es software libre y por tanto gratuito. Puede descargarse desde la web http://www.r-project.org/.
- Es multiplataforma. Existen versiones para Windows, Macintosh, Linux y otras plataformas.

- Está avalado y en constante desarrollo por una amplia comunidad científica distribuida por todo el mundo que lo utiliza como estándar para el análisis de datos.
- Cuenta con multitud de paquetes para todo tipo de análisis estadísticos y representaciones gráficas, desde los más habituales, hasta los más novedosos y sofisticados que no incluyen otros programas. Los paquetes están organizados y documentados en un repositorio CRAN (Comprehensive R Archive Network) desde donde pueden descargarse libremente.
- Es programable, lo que permite que el usuario pueda crear fácilmente sus propias funciones o paquetes para análisis de datos específicos. Existen multitud de libros, manuales y tutoriales libres que permiten su aprendizaje e ilustran el análisis estadístico de datos en distintas disciplinas científicas como las Matemáticas, la Física, la Biología, la Psicología, la Medicina, etc.

1.1 Entornos de desarrollo

Por defecto el entorno de trabajo de R es en línea de comandos, lo que significa que los cálculos y los análisis se realizan mediante comandos o instrucciones que el usuario teclea en una ventana de texto. No obstante, existen distintas interfaces gráficas de usuario que facilitan su uso, sobre todo para usuarios noveles. Algunas de ellas, como las que se enumeran a continuación, son completos entornos de desarrollo que facilitan la gestión de cualquier proyecto:

- RStudio. Probablemente el entorno de desarrollo más extendido para programar con R ya que incorpora multitud de utilidades para facilitar la programación con R.
- RKWard. Es otra otro de los entornos de desarrollo más completos que además incluye a posibilidad de añadir nuevos menús y cuadros de diálogo personalizados.
- Visual Studio Code. Es un entorno de desarrollo de propósito general ampliamente extendido. Aunque no es un entorno de desarrollo específico para R, incluye una extensión con utilidades que facilitan mucho el desarrollo con R.

Chapter 2

Tipos de datos simples

En R existen distintos tipos de datos predefinidos simples:

- `numeric`: Es el tipo de los números. Secuencia de dígitos (pueden incluir el - para negativos y el punto como separador de decimales) que representan números. Por ejemplo, `1`, `-2.0`, `3.1415` o `4.5e3`.
  Por defecto, cualquier número que se teclee tomará este tipo.
- `double`: Es el tipo de los números reales. Secuencia de dígitos que incluyen decimales separados por punto. Por ejemplo `3.1415` o `-2.0`. Son una subclase del tipo de datos numérico.
- `integer`: Es el tipo de los números enteros. Secuencia de dígitos sin separador de decimales que representan un número entero. Por ejemplo `1` o `-2`. Son una subclase del tipo de datos numérico.
- `character`: Es cualquier cadena de caracteres alfanuméricos. Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles. Por ejemplo `"Hola"` o `'Hola'`.
- `logical`: Es el tipo de los booleanos. Puede tomar cualquiera de los dos valores lógicos `TRUE` (verdadero) o `FALSE` (falso). También se pueden abreviar como `T` o `F`.
- `NA`: Se utiliza para representar datos desconocidos o perdidos. Aunque en realidad es un dato lógico, puede considerarse con un tipo de dato especial.
- `NULL`: Se utiliza para representar la ausencia de datos. La principal diferencia con `NA` es que `NULL` aparece cuando se intenta acceder a un dato que no existe, mientras que `NA` se utiliza para representar explícitamente datos perdidos en un estudio.

Para averiguar el tipo de un dato se puede utilizar la siguiente función:

- `class(x)`: Devuelve el tipo del dato `x`.

Ejemplo 2.1. A continuación se muestran los tipos de algunos datos.

```
class(3.1415)
#> [1] "numeric"
class(-1)
#> [1] "numeric"
class("Hola")
#> [1] "character"
class(TRUE)
#> [1] "logical"
class(NA)
#> [1] "logical"
class(NULL)
#> [1] "NULL"
```

También pueden utilizarse las siguientes funciones que devuelven un booleano:

- `is.numeric(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `numeric`.
- `is.double(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `double`.
- `is.integer(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `integer`.
- `is.character(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `character`.
- `is.logical(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `logical`.
- `is.na(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `NA`.
- `is.null(x)`: Devuelve el booleano `TRUE` si `x` es del tipo `NULL`.

2.1 Conversión de tipos

En muchas ocasiones es posible convertir un dato de un tipo a otro distinto. Para ello se usan las siguientes funciones:

- `as.numeric(x)`: Convierte el dato de `x` al tipo `numeric` siempre que sea posible o tenga sentido la conversión. Para convertir una cadena en un número, la cadena tiene que representar un número. El valor lógico `TRUE` se convierte en 1 y el `FALSE` en 0.
- `as.integer(x)`: Convierte el dato de `x` al tipo `integer` siempre que sea posible o tenga sentido la conversión. Para convertir una cadena en un número entero, la cadena tiene que representar un número entero. El valor lógico `TRUE` se convierte en 1 y el `FALSE` en 0.
- `as.character(x)`: Convierte el tipo de dato de `x` al tipo `character` simplemente añadiendo comillas.
- `as.logical(x)`: Convierte el tipo de dato de `x` al tipo lógico. Para datos numéricos, el 0 se convierte en `FALSE` y cualquier otro número en `TRUE`. Para cadenas se obtiene `NA` excepto para las cadenas `"TRUE"` y `"true"` que se convierten a `TRUE` y las cadenas `"FALSE"` y `"false"` que se convierten a `FALSE`.

El tipo `NA` no se puede convertir a ningún otro tipo pues representa la ausencia

del dato. Lo mismo ocurre con `NULL`.

2.2 Operaciones con números

2.2.1 Operadores aritméticos

Los siguientes operadores permiten realizar las clásicas operaciones aritméticas entre datos numéricos:

- `x + y`: Devuelve la suma de `x` e `y`.
- `x - y`: Devuelve la resta de `x` e `y`.
- `x * y`: Devuelve el producto de `x` e `y`.
- `x / y`: Devuelve el cociente de `x` e `y`.
- `x %% y`: Devuelve el resto de la división entera de `x` e `y`.
- `x ^ y`: Devuelve la potencia `x` elevado a `y`.

Ejemplo 2.2. A continuación se muestran varios ejemplos de operaciones aritméticas.

```
2 + 3
#> [1] 5
5 * -2
#> [1] -10
5 / 2
#> [1] 2.5
1 / 0
#> [1] Inf
5 %% 2
#> [1] 1
2 ^ 3
#> [1] 8
```

2.2.2 Operadores relacionales

Comparan dos números y devuelven un valor lógico.

- `x == y` : Devuelve `TRUE` si el número `x` es igual que el número `y`, y `FALSE` en caso contrario.
- `x > y` : Devuelve `TRUE` si el número `x` es mayor que el número `y`, y `FALSE` en caso contrario.
- `x < y` : Devuelve `TRUE` si el número `x` es menor que el número `y`, y `FALSE` en caso contrario.
- `x >= y` : Devuelve `TRUE` si el número `x` es mayor o igual que el número `y`, y `FALSE` en caso contrario.

- `x <= y` : Devuelve `TRUE` si el número `x` es menor o igual a que el número `y`, y `FALSE` en caso contrario.
- `x != y` : Devuelve `TRUE` si el número `x` es distinto del número `y`, y `FALSE` en caso contrario.

Ejemplo 2.3. A continuación se muestran varios ejemplos de operaciones relacionales.

```
3 == 3
#> [1] TRUE
3.1 <= 3
#> [1] FALSE
4 > 3
#> [1] TRUE
-1 != 1
#> [1] TRUE
5 %% 2
#> [1] 1
2 ^ 3
#> [1] 8
(2 + 3) ^ 2
#> [1] 25
```

2.3 Operaciones con cadenas

2.3.1 Funciones de cadenas

Existen muchas funciones para cadenas de texto pero las más comunes son:

- `nchar(c)`: Devuelve un número entero con el número de caracteres de la cadena.
- `paste(x, y, ..., sep=s)`: Concatena las cadenas `x`, `y`, etc. separándolas por la cadena `s`. Por defecto la cadena de separación es un espacio en blanco.
- `substr(c, start=i, stop=j)`: Devuelve la subcadena de la cadena `c` desde la posición `i` hasta la posición `j`. El primer carácter de una cadena ocupa la posición 1.
- `tolower(c)`: Devuelve la cadena que resulta de convertir la cadena `c` a minúsculas.
- `toupper(c)`: Devuelve la cadena que resulta de convertir la cadena `c` a mayúsculas.

Ejemplo 2.4. A continuación se muestran varios ejemplos de operaciones con cadenas de texto.

```
nchar("Me gusta R")
#> [1] 10
paste("Me", "gusta", "R")
```

```
#> [1] "Me gusta R"
paste("Me", "gusta", "R", sep = "-")
#> [1] "Me-gusta-R"
paste("Me", "gusta", "R", sep = "")
#> [1] "MegustaR"
substr("Me gusta R", 4, 8)
#> [1] "gusta"
tolower("Me gusta R")
#> [1] "me gusta r"
toupper("Me gusta R")
#> [1] "ME GUSTA R"
```

2.3.2 Operaciones de comparación de cadenas

- `x == y` : Devuelve `TRUE` si la cadena `x` es igual que la cadena `y`, y `FALSE` en caso contrario.
- `x > y` : Devuelve `TRUE` si la cadena `x` sucede a la cadena `y`, y `FALSE` en caso contrario.
- `x < y` : Devuelve `TRUE` si la cadena `x` antecede a la cadena `y`, y `FALSE` en caso contrario.
- `x >= y` : Devuelve `TRUE` si la cadena `x` sucede o es igual a la cadena `y`, y `FALSE` en caso contrario.
- `x <= y` : Devuelve `TRUE` si la cadena `x` antecede o es igual a la cadena `y`, y `FALSE` en caso contrario.
- `x != y` : Devuelve `TRUE` si la cadena `x` es distinta de la cadena `y`, y `FALSE` en caso contrario.

Utilizan el orden alfabético, las minúsculas van antes que las mayúsculas, y los números antes que las letras.

Ejemplo 2.5. A continuación se muestran varios ejemplos de operaciones de comparación de cadenas.

```
"R" == "R"
#> [1] TRUE
"R" == "r"
#> [1] FALSE
"uno" < "dos"
#> [1] FALSE
"A" > "a"
#> [1] TRUE
"" < "R"
#> [1] TRUE
```

2.4 Operaciones con datos lógicos o booleanos

2.4.1 Operadores lógicos

A la hora de comparar valores lógicos R asocia a `TRUE` el valor 1 y a `FALSE` el valor 0.

- `x == y`: Devuelve `TRUE` si los booleanos `x` y `y` son iguales, y `FALSE` en caso contrario.
- `x < y`: Devuelve `TRUE` si el booleano `x` es menor que el booleano `y`, y `FALSE` en caso contrario.
- `x <= y`: Devuelve `TRUE` si el booleano `x` es menor o igual que el booleano `y`, y `FALSE` en caso contrario.
- `x > y`: Devuelve `TRUE` si el booleano `x` es mayor que el booleano `y`, y `FALSE` en caso contrario.
- `x >= y`: Devuelve `TRUE` si el booleano `x` es mayor o igual que el booleano `y`, y `FALSE` en caso contrario.
- `x != y`: Devuelve `TRUE` si el booleano `x` es distinto que el booleano `y`, y `FALSE` en caso contrario.
- Negación `!b`: Devuelve `TRUE` si el booleano `b` es `FALSE`, y `FALSE` si es `TRUE`.
- Conjunción `x & y`: Devuelve `TRUE` si los booleanos `x`, y `y` son `TRUE` y `FALSE` en caso contrario.
- Disyunción `x | y`: Devuelve `TRUE` si alguno de los booleanos `x` o `y` son `TRUE`, y `FALSE` en caso contrario.

Tabla de verdad

x	y	!x	x & y	x | y
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	TRUE	FALSE	TRUE	TRUE

Ejemplo 2.6. A continuación se muestran varios ejemplos de operaciones lógicas con booleanos.

```
!TRUE
#> [1] FALSE
FALSE | TRUE
#> [1] TRUE
FALSE | FALSE
#> [1] FALSE
TRUE & FALSE
#> [1] FALSE
TRUE & TRUE
#> [1] TRUE
```

2.5 Variables

Una variable es un identificador ligado a algún valor.

Reglas para nombrarlas:

- Comienzan siempre por una letra o punto, seguida de otras letras, números, puntos o guiones bajos. Si empieza por punto no puede seguirle un número.
- No se pueden utilizarse palabras reservadas del lenguaje.

A diferencia de otros lenguajes de programación, las variables no tienen asociado un tipo y no es necesario declararlas antes de usarlas (tipado dinámico).

Para asignar un valor a una variable se utiliza el operador de asignación `<-`:

- `x <- y`: Asigna el valor `y` a la variable `x`.

Aunque es menos común también se puede utilizar el operador `=`.

Se puede crear una variable sin ningún valor asociado asignándole el valor `NULL`.

Una vez definida una variable, puede utilizarse en cualquier expresión donde tenga sentido el valor que tiene asociado.

Si una variable ya no va a usarse, es posible eliminarla y liberar el espacio que ocupan sus datos asociados con la siguiente función:

- `rm(x)`: Elimina la variable `x`.

Ejemplo 2.7. A continuación se muestran varios ejemplos de asignaciones de valores a variables.

```
x <- 3
x
#> [1] 3
y <- x + 2
y
#> [1] 5
# Valor no definido
x <- NULL
x
#> NULL
# Eliminar y
rm(y)
# A partir de aquí, una llamada a y produce un error.
```

2.5.1 Prioridad de los operadores

Al evaluar una expresiones R utiliza el siguiente orden de prioridad de evaluación:

1	Funciones predefinidas
2	Potencias (`^`)
3	Productos y cocientes (`*`, `/`, `%%`)
4	Sumas y restas (`+`, `-`)
5	Operadores relacionales (`==`, `>`, `<`, `>=`, `<=`, `!=`)
6	Negación (`!`)
7	Conjunción (`&`)
8	Disyunción (`|`)
9	Asignación (`<-`)

Se puede saltar el orden de evaluación utilizando paréntesis `( )`.

Ejemplo 2.8. A continuación se muestran varios ejemplos de evaluación de expresiones.

```
4 + 8 / 2 ^ 2
#> [1] 6
4 + (8 / 2) ^ 2
#> [1] 20
(4 + 8) / 2 ^ 2
#> [1] 3
(4 + 8 / 2) ^ 2
#> [1] 64
x <- 2
y <- 3
z <- ! x + 1 > y & y * 2  < x ^ 3
z
#> [1] TRUE
```

Ejercicio 2.1. Se dispone de los siguientes datos de una persona:

Variable	Valor
edad	20
estatura	165
peso	60
sexo	mujer

1. Declarar las variables anteriores y asignarles los valores correspondientes.
2. Definir la variable numérica imc con el índice de masa corporal aplicando la siguiente fórmula a las variables anteriores:

$$\text{imc} = \frac{\text{peso (kg)}}{\text{estatura (m)}^2}$$

3. Definir la variable booleana obesa con el valor correspondiente a la siguiente condición: ser mujer y no tener una edad superior a 60 y tener un índice de masa corporal mayor o igual que 30. ¿Es esta persona obesa?

Solución. Solución el ejercicio.

```
# Declaración de variables
edad <- 20
estatura <- 165
peso <- 60
sexo <- "mujer"
# Cálculo del índice de masa corporal
imc <- peso / (estatura / 100) ^ 2
imc
#> [1] 22.03857
# Cálculo de la obesidad
obesa <- sexo == "mujer" & ! edad > 60 & imc >= 30
obesa
#> [1] FALSE
```

2.6

Chapter 3

Tipos de datos estructurados

Los tipos estructurados de datos, a diferencia de los simples, son colecciones de datos con una determinada estructura. En R existen varios tipos tipos estructurados de datos que pueden clasificarse de acuerdo a su dimensión y a si son homogéneos (todos sus elementos son del mismo tipo) o heterogéneos.

Dimensiones	Homogéneos	Heterogéneos
1	Vector	Lista
2	Matriz	Data frame
n	Array	

Para averiguar la estructura de un dato estructurado se puede utilizar la función siguiente:

- `str(x)`: Devuelve una cadena de texto con la estructura de `x` en un formato amigable para los humanos.

3.1 Vectores

El vector es el tipo de dato estructurado más básicos en R. Un vector es una colección ordenada de elementos del mismo tipo.

3.1.1 Creación de vectores

Para construir un vector se utiliza la función de combinación `c()`:

- `c(x1, x2, ...)`: Devuelve el vector formado por los elementos `x1`, `x2`, etc.

También es posible utilizar el operador `:` para generar un vector de números enteros consecutivos:

- `x:y`: Devuelve el vector de números enteros consecutivos desde `x` hasta `y`.

Ejemplo 3.1. A continuación se muestran varios ejemplos de construcción de vectores.

```
c(1, 2, 3)
#> [1] 1 2 3
c("uno", "dos", "tres")
#> [1] "uno"  "dos"  "tres"
# Vector vacío
c()
#> NULL
# Vector con elementos perdidos
c(1, NA, 3)
#> [1]  1 NA  3
# Vector de números enteros consecutivos del 2 al 6
2:6
#> [1] 2 3 4 5 6
```

3.1.1.1 Vectores con nombres

Es posible asignar un nombre a cada elemento de un vector. Los nombres son etiquetas de texto que se asocian a cada elemento. Para asociar un nombre a un elemento se utiliza la sintaxis `nombre = valor`, donde `nombre` es una cadena de caracteres y `valor` es el elemento del vector.

Ejemplo 3.2. A continuación se muestra un ejemplo de creación de un vector con nombres.

```
c("Matemáticas" = 8.2, "Física" = 6.5, "Economía" = 4.5)
#> Matemáticas      Física    Economía
#>         8.2         6.5         4.5
```

Para acceder a los nombres de un vector se utiliza la siguiente función:

- `names(x)`: Devuelve un vector de cadenas de caracteres con los nombres de los elementos del vector `x`.

Ejemplo 3.3. A continuación se muestra un ejemplo de acceso a los nombres de un vector.

```
notas <- c("Matemáticas" = 8.2, "Física" = 6.5, "Economía" = 4.5)
names(notas)
#> [1] "Matemáticas" "Física"      "Economía"
```

3.1.2 Tamaño de un vector

El número de elementos de un vector es su *tamaño* y puede averiguarse con la siguiente función.

- `lenght(x)`: Devuelve el número de elementos del vector `x`.

Ejemplo 3.4. A continuación se muestran varios ejemplos de la obtención del tamaño de un vector.

```
length(c(1, 2, 3))
#> [1] 3
length(c())
#> [1] 0
```

3.1.3 Coerción de elementos

Puesto que los elementos de un vector tienen que ser del mismo tipo, cuando se crea un vector con datos de distintos tipos, la función `c()` los convertirá al mismo tipo, lo que se conoce como *coerción* de tipos. La coerción se produce de los tipos menos flexibles a los más flexibles: `logical` < `integer` < `double` < `character`.

Ejemplo 3.5. A continuación se muestran varios ejemplos de coerciones.

```
c(1, 2.5)
#> [1] 1.0 2.5
c(FALSE, TRUE, 2)
#> [1] 0 1 2
c(FALSE, TRUE, 2, "tres")
#> [1] "FALSE" "TRUE"  "2"     "tres"
```

3.1.4 Acceso a los elementos de un vector

Para acceder a los elementos de un vector se utiliza un índice. Como veremos a continuación, este índice puede ser entero, lógico o de cadena de caracteres y se indica siempre entre corchetes `[ ]` a continuación del vector.

3.1.4.1 Acceso mediante un índice entero

Los elementos de un vector están ordenados y el acceso más simple a ellos es mediante su número de orden, es decir, indicando entre corchetes el entero que corresponde a su número de orden. Se puede acceder simultáneamente a varios elementos mediante un vector con sus números de orden. Por otro lado, también es posible usar enteros negativos y en tal caso se obtendrán todos los elementos del vector excepto los que ocupan las posiciones correspondientes al valor absoluto de los índices negativos. Esta es la forma más habitual de eliminar elementos de un vector.

En R los índices enteros para acceder a los elementos de un vector comienzan en 1, a diferencia de otros lenguajes de programación que empiezan en 0.

Ejemplo 3.6. A continuación se muestran varios ejemplos de acceso a los elementos de un vector mediante índices enteros.

```
x <- c(2,4,6,8,10)
# Acceso al elemento que está en la tercera posición
x[3]
#> [1] 6
# Acceso a los elementos de las posiciones 2 y 4
x[c(2, 4)]
#> [1] 4 8
# Acceso a todos los elementos excepto el primero y el quinto
x[c(-1, -5)]
#> [1] 4 6 8
```

3.1.4.2 Acceso mediante un índice lógico

Cuando se utiliza un índice lógico, se obtienen los elementos correspondientes a las posiciones donde está el valor booleano `TRUE`.

Ejemplo 3.7. A continuación se muestran varios ejemplos de acceso a los elementos de un vector mediante índices lógicos.

```
x <- c(2,4,6,8,10)
# Acceso al elemento que está en la tercera posición
x[c(F,F,T,F,F)]
#> [1] 6
# Acceso a los elementos de las posiciones 2 y 4
x[c(F,T,F,T,F)]
#> [1] 4 8
```

Esta forma de acceder es útil cuando se genera el vector de índices mediante operadores relacionales. Cuando se aplica un operador relacional a un vector se obtiene otro vector lógico que resulta de aplicar el operador relacional a cada uno de los elementos del vector. De esta manera se puede realizar filtros para obtener los elementos de un vector que cumplen una determinada condición.

Ejemplo 3.8. A continuación se muestran varios ejemplos de acceso a los elementos de un vector mediante condiciones.

```
x <- 1:6
x %% 2 == 0
#> [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE
# Filtrado de los valores pares
x[x %% 2 == 0]
#> [1] 2 4 6
# Filtrado de los valores pares menores que 5
```

```
x[x %% 2 == 0 & x < 5]
#> [1] 2 4
```

3.1.4.3 Acceso mediante un índice de cadena

Si los elementos de un vector tienen nombre, es posible acceder a ellos usando sus nombres como índices.

Ejemplo 3.9. A continuación se muestran varios ejemplos de acceso a los elementos de un vector mediante índices de cadena.

```
notas <- c("Matemáticas" = 8.2, "Física" = 6.5, "Economía" = 4.5)
notas["Física"]
#> Física
#>    6.5
notas[c("Matemáticas", "Economía")]
#> Matemáticas    Economía
#>         8.2         4.5
```

3.1.5 Pertenencia a un vector

Para comprobar si un valor en particular es un elemento de un vector se puede utilizar el operador `%in%`:

- `x %in% y`: Devuelve el booleano `TRUE` si `x` es un elemento del vector `y`, y `FALSE` en caso contrario.

Ejemplo 3.10. A continuación se muestran varios ejemplos de pertenencia de elementos a un vector.

```
x <- 1:3
2 %in% x
#> [1] TRUE
4 %in% x
#> [1] FALSE
```

3.1.6 Modificación de los elementos de un vector

Para modificar uno o varios elementos de un vector basta con acceder a esos elementos y usar el operador de asignación para asignar nuevos valores.

Ejemplo 3.11. A continuación se muestran varios ejemplos de modificación de los elementos de un vector.

```
x <- c(1, 2, 3)
x[2] <- 0
x
#> [1] 1 0 3
```

```
x[c(1, 3)] <- 1
x
#> [1] 1 0 1
```

3.1.7 Añadir elementos a un vector

Para añadir nuevos elementos a un vector pueden usarse las siguientes funciones:

- `c(x, y)`: Devuelve el vector que resulta de añadir al vector `x` los elementos del vector `y`.
- `append(x, y, pos)`: Devuelve el vector que resulta de añadir al vector `x` los elementos del vector `y`, a continuación de la posición `pos`. El parámetro `pos` es opcional y si no se indica, los elementos de `y` se añaden al final de los de `x`.

Ejemplo 3.12. A continuación se muestran varios ejemplos de añadir nuevos elementos a un vector.

```
x <- c(1, 2, 3)
y <- c(x, c(4, 5))
y
#> [1] 1 2 3 4 5
y <- append(x, c(4, 5), 2)
y
#> [1] 1 2 4 5 3
```

3.1.8 Eliminación de un vector

Para eliminar los elementos de un vector basta con asignar `NULL` a la variable que lo contiene, pero si se quiere liberar la memoria que ocupa la variable se utiliza la función `rm()`.

3.1.9 Operaciones aritméticas con vectores

3.1.9.1 Operaciones aritméticas elemento a elemento

Para vectores numéricos las operaciones aritméticas habituales se aplican elemento a elemento. Si los vectores tienen distinto tamaño, el tamaño del vector más pequeño se equipara al tamaño del mayor, reutilizando sus elementos, empezando por el primero.

Ejemplo 3.13. A continuación se muestran varios ejemplos de operaciones aritméticas con vectores numéricos.

```
x <- c(1, 2, 3)
y <- c(0, 1, -1)
x + y
#> [1] 1 3 2
```

```
x * y
#> [1]  0  2 -3
x / y
#> [1] Inf   2  -3
x ^ y
#> [1] 1.0000000 2.0000000 0.3333333
```

3.1.9.2 Producto escalar de vectores

Para calcular el producto escalar de dos vectores numéricos se utiliza el operador `%*%`. Si los vectores tienen distinto tamaño se produce un error.

Ejemplo 3.14. A continuación se muestra un ejemplo del producto escalar de dos vectores.

```
x <- c(1, 2, 3)
y <- c(0, 1, -1)
x %*% y
#>      [,1]
#> [1,]   -1
```

3.2 Factores

3.2.1 Operaciones con factores

Un factor es una estructura de datos especial que se utiliza para representar categorías de variables cualitativas y por tanto puede tomar un conjunto finito de valores predefinidos conocido como *niveles* del factor.

Para definir un factor se utiliza la siguiente función:

- `factor(x, levels = niveles)`: Crea un dato de tipo factor con los elementos del vector `x`. Los niveles del factor pueden indicarse mediante el parámetro `levels`, pasándole un vector con los valores posibles. Si no se indica el parámetro `levels` los niveles del factor se obtienen automáticamente a partir de los elementos del vector `x` (tantos niveles con valores distintos tenga).

Los factores son en realidad vectores de números enteros a los que se le añade un atributo especial para indicar los niveles del factor.

Ejemplo 3.15. A continuación se muestran varios ejemplos de creación de factores.

```
sexo <- factor(c("mujer", "hombre", "mujer"))
sexo
#> [1] mujer  hombre mujer
#> Levels: hombre mujer
```

```
class(sexo)
#> [1] "factor"
str(sexo)
#>  Factor w/ 2 levels "hombre","mujer": 2 1 2
grupo.sanguineo <- factor(c("B", "A", "A"), levels = c("A", "B", "AB", "0"), )
grupo.sanguineo
#> [1] B A A
#> Levels: A B AB 0
```

Es posible establecer un orden entre los niveles de un factor añadiendo el parámetro `ordered = TRUE` a la función anterior. Esto es útil para representar categorías ordinales entre las que existe un orden natural.

Ejemplo 3.16. A continuación se muestra un ejemplo de creación de un factor ordenado.

```
nivel.estudio <- factor(c("Secundarios", "Graduado", "Bachiller"), levels = c("Sin estu
nivel.estudio
#> [1] Secundarios Graduado    Bachiller
#> 5 Levels: Sin estudios < Primarios < ... < Graduado
```

Para comprobar si una estructura es del tipo factor se utiliza la siguiente función:

- is.factor(`x`): Devuelve el booleano `TRUE` si `x` es del tipo factor, y `FALSE` en caso contrario.

3.2.2 Acceso a los elementos de un factor

Se puede acceder a los elementos de un factor de la misma manera que se accede a los elementos de un vector. Y para obtener sus niveles se utiliza la siguiente función:

- `levels(x)`: Devuelve un vector con los niveles del factor `x`.

Ejemplo 3.17. A continuación se muestran varios ejemplos de acceso a los elementos y los niveles de un factor.

```
sexo <- factor(c("mujer", "hombre", "mujer"))
sexo[2]
#> [1] hombre
#> Levels: hombre mujer
sexo[c(1, 2)]
#> [1] mujer  hombre
#> Levels: hombre mujer
sexo[-2]
#> [1] mujer mujer
#> Levels: hombre mujer
levels(sexo)
#> [1] "hombre" "mujer"
```

3.2.3 Modificación de los elementos de un factor

Se puede modificar los elementos de un factor de manera similiar a como se modifican los elementos de un vector, es decir accediendo al elemento que se quiere modificar y reasignándole un nuevo valor. La única diferencia con los vectores es que si el nuevo valor que se quiere asignar no está entre los niveles del factor, se obtiene el valor `NA`.

Ejemplo 3.18. A continuación se muestran varios de modificación de los elementos de un factor.

```
grupo.sanguineo <- factor(c("B", "A", "A"), levels = c("A", "B", "AB", "0"))
grupo.sanguineo
#> [1] B A A
#> Levels: A B AB 0
grupo.sanguineo[2] <- "AB"
grupo.sanguineo
#> [1] B  AB A
#> Levels: A B AB 0
grupo.sanguineo[1] <- "C"
#> Warning in `[<-.factor`(`*tmp*`, 1, value = "C"): invalid
#> factor level, NA generated
grupo.sanguineo
#> [1] <NA> AB   A
#> Levels: A B AB 0
```

3.3 Listas

Las listas son colecciones ordenadas de elementos de que pueden ser de distintos tipos. Los elementos de una lista también pueden ser de tipos estructurados (vectores o listas), lo que las convierte en el tipo de dato más versátil de R. Como veremos más adelante, otras estructuras de datos como los *data frames* o los propios modelos estadísticos se construyen usando listas.

3.3.1 Creación de listas

Para construir una lista se utiliza la función `list()`:

- `list(x1, x2, ...)`: Devuelve la lista con los elementos `x1`, `x2`, etc.

Ejemplo 3.19. A continuación se muestran varios ejemplos de creación de listas.

```
list(1, "dos", TRUE)
#> [[1]]
#> [1] 1
#>
#> [[2]]
```

```
#> [1] "dos"
#>
#> [[3]]
#> [1] TRUE
# Lista con vectores y listas
x <- list(1, c("dos", "tres"), list(4, "cinco"))
x
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "dos"  "tres"
#>
#> [[3]]
#> [[3]][[1]]
#> [1] 4
#>
#> [[3]][[2]]
#> [1] "cinco"
str(x)
#> List of 3
#>  $ : num 1
#>  $ : chr [1:2] "dos" "tres"
#>  $ :List of 2
#>   ..$ : num 4
#>   ..$ : chr "cinco"
# Lista vacía
list()
#> list()
```

3.3.1.1 Listas con nombres

Ejemplo 3.20. A continuación se muestra un ejemplo de creación de una lista con nombres.

```
list("nombre" = "María", "edad" = 21, "dirección" = list("calle" = "Delicias", "número"
#> $nombre
#> [1] "María"
#>
#> $edad
#> [1] 21
#>
#> $dirección
#> $dirección$calle
#> [1] "Delicias"
#>
```

```
#> $dirección$número
#> [1] 24
#>
#> $dirección$municipio
#> [1] "Madrid"
```

Para obtener los nombres de una lista se utiliza la siguiente función:

- `names(x)`: Devuelve un vector de cadenas de caracteres con los nombres de los elementos de la lista `x`.

Ejemplo 3.21. A continuación se muestra un ejemplo de acceso a los nombres de una lista.

```
persona <- list("nombre" = "María", "edad" = 21, "dirección" = list("calle" = "Delicias", "número
names(persona)
#> [1] "nombre"    "edad"      "dirección"
```

3.3.2 Tamaño de una lista

El número de elementos de una lista es su *tamaño* y puede averiguarse con la siguiente función:

- `lenght(x)`: Devuelve el número de elementos de la lista `x`.

Ejemplo 3.22. A continuación se muestran varios ejemplos la obtención del tamaño de una lista.

```
length(list(1, "dos", TRUE))
#> [1] 3
length(list(1, c("dos", "tres"), list(4, "cinco")))
#> [1] 3
length(list())
#> [1] 0
```

3.3.3 Acceso a los elementos de una lista

Se accede a los elementos de una lista de forma similar a los vectores, mediante índices enteros, lógicos o de cadena, entre corchetes `[ ]`.

3.3.3.1 Acceso mediante un índice entero

Al igual que los vectores, los elementos de una lista están ordenados y se puede utilizar un índice entero para acceder a los elementos que ocupan una determinada posición.

Ejemplo 3.23. A continuación se muestran varios ejemplos de acceso a los elementos de una lista mediante índices enteros.

```
x <- list(1, "dos", TRUE, 4.5)
# Acceso al elemento que está en la segunda posición
x[2]
#> [[1]]
#> [1] "dos"
# Acceso a los elementos de las posiciones 1 y 3
x[c(1, 3)]
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] TRUE
# Acceso a todos los elementos excepto el primero y el cuarto
x[c(-1, -4)]
#> [[1]]
#> [1] "dos"
#>
#> [[2]]
#> [1] TRUE
```

3.3.3.2 Acceso mediante un índice lógico

Cuando se utiliza un índice lógico, se obtienen los elementos correspondientes a las posiciones donde está el valor booleano `TRUE`.

Ejemplo 3.24. A continuación se muestran varios ejemplos de acceso a los elementos de una lista mediante índices lógicos.

```
x <- list(1, "dos", TRUE, 4.5)
x[c(T,F,F,T)]
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 4.5
x < 2
#> Warning: NAs introducidos por coerción
#> [1]  TRUE    NA  TRUE FALSE
# Filtrado de valores menores que 2
x[x < 2]
#> Warning: NAs introducidos por coerción
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> NULL
```

```
#>
#> [[3]]
#> [1] TRUE
```

Obsérvese que para los elementos que no tiene sentido la comparación se obtiene `NA`, y que el acceso mediante este índice devuelve `NULL`.

3.3.3.3 Acceso mediante un índice de cadena

Si los elementos de una lista tienen nombre, se puede acceder a ellos utilizando sus nombres como índices. La única diferencia con el acceso mediante cadenas de vectores es que se obtiene siempre una lista, incluso cuando sólo se quiere acceder a un elemento. Para obtener un elemento, y no una lista con ese único elemento, se utilizan dobles corchetes `[[ ]]`.

Ejemplo 3.25. A continuación se muestran varios ejemplos de acceso a los elementos de una lista mediante índices de cadena.

```
persona <- list("nombre" = "María", "edad" = 21, "dirección" = list("calle" = "Delicias", "número
persona[c("edad", "nombre")]
#> $edad
#> [1] 21
#>
#> $nombre
#> [1] "María"
persona["nombre"]
#> $nombre
#> [1] "María"
typeof(persona["nombre"])
#> [1] "list"
# Acceso a un único elemento
persona[["nombre"]]
#> [1] "María"
# Acceso a una lista anidada
persona[["dirección"]][["municipio"]]
#> [1] "Madrid"
```

Una alternativa a los dobles corchetes es el operador de acceso a listas `$`. Este operador además permite utilizar coincidencias parciales en los nombres de los elementos para acceder a ellos.

Ejemplo 3.26. A continuación se muestran varios ejemplos de acceso a los elementos de una lista mediante el operador `$`.

```
persona <- list("nombre" = "María", "edad" = 21, "dirección" = list("calle" = "Delicias", "número
# Acceso a un único elemento
persona$nombre
#> [1] "María"
```

```
# Acceso mediante coincidencia parcial
persona$nom
#> [1] "María"
# Acceso a una lista anidada
persona$dirección$municipio
#> [1] "Madrid"
```

3.3.4 Modificación de los elementos de una lista

Para modificar uno o varios elementos de una lista basta con acceder a esos elementos y reasignarles valores con el operador de asignación.

Ejemplo 3.27. A continuación se muestran varios ejemplos de modificación de los elementos de una lista.

```
persona <- list("nombre" = "María", "edad" = 21)
persona$edad <- 22
persona
#> $nombre
#> [1] "María"
#>
#> $edad
#> [1] 22
```

3.3.5 Añadir elementos a una lista

La forma más sencilla de añadir un elemento con nombre a una lista es indicando el nombre con el operador `$` y asignándole un valor con el operador de asignación `<-`:

- `x$nombre <- y`: Añade el elemento `y` a la lista `x` con el nombre `nombre`.

El nuevo elemento se añade siempre al final de la lista.

Para añadir elementos sin nombre o en una posición determinada se puede utilizar la función `append()`:

- `append(x, y, pos)`: Devuelve la lista vector que resulta de añadir a `x` los elementos de la lista `y`, a continuación de la posición `pos`. El parámetro `pos` es opcional y si no se indica, los elementos de `y` se añaden al final de los de `x`.

Ejemplo 3.28. A continuación se muestran varios ejemplos de añadir nuevos elementos a una lista.

```
persona <- list("nombre" = "María", "edad" = 21)
persona$email <- "maria@ceu.es"
persona
#> $nombre
```

```
#> [1] "María"
#>
#> $edad
#> [1] 21
#>
#> $email
#> [1] "maria@ceu.es"
append(persona, list("sexo" = "Mujer"), 2)
#> $nombre
#> [1] "María"
#>
#> $edad
#> [1] 21
#>
#> $sexo
#> [1] "Mujer"
#>
#> $email
#> [1] "maria@ceu.es"
```

3.3.6 Conversión de una lista en un vector

Es posible convertir una lista en un vector con la siguiente función:

- `unlist(x)`: Devuelve el vector que resulta de aplanar recursivamente la lista `x` y convertir todos los elementos al mismo tipo mediante coerción de tipos.

Ejemplo 3.29. A continuación se muestran varios ejemplos de conversión de una lista en un vector.

```
persona <- list("nombre" = "María", "edad" = 21, "dirección" = list("calle" = "Delicias", "número
unlist(persona)
#>              nombre                edad     dirección.calle
#>             "María"                "21"          "Delicias"
#>    dirección.número dirección.municipio
#>                "24"            "Madrid"
typeof(unlist(persona))
#> [1] "character"
```

3.4 Matrices

Una matriz es una estructura de datos bidimensional de elementos del mismo tipo organizados en filas y columnas. Una matriz es similar a un vector pero contiene una atributo adicional con sus dimensiones (número de filas y número de columnas).

3.4.1 Creación de matrices

Para crear una matriz se utiliza la siguiente función:

- `matrix(x, nrow = m, ncol = n)`: Devuelve la matriz con los elementos del vector `x` organizados en `n` filas y `m` columnas. Habitualmente basta con especificar el número de filas o el número de columnas.

Ejemplo 3.30. A continuación se muestran varios ejemplos de creación de matrices.

```
matrix(1:6, nrow = 2, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
matrix(1:6, nrow = 2)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
matrix(1:6, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
# La matriz de 1 x 1
matrix()
#>      [,1]
#> [1,]   NA
```

Como se puede observar en el ejemplo anterior, los elementos se disponen por columnas, pero se pueden disponer los elementos por filas pasando el parámetro `byrow = TRUE` a la función `matrix`.

Ejemplo 3.31. A continuación se muestran varios ejemplos de creación de matrices.

```
matrix(1:6, nrow = 2)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
matrix(1:6, nrow = 2, byrow = TRUE)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
```

3.4.1.1 Matrices con nombres de filas y columnas

Es posible poner nombres a las filas y a las columnas de una matriz añadiendo el parámetro `dimnames` y pasándole una lista de dos vectores de cadenas con los nombres de las filas y las columnas respectivamente.

Ejemplo 3.32. A continuación se muestran varios ejemplos de creación de matrices con nombres de filas y columnas.

```
matrix(1:6, nrow = 2, ncol = 3, dimnames = list(c("fila1", "fila2"), c("columna1", "columna2", "c
#>       columna1 columna2 columna3
#> fila1        1        3        5
#> fila2        2        4        6
```

Para obtener los nombres de las filas y las columnas de una matriz se utilizan las siguientes funciones:

- `rownames(x)`: Devuelve un vector de cadenas de caracteres con los nombres de las filas de la matriz `x`.
- `colnames(x)`: Devuelve un vector de cadenas de caracteres con los nombres de las columnas de la matriz `x`.

Ejemplo 3.33. A continuación se muestran varios ejemplos de creación de matrices con nombres de filas y columnas.

```
x <- matrix(1:6, nrow = 2, ncol = 3, dimnames = list(c("fila1", "fila2"), c("columna1", "columna2
rownames(x)
#> [1] "fila1" "fila2"
colnames(x)
#> [1] "columna1" "columna2" "columna3"
```

3.4.2 Tamaño y dimensiones de una matriz

Para obtener el número de elementos y las dimensiones de una matriz se pueden utilizar las siguientes funciones:

- `length(x)`: Devuelve un entero con el número de elementos de la matriz `x`.
- `nrow(x)`: Devuelve un entero con el número de filas de la matriz `x`.
- `ncol(x)`: Devuelve un entero con el número de columnas de la matriz `x`.
- `dim(x)`: Devuelve un vector de dos enteros con el número de filas y el número de columnas de la matriz `x`.

Ejemplo 3.34. A continuación se muestran varios ejemplos de acceso a las dimensiones de una matriz.

```
x <- matrix(1:6, nrow = 2)
length(x)
#> [1] 6
nrow(x)
#> [1] 2
ncol(x)
#> [1] 3
dim(x)
#> [1] 2 3
```

Usando esta última función se pueden modificar las dimensiones de una matriz asignando un vector de dos enteros con las nuevas dimensiones. Esto también permite crear una matriz a partir de un vector.

Ejemplo 3.35. A continuación se muestran varios ejemplos de modificación de las dimensiones de una matriz.

```
x <- 1:6
dim(x) <- c(2, 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
dim(x) <- c(3, 2)
x
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
```

3.4.3 Acceso a los elementos de una matriz

Para acceder a los elementos de una matriz se utilizan dos índices (uno para las filas y otro para las columnas), separados por comas y entre corchetes `[]` a continuación de la matriz. Al igual que para los vectores, los índices pueden ser enteros, lógicos o de cadenas de caracteres.

3.4.3.1 Acceso mediante índices enteros

Para acceder a los elementos de una matriz mediante índices enteros se indica el número de fila y el número de columna del elemento entre corchetes:

- `x[i,j]`: Devuelve el elemento de la matriz `x` que está en la fila `i` y la columna `j`.

Se puede acceder a más de un elemento indicando un vector de enteros para las filas y otro para las columnas. De esta manera se obtiene una submatriz. Si no se indica la fila o la columna se obtienen todos los elementos de todas las filas o columnas. Al igual que para vectores, se pueden utilizar enteros negativos para descartar filas o columnas

Ejemplo 3.36. A continuación se muestran varios ejemplos de acceso a los elementos de una matriz.

```
x <- matrix(1:9, nrow = 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
```

```
#> [3,]    3    6    9
# Acceso al elemento de la segunda fila y tercera columna
x[2,3]
#> [1] 8
# Acceso a la submatriz de la primera y tercera filas, y tercera y segunda columnas
x[c(1, 3), c(3, 2)]
#>      [,1] [,2]
#> [1,]    7    4
#> [2,]    9    6
# Acceso a la primera fila
x[1, ]
#> [1] 1 4 7
# Acceso a la segunda columna
x[, 2]
#> [1] 4 5 6
# Acceso a la submatriz con todos los elementos salvo la tercera fila y la segunda columna
x[-3, -2]
#>      [,1] [,2]
#> [1,]    1    7
#> [2,]    2    8
```

3.4.3.2 Acceso mediante índices lógicos

Cuando se utilizan índices lógicos, se obtienen los elementos correspondientes a las filas y columnas donde está el valor booleano `TRUE`.

Ejemplo 3.37. A continuación se muestran varios ejemplos de acceso a los elementos de una matriz.

```
x <- matrix(1:9, nrow = 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
# Acceso al elemento de la segunda fila y tercera columna
x[c(F, T, F), c(F, F, T)]
#> [1] 8
# Acceso a la submatriz de la primera y tercera filas, y segunda y tercera columnas
x[c(T, F, T), c(F, T, T)]
#>      [,1] [,2]
#> [1,]    4    7
#> [2,]    6    9
# Acceso a la primera fila
x[c(T, F, F), ]
#> [1] 1 4 7
```

```
# Acceso a la segunda columna
x[, c(F, T, F)]
#> [1] 4 5 6
```

3.4.3.3 Acceso mediante índices de cadena

Si las filas y las columnas de una matriz tienen nombre, es posible acceder a sus elementos usando los nombres de las filas y columnas como índices.

```
x <- matrix(1:9, nrow = 3, dimnames = list(c("f1", "f2", "f3"), c("c1", "c2", "c3")))
x
#>    c1 c2 c3
#> f1  1  4  7
#> f2  2  5  8
#> f3  3  6  9
# Acceso al elemento de la segunda fila y tercera columna
x["f2", "c3"]
#> [1] 8
# Acceso a la submatriz de la primera y tercera filas, y tercera y segunda columnas
x[c("f1", "f3"), c("c3", "c2")]
#>    c3 c2
#> f1  7  4
#> f3  9  6
```

Finalmente, es posible combinar distintos tipos de índices (enteros, lógicos o de cadena) para indicar las filas y las columnas a las que acceder.

3.4.4 Pertenencia a una matriz

Para comprobar si un valor en particular es un elemento de una matriz se puede utilizar el operador `%in%`:

- `x %in% y`: Devuelve el booleano `TRUE` si `x` es un elemento de la matriz `y`, y `FALSE` en caso contrario.

Ejemplo 3.38. A continuación se muestran varios ejemplos de pertenencia de elementos a una matriz.

```
x <- matrix(1:9, nrow = 3)
2 %in% x
#> [1] TRUE
-1 %in% x
#> [1] FALSE
```

3.4.5 Modificación de los elementos de una matriz

Para modificar uno o varios elementos de una matriz basta con acceder a esos elementos y usar el operador de asignación para asignar nuevos valores.

Ejemplo 3.39. A continuación se muestran varios ejemplos de modificación de los elementos de un vector.

```
x <- matrix(1:9, nrow = 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
x[2,3] <- 0
x
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    0
#> [3,]    3    6    9
x[c(1, 3), 1:2] <- -1
x
#>      [,1] [,2] [,3]
#> [1,]   -1   -1    7
#> [2,]    2    5    0
#> [3,]   -1   -1    9
```

3.4.6 Añadir elementos a una matriz

Para añadir nuevas filas o columnas a una matriz se utilizan las siguientes funciones:

- `rbind(x, y)`: Devuelve la matriz que resulta de añadir nuevas filas a la matriz `x` con los elementos del vector `y`.
- `rbind(x, y)`: Devuelve la matriz que resulta de añadir nuevas columnas a la matriz `x` con los elementos del vector `y`.

Ejemplo 3.40. A continuación se muestran varios ejemplos de añadir nuevas filas y columnas a una matriz.

```
x <- matrix(1:6, nrow = 2)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
# Añadir una nueva fila
rbind(x, c(7, 8, 9))
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#> [3,]    7    8    9
# Añadir una nueva columna
```

```
cbind(x, c(7, 8))
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    3    5    7
#> [2,]    2    4    6    8
```

Obsérvese que si el número de elementos proporcionados en el vector es menor del necesario para completar la fila o columna, se reutilizan los elementos del vector empezando desde el principio.

3.4.7 Trasponer una matriz

Para trasponer una matriz se utiliza la función siguiente:

- `t(x)`: Devuelve la matriz traspuesta de la matriz `x`.

Ejemplo 3.41. A continuación se muestran un ejemplo de la trasposición de una matriz.

```
x <- matrix(1:6, nrow=2)
t(x)
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
#> [3,]    5    6
```

3.4.8 Operaciones aritméticas con matrices

3.4.8.1 Operaciones aritméticas elemento a elemento

Para matrices numéricas las operaciones aritméticas habituales se aplican elemento a elemento. Si las dimensiones de las matrices son distintas se produce un error.

Ejemplo 3.42. A continuación se muestran varios ejemplos de operaciones aritméticas elemento a elemento con matrices numéricas.

```
x <- matrix(1:6, nrow = 2)
y <- matrix(c(0, 1, 0, -1, 0, 1), nrow = 2)
x + y
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    3    3    7
x * y
#>      [,1] [,2] [,3]
#> [1,]    0    0    0
#> [2,]    2   -4    6
x / y
#>      [,1] [,2] [,3]
```

```
#> [1,]  Inf  Inf  Inf
#> [2,]    2   -4    6
x ^ y
#>      [,1] [,2] [,3]
#> [1,]    1 1.00    1
#> [2,]    2 0.25    6
```

3.4.8.2 Multiplicación de matrices

Para multiplicar dos matrices numéricas se utiliza el operador `%*%`. Si el número de columnas de la primera matriz no es igual que el número de filas de la segunda se produce un error.

Ejemplo 3.43. A continuación se muestran varios ejemplos del producto de dos matrices numéricas.

```
x <- matrix(1:6, ncol = 3)
y <- matrix(1:6, nrow = 3)
x %*% y
#>      [,1] [,2]
#> [1,]   22   49
#> [2,]   28   64
y %*% x
#>      [,1] [,2] [,3]
#> [1,]    9   19   29
#> [2,]   12   26   40
#> [3,]   15   33   51
```

3.4.9 Determinante de una matriz

Para calcular el determinante de una matriz numérica cuadrada se utiliza la siguiente función:

- `det(x)`: Devuelve el determinante de la matriz `x`. Si `x` no es una matriz numérica cuadrada produce un error.

Ejemplo 3.44. A continuación se muestra un ejemplo del cálculo del determinante de una matriz numérica cuadrada.

```
x <- matrix(1:4, ncol = 2)
det(x)
#> [1] -2
```

3.4.10 Inversa de una matriz

Para calcular la matriz inversa de una matriz numérica cuadrada se utiliza la siguiente función:

- `solve(x)`: Devuelve la matriz inversa de la matriz `x`. Si `x` no es una matriz numérica cuadrada produce un error. Si la matriz no es invertible por tener determinante nulo también se obtiene un error.

Ejemplo 3.45. A continuación se muestra un ejemplo del cálculo del determinante de una matriz numérica cuadrada.

```
x <- matrix(1:4, nrow = 2)
solve(x)
#>      [,1] [,2]
#> [1,]   -2  1.5
#> [2,]    1 -0.5
# El producto de una matriz por su inversa es la matriz identidad.
x %*% solve(x)
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0    1
```

3.4.11 Autovalores y autovectores de una matriz

Para calcular los autovalores y los autovectores de una matriz numérica cuadrada se utiliza la siguiente función:

- `eigen(x)`: Devuelve una lista con los autovalores y los autovectores de la matriz `x`. Para acceder a los autovalores se utiliza el nombre `values` y para acceder a los autovectores se utiliza el nombre `vectors`.

Ejemplo 3.46. A continuación se muestra un ejemplo del cálculo los autovalores y los autovectores de una matriz numérica cuadrada. Si `x` no es una matriz numérica cuadrada produce un error.

```
x <- matrix(1:4, nrow = 2)
# Autovalores
eigen(x)$values
#> [1]  5.3722813 -0.3722813
# Autovectores
eigen(x)$vectors
#>            [,1]       [,2]
#> [1,] -0.5657675 -0.9093767
#> [2,] -0.8245648  0.4159736
```

3.5 Data frames

Un *data frame* es una estructura bidimensional cuyos elementos se organizan por filas y columnas de manera similar a una matriz. La principal diferencia con las matrices es que sus columnas están formadas por vectores, pero pueden

tener tipos de datos distintos. Un data frame es un caso particular de lista formada por vectores del mismo tamaño con nombre.

Los data frames son las estructuras de datos más utilizadas en R para almacenar los datos en los análisis estadísticos.

3.5.1 Creación de un data frame

Para crear un data frame se utiliza la siguiente función:

- `data.frame(nombrex = x, nombrey = y, ...)`: Devuelve el data frame con columnas los vectores `x`, `y`, etc. y nombres de columna `nombrex`, `nombrey`, etc.

Ejemplo 3.47. A continuación se muestran varios ejemplos de la creación de data frames.

```
df <- data.frame(asignatura = c("Matemáticas", "Física", "Economía"), nota = c(8.5, 7, 4.5))
df
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
#> 3    Economía  4.5
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ asignatura: chr  "Matemáticas" "Física" "Economía"
#>  $ nota      : num  8.5 7 4.5
# Data frame vacío
data.frame()
#> data frame with 0 columns and 0 rows
```

Para grandes conjuntos de datos es más común crear un data frame a partir de un fichero en formato csv mediante la siguiente función:

- `read.csv(f)`: Devuelve el data frame que se genera a partir de los datos del fichero csv `f`. Cada fila del fichero csv se corresponde con una fila del data frame y por defecto utiliza la coma `,` parara separar los datos de las columnas y punto `.` como separador de decimales de los datos numéricos. Los nombres de las columnas se obtienen automáticamente a partir de la primera fila del fichero.
- `read.csv2(f)`: Funciona igual que la función anterior pero utiliza como separador de columnas el punto y coma `;` y como separador de decimales la coma `,`.

Ejemplo 3.48. A continuación se muestra un ejemplo de creación de un data frame a partir de un fichero csv.

```
df <- read.csv('https://raw.githubusercontent.com/asalber/manual-r/master/datos/colesterol.csv')
df
```

```
#>                              nombre edad sexo peso altura
#> 1     José Luis Martínez Izquierdo   18    H   85   1.79
#> 2                  Rosa Díaz Díaz   32    M   65   1.73
#> 3            Javier García Sánchez   24    H   NA   1.81
#> 4              Carmen López Pinzón   35    M   65   1.70
#> 5             Marisa López Collado   46    M   51   1.58
#> 6                Antonio Ruiz Cruz   68    H   66   1.74
#> 7          Antonio Fernández Ocaña   51    H   62   1.72
#> 8            Pilar Martín González   22    M   60   1.66
#> 9             Pedro Gálvez Tenorio   35    H   90   1.94
#> 10         Santiago Reillo Manzano   46    H   75   1.85
#> 11           Macarena Álvarez Luna   53    M   55   1.62
#> 12      José María de la Guía Sanz   58    H   78   1.87
#> 13 Miguel Angel Cuadrado Gutiérrez   27    H  109   1.98
#> 14           Carolina Rubio Moreno   20    M   61   1.77
#>    colesterol
#> 1         182
#> 2         232
#> 3         191
#> 4         200
#> 5         148
#> 6         249
#> 7         276
#> 8          NA
#> 9         241
#> 10        280
#> 11        262
#> 12        198
#> 13        210
#> 14        194
```

3.5.2 Coerción de otras estructuras de datos a data frames

Para convertir otras estructuras de datos en data frames, se utiliza la siguiente función:

- `as.data.frame(x)`: Devuelve el data frame que se obtiene a partir la estructura de datos `x` a plicanco las siguientes reglas de coerción:
  - Si `x` es un vector se obtiene un data frame con una sola columna.
  - Si `x` es una lista se obtiene un data frame con tantas columnas como elementos tenga la lista. Si los elementos de la lista tienen tamaños distintos se obtiene un error.
  - Si `x` es una matriz se obtiene un data frame con el mismo número de columnas y filas que la matriz.

3.5.3 Acceso a los elementos de un data frame

Puesto que un data frame es una lista, se puede acceder a sus elementos como se accede a los elementos de una lista utilizando índices. Con corchetes simples `[ ]` se obtiene siempre un data frame, mientras que con corchetes dobles `[[ ]]` o `$` se obtiene un vector. Pero también se puede acceder a los elementos de un data frame como si fuese una matriz, indicando un par de índices para las filas y las columnas respectivamente.

Ejemplo 3.49. A continuación se muestran varios ejemplos de acceso a los elementos de un data frame.

```
df <- data.frame(asignatura = c("Matemáticas", "Física", "Economía"), nota = c(8.5, 7, 4.5))
df
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
#> 3    Economía  4.5
# Acceso como lista
df["asignatura"]
#>    asignatura
#> 1 Matemáticas
#> 2      Física
#> 3    Economía
df$asignatura
#> [1] "Matemáticas" "Física"      "Economía"
# Acceso como matriz
df[2:3, "nota"]
#> [1] 7.0 4.5
df[df$nota >= 5, ]
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
```

Obsérvese en el último ejemplo anterior cómo se pueden utilizar condiciones lógicas para filtrar un data frame.

Para acceder a las primeras o últimas filas de un data frame se pueden utilizar las siguientes funciones:

- `head(df, n)`: Devuelve un data frame con las `n` primeras filas del data frame `df`.
- `tail(df, n)`: Devuelve un data frame con las `n` últimas filas del data frame `df`.

Estas funciones son útiles para darse una idea del contenido de un data frame con muchas filas.

Ejemplo 3.50. A continuación se muestran varios ejemplos de acceso a las

primeras o últimas filas de un data frame.

```
df <- data.frame(x = 1:26, y = letters) # letters es un vector predefinido con las let
head(df, 3)
#>   x y
#> 1 1 a
#> 2 2 b
#> 3 3 c
tail(df, 2)
#>     x y
#> 25 25 y
#> 26 26 z
```

3.5.4 Modificación de los elementos de un data frame

Para modificar uno o varios elementos de un data frame basta con acceder a esos elementos y usar el operador de asignación para asignar nuevos valores.

Ejemplo 3.51. A continuación se muestran varios ejemplos de modificación de los elementos de un vector.

```
df <- data.frame(asignatura = c("Matemáticas", "Física", "Economía"), nota = c(8.5, 7,
df
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
#> 3    Economía  4.5
df[3, "nota"] <- 5
df
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
#> 3    Economía  5.0
```

3.5.5 Añadir elementos a una matriz

Para añadir nuevas filas o columnas a una matriz se utilizan las mismas funciones que para matrices:

- `rbind(df, x)`: Devuelve el data frame que resulta de añadir nuevas filas al data frame `df` con los elementos de la lista `x`.
- `cbind(df, nombrex = x)`: Devuelve el data frame que resulta de añadir nuevas columnas al data frame `df` con los elementos del vector `x` con nombre `nombrex`.

Ejemplo 3.52. A continuación se muestran varios ejemplos de añadir nuevas filas y columnas a un data frame.

```r
df <- data.frame(asignatura = c("Matemáticas", "Física", "Economía"), nota = c(8.5, 7, 4.5))
df
#>    asignatura nota
#> 1 Matemáticas  8.5
#> 2      Física  7.0
#> 3    Economía  4.5
# Añadir una nueva fila
rbind(df, list("Programación" , 10))
#>     asignatura nota
#> 1  Matemáticas  8.5
#> 2       Física  7.0
#> 3     Economía  4.5
#> 4 Programación 10.0
# Añadir una nueva columna
cbind(df, créditos = c(6, 4, 3))
#>    asignatura nota créditos
#> 1 Matemáticas  8.5        6
#> 2      Física  7.0        4
#> 3    Economía  4.5        3
```

3.5.6 Eliminar filas y columnas de un data frame

Para eliminar una columna de un data frame basta con acceder a la columna y asignarle el valor `NULL`, mientras que para eliminar una fila basta con acceder a la fila con índice negativo.

Ejemplo 3.53. A continuación se muestran varios ejemplos de eliminación de filas y columnas de un data frame.

```r
df <- data.frame(asignatura = c("Matemáticas", "Física", "Economía"), nota = c(8.5, 7, 4.5), créd
df
#>    asignatura nota créditos
#> 1 Matemáticas  8.5        6
#> 2      Física  7.0        4
#> 3    Economía  4.5        3
# Eliminar una columna
df$nota <- NULL
df
#>    asignatura créditos
#> 1 Matemáticas        6
#> 2      Física        4
#> 3    Economía        3
# Eliminar una fila
df[-2, ]
#>    asignatura créditos
#> 1 Matemáticas        6
```

```
#> 3    Economía        3
```

Chapter 4

Estructuras de control

4.1 Estructuras condicionales

4.1.1 La instrucción `if`

4.1.2 La función `ifelse()`

4.1.3 La función `switch()`

4.2 Bucles

4.2.1 Bucles iterativos (`for`)

4.2.2 Bucles condicionales

4.2.2.1 La instrucción `while`

4.2.2.2 La instrucción `repeat`

Chapter 5

Funciones

Una función es un bloque de código que tiene asociado un nombre, de manera que cada vez que se quiera ejecutar el bloque de código basta con invocar el nombre de la función. Las funciones permite dividir el código en unidades lógicas que resultan más fáciles de manejar y mantener.

En R las funciones son objetos en sí mimas y pueden usarse como cualquier otro dato. El tipo de dato de las funciones es `function`.

5.1 Definición y llamada a funciones

Para definir una función se utiliza la siguiente estructura de código:

```
nombre.funcion <- function (parámetros) {
  código
}
```

El código que va entre llaves se conoce como *cuerpo de la función*.

Para llamar a la función y que se ejecute el código de su cuerpo hay que utilizar el nombre de la función y a continuación los valores pasados a sus parámetros entre paréntesis.

Ejemplo 5.1. A continuación se muestra un ejemplo de creación y llamada a una función.

```
# Definición de la función
saludo <- function() {
  print("¡Hola!")
}
class(saludo)
#> [1] "function"
# Llamada a la función
```

```
saludo()
#> [1] "¡Hola!"
```

5.2 Parámetros y argumentos de una función

Una función puede recibir valores cuando se invoca a través de unas variables conocidas como *parámetros* que se definen entre paréntesis en la declaración de la función. En el cuerpo de la función se pueden usar estos parámetros como si fuesen variables.

Los valores que se pasan a la función en una llamada o invocación concreta de ella se conocen como *argumentos* y se asocian a los parámetros de la declaración de la función.

Ejemplo 5.2. A continuación se muestra un ejemplo de una función con parámetros.

```
# Función con un parámetro
saludo <- function(nombre) {
  print(paste("¡Hola ", nombre, "!", sep = ""))
}
# Llamada a la función con un argumento
saludo("Alf")
#> [1] "¡Hola Alf!"
```

En este ejemplo la función `saludo` tiene un parámetro `nombre`. En la llamada a la función se pasa la cadena `Alf` como argumento que se asocia al parámetro `nombre` en el cuerpo de la función.

5.2.1 Paso de argumentos a una función

Los argumentos de una función pueden pasarse de dos formas:

- **Argumentos posicionales**: Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.
- **Argumentos nominales**: Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`. En este caso el orden de los argumentos no importa.

Ejemplo 5.3. A continuación se muestran varios ejemplos de pasos de argumentos posicionales y nominales.

```
# Función con un argumento por defecto
area.triangulo <- function(base, altura) {
  base * altura / 2
}
# Cálculo del área de un triángulo de base 4 y altura 3
# Paso de argumentos por posición.
```

```
area.triangulo(4, 3)
#> [1] 6
# Paso de argumentos por nombre
area.triangulo(altura = 3, base = 4)
#> [1] 6
```

5.2.2 Argumentos por defecto

En la definición de una función se puede asignar a cada parámetro un argumento por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el argumento por defecto.

Ejemplo 5.4. A continuación se muestra un ejemplo de definición de una función con un argumento por defecto.

```
saludo <- function(nombre, lenguaje = "R") {
  print(paste("¡Hola ", nombre, "! ¡Bienvenido a ", lenguaje, "!", sep = ""))
}
# Llamada a la función con un argumento
saludo("Alf")
#> [1] "¡Hola Alf! ¡Bienvenido a R!"
```

5.3 Retorno de una función

Una función puede devolver un objeto de cualquier tipo tras su invocación. Para ello se utiliza la función `return()`, indicando entre paréntesis el valor que devuelve la función. El retorno suele realizarse al final del cuerpo de la función, porque con él finaliza la ejecución de la función y se devuelve el control de la ejecución al punto desde donde se llamó a la función, de manera que cualquier instrucción de cuerpo que vaya después no se ejecutará. Si no se indica ningún objeto, la función devolverá el valor de la última expresión calculada en el cuerpo de la función.

Ejemplo 5.5. A continuación se muestran varios ejemplos de retornos de funciones.

```
# Función que devuelve el area de un triángulo
area.triangulo <- function(base, altura) {
  return(base * altura / 2)
}
area.triangulo(4, 3)
#> [1] 6
# Función que devuelve el valor absoluto de un número
valor.absoluto <- function(x) {
  if (x < 0)
    return(x * -1)
```

```
  else
    return(x)
}
valor.absoluto(-1)
#> [1] 1
valor.absoluto(2)
#> [1] 2
```

Para devolver más de un valor se pueden utilizar estructuras de datos como vectores, listas, matrices o data frames.

Ejemplo 5.6. A continuación se muestra un ejemplo de una función de devuelve una lista.

```
circulo <- function(radio) {
  return(list(perimetro = 2 * pi * radio, area = pi * radio ^ 2))
}
circulo(5)
#> $perimetro
#> [1] 31.41593
#>
#> $area
#> [1] 78.53982
circulo(5)$perimetro
#> [1] 31.41593
circulo(5)$area
#> [1] 78.53982
```

5.4 Entorno y ámbito de las variables

El entorno de un programa en R es el conjunto de todos los objetos (funciones, variables, etc.) creados durante la ejecución del programa. Cuando se ejecuta el interprete de R siempre se crea un primer entorno `R_GlobalEnv` conocido como entorno global. Es posible referirse a él en cualquier momento con la constante `.GlobalEnv`.

Para ver el entorno activo en cada momento de la ejecución y el contenido del mismo se utiliza la siguiente función:

- `environment()`: Devuelve el nombre del entorno actual.
- `ls()`: Devuelve un vector con los nombres de las objetos (variables, funciones, etc.) que contiene el entorno global.

Ejemplo 5.7. A continuación se muestra un ejemplo acceso al entorno global de un programa.

```
x <- 4
y <- 3
```

```
area.triangulo <- function(base, altura) {
  base * altura / 2
}
environment()
#> <environment: R_GlobalEnv>
ls()
#> [1] "area.triangulo" "x"                "y"
```

Como se puede observar en el ejemplo anterior, los parámetros de la función `base` y `altura` no aparecen en el entorno global. En R, cuando se ejecuta una función se crea un nuevo entorno hijo dentro del entorno al que pertenece la función. Durante la ejecución de la función este pasa a ser el entorno activo y cuando termina la ejecución de la función deja de serlo y vuelve a activarse el entorno padre desde donde se llamó a la función.

Ejemplo 5.8. A continuación se muestra un ejemplo de activación del entorno de una función.

```
x <- 4
y <- 3
area.triangulo <- function(base, altura) {
  print("Entorno de la función area.triangulo")
  print(environment())
  print(ls())
  return(base * altura / 2)
}
print("Entorno fuera de la función")
#> [1] "Entorno fuera de la función"
environment()
#> <environment: R_GlobalEnv>
ls()
#> [1] "area.triangulo" "x"                "y"
area.triangulo(x, y)
#> [1] "Entorno de la función area.triangulo"
#> <environment: 0x5611d4edbda8>
#> [1] "altura" "base"
#> [1] 6
```

Los parámetros y los objetos (funciones, variables, etc.) definidos dentro de una función son de *ámbito local*, mientras que los objetos definidos fuera de ella en alguno de los entornos ancestros son de *ámbito global*.

Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función, es decir, cuando termina la ejecución de la función estas variables desaparecen y no son accesibles desde fuera de la función.

Cuando una función declara un objeto (función, variable, etc.) que ya existe en

alguno de los entornos ancestros con ámbito global, durante la ejecución de la función el objeto global queda eclipsado por el local y no es accesible hasta que finaliza la ejecución de la función.

Ejemplo 5.9. A continuación se muestra un ejemplo de eclipse de una variable de ámbito global por otra de ámbito local.

```
lenguaje = "Python"
saludo <- function(lenguaje) {
  print(paste("Bienvenido a", lenguaje))
}
saludo("R")
#> [1] "Bienvenido a R"
```

Obsérvese cómo al ejecutar la función anterior, la variable `lenguaje` queda inaccesible al tener la función un parámetro con el mismo nombre.

Las variables globales están accesibles siempre que no sean eclipsadas por otras con el mismo nombre de ámbito local. Si embargo, cuando se intenta asignar un valor a una variable global en el ámbito local, se crea una nueva variable local. Para asignar valores a variables globales en el ámbito local se tiene que utilizar el operador de superasignación `<<-`. Cuando se utiliza este operador para asignar un valor a una variable, R busca la variable entorno padre, y si no existe continua con la búsqueda en los entornos ancestros hasta llegar a entorno global. Si la búsqueda tiene éxito, asigna el nuevo valor a la variable global, mientras que si no tiene éxito se crea una nueva variable de ámbito local y se le asigna el valor.

Ejemplo 5.10. A continuación se muestra un ejemplo del uso del operador de superasignación.

```
saludo <- function() {
  lenguaje <<- "R"
  return(paste("Bienvenido a", lenguaje))
}
lenguaje
#> [1] "Python"
```

5.5 Componentes de una función

Los tres componentes de una función son:

- **Cuerpo**: Es el código dentro de la función.
- **Parámetros**: Es la lista de parámetros que requiere la función.
- **Entorno**: Es donde se ubican las variables de la función.

Para acceder a estos componentes se pueden utilizar las siguientes funciones:

- `body(f)`: Devuelve el cuerpo de la función `f`.

- `formals(f)`: Devuelve la lista de parámetros de la función `f`.
- `environment(f)`: Devuelve el entorno de la función `f`.

Ejemplo 5.11. A continuación se muestra un ejemplo de acceso a los componentes de una función.

```
# Definición de la función
area.triangulo <- function(base, altura) {
  base * altura / 2
}
body(area.triangulo)
#> {
#>     base * altura/2
#> }
formals(area.triangulo)
#> $base
#>
#>
#> $altura
environment(saludo)
#> <environment: R_GlobalEnv>
```

5.6 Funciones recursivas

Una función recursiva es una función que en su cuerpo contiene una llama a sí misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función. De lo contrario la recursión no tendría fin y nunca terminaría la ejecución de la función.

Ejemplo 5.12. A continuación se muestra un ejemplo de una función recursiva.

```
factorial <- function(n) {
  if (n <= 1) return(n)
  else return(n * factorial(n - 1))
}
factorial(4)
#> [1] 24
```

5.7 Paquetes

Para facilitar la reutilización código y datos R permite la creación de paquetes que pueden importarse desde otros programas. Un paquete es una colección de código, funciones y datos que se almacenan en un fichero dentro de un directorio llamado `library` en el entorno de R. Para ver la ubicación de este directorio dentro del sistema de archivos local se puede utilizar la función `.libPaths()`.

A continuación se muestra un ejemplo de la ubicación del directorio `library`.

```
.libPaths()
#> [1] "/home/alf/R/x86_64-pc-linux-gnu-library/4.1"
#> [2] "/usr/lib/R/library"
```

Durante la instalación de R también se instalan varios paquetes básicos que están disponibles en cualquier sesión de trabajo con R. Pero añadir nuevas funciones o procedimientos es necesario instalar el paquete que los contiene y después cargarlo en la sesión de trabajo.

Para ver los paquetes instalados en un ordenador se utiliza la función `library()`.

5.7.1 Instalación de paquetes

La mayor parte de los paquetes para R están disponibles en el repositorio oficial CRAN (Comprehensive R Archive Network), aunque cualquier persona puede desarrollar un paquete y ponerlo a disposición de la comunidad en cualquier otro repositorio.

Existen distintas formas de instalar un paquete en R:

- Directamente desde el repositorio oficial CRAN
- Desde otros repositorios no oficiales (por ejemplo Github)
- Descargando el paquete e instalándolo manualmente.

5.7.1.1 Instalación de paquetes desde el repositorio CRAN

Para instalar un paquete desde el repositorio oficial CRAN se utiliza la siguiente función:

- `install.packages(x)`: Obtiene el paquete con el nombre `x` desde un servidor con el repositorio CRAN y lo instala localmente en el directorio `library` del entorno de R. Se puede instalar más de un paquete a la vez pasando un vector con los nombres de los paquetes.

A continuación se muestra un ejemplo de instalación de paquetes desde el repositorio CRAN.

```
install.packages("devtools")
```

5.7.1.2 Instalación desde otros repositorios (GitHub, GitLab, etc.)

El paquete `remotes` incorpora funciones para instalar paquetes alojados en otros repositorios habituales para el desarrollo de software como GitHub, GitLab o Bioconductor.

A continuación se muestra un ejemplo de instalación de paquetes desde GitHub.

```
install.packages("remotes")
remotes::install_github("rkward-community/rk.Teaching")
```

5.7.1.3 Instalación manual

Finalmente es posible instalar un paquete manualmente a partir de su código fuente. Para ello hay previamente hay que descargar el código fuente del paquete en un fichero comprimido en formato zip y después utilizar la siguiente función:

- `install.packages(x, repos = NULL, type = "source")`: Instala el paquete ubicado en la ruta `x` del sistema de archivos local en la librería `library`.

Una vez instalado un paquete ya está disponible para cargarlo en cualquier sesión de trabajo de R y no es necesario volver a instalarlo.

5.7.2 Carga de un paquete

Una vez instalado un paquete, para poder ejecutar su contenido es necesario cargarlo en el entorno de trabajo de R. Para ello se utiliza la siguiente función:

- `library(x)`: Ejecuta el código del paquete `x` en la sesión de trabajo activa.

A continuación se muestra un ejemplo de carga de un paquete.

```
library("remotes")
```

5.7.3 Paquetes habituales

Chapter 6

Preprocesamiento de datos

Chapter 7

Gráficos y visualización de datos

7.1 Gramática de gráficos y el paquete `ggplot2`

7.2 Diagramas de puntos

7.3 Diagramas de barras

7.4 Diagramas de líneas

7.5 Histogramas

7.6 Diagramas de cajas

7.7 Diagramas de dispersión

7.8 Personalización de gráficos

7.8.1 Ejes

7.8.2 Leyendas

7.8.3 Facetas

Chapter 8

Estadística descriptiva

8.1 Tablas de frecuencias

8.1.1 Tablas de frecuencias de una variable

8.1.2 Tablas de frecuencias de dos variables (tablas de contingencia)

8.2 Estadísticos de tendencia central

8.2.1 Media

8.2.2 Mediana

8.2.3 Moda

8.3 Estadísticos de posición

8.3.1 Mínimo y máximo

8.3.2 Percentiles

8.4 Estadísticos de dispersión

8.4.1 Rango

8.4.2 Rango intercuartílico

8.4.3 Varianza y cuasivarianza

8.4.4 Desviación típica y cuasidesviación típica

8.4.5 Coeficiente de variación

8.5 Estadísticos de forma

8.5.1 Coeficiente de asimetría

8.5.2 Coeficiente de apuntamiento

8.6 Resúmenes descriptivos

Chapter 9

Estimación de parámetros y contrastes de hipótesis de variables cuantitativas

Chapter 10

Estimación de parámetros y contrastes de hipótesis de variables cualitativas

Chapter 11

Análisis de regresión simple