Capítulo 4: Tipos de objeto

Empecemos a familiarizarnos con el entorno de R, específicamente con su sintaxis. Para ello, primero vamos a plantear un ejemplo:

Si tengo una cesta de frutas con 5 naranjas, 7 limones y 2 manzanas, ¿cómo se escribiría en la consola?

```
naranjas <- 5
limones <- 7
manzanas <- 4
```

Para después responder preguntas como: ¿Cuántas frutas hay en total en mi cesta? o, si me como 2 manzanas, ¿cuántas frutas quedan en total en mi cesta?; se puede realizar operaciones básicas como en cualquier calculadora, pero con las variables previamente asignadas:

```
#Pregunta 1:
naranjas + limones + manzanas
```

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

```
#Pregunta 2:
naranjas + limones - manzanas
```

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

¿Cómo sería para responderlo en oración?

```
mi_nombre <- "Alexis"   #notamos las comillas para textos
mi_cesta <- naranjas + limones + manzanas
cat("Hola, mi nombre es", mi_nombre, "y tengo", mi_cesta, "frutas")
```

```
## Hola, mi nombre es Alexis y tengo 16 frutas
```

Entonces esta sección nos va a ayudar a responder este tipo de preguntas y algunas más complicadas como veremos más adelante.

Declaración de variables

R tiene un operador especial para asignar valores a variables, diferentes a otros lenguajes de programación. Además, las variables son flexibles y pueden pasar de contener un tipo de dato a otro sin problema. Por lo mismo no es necesario especificar el tipo de dato como se hace en C o en Java. El operador de asignación es: `<-` o `->`.

También se puede asignar variables con = (al igual que en Python) pero, por convención, = se reserva para operaciones dentro de funciones o paréntesis.

Además, R trabaja con numerosos tipos de datos. Algunos de los tipos más básicos para empezar son:

- Los valores decimales como ‘4.5‘ se llaman numéricos.
- Los números enteros como ‘4‘ se llaman números enteros. Los números enteros también son numéricos.
- Los valores booleanos (TRUE o FALSE) se llaman lógicos.
- Los valores de texto (o cadena) se llaman caracteres.

Observa cómo las comillas del editor indican que “ejemplo” es una cadena.

Vamos a asignar algunos valores a una serie de variables:

```
mi_numero <- 10
mi_caracter <- "mundo"
mi_logico <- FALSE

90 -> a
a
```

```
## [1] 90
```

También existe una función llamada `assign()` que es otra manera no tan común de asignar variables.

```
assign("x", 777)
x
```

```
## [1] 777
```

Si queremos ver si existe una variable, usamos la función `exists()`que es una manera de preguntarle a la consola si ya guardo esa variable:

```
exists("a")
```

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

Para borrar una variable en específico utilizo la función `rm()`:

```
rm(a)
#si ahora le pregunto a la consola si existe a:

exists("a")
```

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

si ahora quisiera borrar todas las variables utilizo la función `rm(list=ls())`, la explicación es que por si sola `ls()` hace lo siguiente:

```
ls() #enlista todas las variables
```

```
## [1] "limones"     "manzanas"    "mi_caracter" "mi_cesta"    "mi_logico"
## [6] "mi_nombre"   "mi_numero"   "naranjas"    "x"
```

entonces si combino eso con la línea de código anterior, va a borrar todo el `enviroment`hasta ese momento, pero no borra el historial de lineas de código que has ejecutado. De momento no vamos a utilizar esa función porque vamos a seguir trabajando con lo que hemos hecho hasta el momento.

¿Qué pasa si escribimos, por ejemplo, `5 + "seis"` en la consola? Lo que nos va a dar como resultado es un error. El error se debe a una falta de coincidencia en los datos. Para evitar este tipo de errores , comprueba previamente el tipo de datos de una variable. Puedes hacerlo con la función `class()`, como muestra el siguiente código:

```
class(mi_numero)
```

```
## [1] "numeric"
```

```r
class(mi_caracter)
```

```
## [1] "character"
```

```r
class(mi_logico)
```

```
## [1] "logical"
```

Nótese que se puede hacer antes o despues de ingresar valores a las variables:

```r
class(10)
```

```
## [1] "numeric"
```

```r
class("mundo")
```

```
## [1] "character"
```

```r
class(FALSE)
```

```
## [1] "logical"
```

Como describimos, no se necesita especificar qué tipo de dato queremos en cada variable ya que esto puede cambiar más adelante. Hablando de tipos de datos, vamos a ver cuáles son las opciones que maneja R:

Tipo de datos

En R, los tipos de datos más comunes son:

1. **Numérico (numeric)**: Se utilizan para representar números reales, ya sea enteros o decimales.
2. **Entero (integer)**: Utilizado para representar números enteros.
3. **Carácter (character)**: Utilizado para representar texto. Las cadenas de texto se escriben entre comillas simples o dobles.
4. **Lógico (logical)**: Representa valores booleanos que pueden ser `TRUE` o `FALSE`.
5. **Complejo (complex)**: Utilizado para representar números complejos con una parte real y una imaginaria.
6. **Fecha y hora (date y datetime)**: Existen clases específicas para manejar fechas y horas.
7. **Factores (factor)**: Se utilizan para representar variables categóricas.
8. **Listas (list)**: Estructura de datos que puede contener elementos de diferentes tipos.
9. **Vectores (vector)**: Colecciones ordenadas de elementos del mismo tipo, como vectores numéricos, de caracteres, lógicos, etc.

Estos son los tipos de datos más utilizados en R, pero existen otros tipos más especializados y estructuras de datos complejas como matrices, data frames, arrays, entre otros.

Ejemplo de diferentes tipos de datos en R

Para visualizar que tipo de dato es vamos a usar la función previamente mencionada `class()` y otra llamada `typeof()`:

```r
# Numérico
numero_real <- pi
print(numero_real)
```

```
## [1] 3.141593
```

```r
typeof(numero_real)
```

```
## [1] "double"
```

```r
class(numero_real)
```

```
## [1] "numeric"
```

En R, un `double` es un tipo de dato numérico que representa números reales (también conocidos como números de punto flotante o decimales).Es el tipo de dato numérico más común en R y tiene suficiente precisión para la mayoría de los cálculos científicos y estadísticos.

Los `doubles` pertenecen a la clase `numeric`. Si verificas el tipo con `typeof()`, obtendrás "double", pero la clase con `class()` será "numeric".

Tambíen puedes verificar si un número es un double con `is.double()`:

```r
# Entero
(numero_entero <- 42L)
```

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

```r
typeof(numero_entero)
```

```
## [1] "integer"
```

```r
class(numero_entero)
```

```
## [1] "integer"
```

```r
is.double(numero_entero)
```

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

Entonces, si `class` y `typeof` son funciones que se usan para inspeccionar los objetos, ¿por qué tienen diferentes respuestas de la consola? Si bien en un principio si son funciones que inspeccionan objetos pero tienen propósitos diferentes y proporcionan información en distintos niveles:

Por un lado, `class` indica la clase o el tipo de objeto a nivel de abstracción del usuario. Desde la perspectiva, describe cómo el objeto se comporta en términos de programación orientada a objetos (OO).

Por otro lado, `typeof` indica el tipo de dato subyacente en la implementación interna de R y desde la perspectiva, describe cómo está representado el objeto en memoria.

Entonces, ¿cuándo uso cada uno?:

`class()`: Úsalo cuando quieras entender cómo se comportará un objeto en tu análisis o script (por ejemplo, saber si es un factor, data.frame, matrix, etc.). Esto lo retomaremos más adelante.

`typeof()`: Úsalo cuando necesites comprender la representación interna para optimizar código o trabajar con funciones de bajo nivel.

Para nuestro propósito, sigamos revisando las dos respuestas de momento, sin darle mayor importancia a una u otra:

```r
# Carácter
texto <- "Hola, mundo!"
print(texto)
```

```
## [1] "Hola, mundo!"
```

```r
typeof(texto)
```

```
## [1] "character"
```

```r
class(texto)
```

```
## [1] "character"
```

```r
# Lógico
(es_verdadero <- TRUE)
```

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

```r
typeof(es_verdadero)
```

```
## [1] "logical"
```

```r
class(es_verdadero)
```

```
## [1] "logical"
```

```r
is.logical(es_verdadero)
```

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

como arriba vimos que existe la función `is.double`, también existen funciones como: `is.logical()`, `is.integer()`, `is.complex()`, etc., cuyas respuesta es un TRUE o FALSE.

```r
# Complejo
numero_complejo <- 2 + 3i
print(numero_complejo)
```

```
## [1] 2+3i
```

```r
typeof(numero_complejo)
```

```
## [1] "complex"
```

```r
class(numero_complejo)
```

```
## [1] "complex"
```

```r
is.complex(numero_complejo)
```

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

```r
# Fecha y hora
fecha <- as.Date("2022-12-31")
print(fecha)
```

```
## [1] "2022-12-31"
```

```r
typeof(fecha)
```

```
## [1] "double"
```

```r
class(fecha)
```

```
## [1] "Date"
```

```r
# Factores
factor_variable <- factor(c("A", "B", "A", "C", "B"))
print(factor_variable)
```

```
## [1] A B A C B
## Levels: A B C
```

```r
typeof(factor_variable)
```

```
## [1] "integer"
```

```r
class(factor_variable)
```

```
## [1] "factor"
```

```r
is.factor(factor_variable)
```

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

```r
# Listas
lista <- list(nombre = "Juan", edad = 30, casado = TRUE)
print(lista)
```

```
## $nombre
## [1] "Juan"
##
## $edad
## [1] 30
##
## $casado
## [1] TRUE
```

```r
typeof(lista)
```

```
## [1] "list"
```

```r
class(lista)
```

```
## [1] "list"
```

```r
is.list(lista)
```

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

```r
# Vectores
(vector_numerico <- c(1, 2, 3, 4, 5))
```

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

```r
typeof(vector_numerico )
```

```
## [1] "double"
```

```r
class(vector_numerico)
```

```
## [1] "numeric"
```

```r
is.vector(vector_numerico)
```

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

getwd & setwd

Una de las cosas importantes del uso de R es saber dónde estamos trabajando, para saber dónde se está guardando la información que estamos generando utilizamos la siguiente función: `getwd()`. Esto nos va a dar como respuesta el directorio de la computadora donde estamos trabajando. Pero si quisieramos cambiar de directorio, utilizaremos la siguiente función: `setwd()` y dentro del parentesis pondremos entre comillas el directorio en el que quisiéramos trabajar ahora.

Otras funciones importantes exclusivas para Windows:

- choose.dir(). Imprime la ruta a un folder.
- choose.files(). Imprime la ruta a un archivo.

Estas funciones nos van a ayudar en la navegación de archivos dentro de nuestra computadora.

Una alternativa a `choose.file()` existe la función `file.choose()`. Que nos va a dar la ruta a un archivo específico, sirve para Mac y Windows.

```r
# ruta_archivo <- file.choose()
# ruta_archivo
```

Biblioteca `readr`

"El objetivo de 'readr' es proporcionar una forma rápida y sencilla de leer datos rectangulares (como 'csv', 'tsv' y 'fwf'). Está diseñado para analizar de forma flexible muchos tipos de datos que se encuentran en la red, y al mismo tiempo fallar sin problemas cuando los datos cambian inesperadamente" (R Core Team, s.f.).

```
library(readr)
 # mitabla <- read_csv(file.choose())
```

Esto nos va a facilitar leer bases de datos y los va a guardar dentro de la variable mitabla. Funciona para macOs y Windows

Otras funciones:

```
list.dirs()
```

```
## [1] "."                              "./Clase-2_files"
## [3] "./Clase-2_files/figure-latex"
```

Me enlista los directorios o folderos del directorio de trabajo actual.

```
list.files()
```

```
## [1] "Clase 2.Rmd"   "Clase-2_files" "Clase-2.pdf"   "Clase-2.Rmd"
## [5] "Clase-2.tex"
```

Me lista todos los archivos que tenemos del directorio de trabajo actual.

Operadores

La mayor parte de los operadores de R van a ser similares a los de otros sistemas operativos, veamos algunos ejemplos:

Operadores aritméticos

Las expresiones aritméticas expresan cálculos numéricos. Como hemos visto, pueden usarse en asignaciones. El resultado de evaluar la expresión es lo que se asigna a la variable. Una expresión aritmética puede contener:

- valores literales: son valores concretos de un tipo de dato, como el valor 7.
- variables: al evaluar la expresión una variable se sustituye por su valor asociado.
- operadores aritméticos, como: $+$, $-$, $*$, $/$, $\wedge$
- llamadas a funciones: las funciones producen valores en función de sus parámetros.

Veamos algunos ejemplos de expresiones aritméticas:

```
#Suma(+)
(2+2)
```

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

```
#Resta(-)
(3-2)
```

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

```
#Multiplicación(*)
(2*4)
```

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

```
#División(/)
(8/2)
```

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

```
#División entera (%/%)
10 %/% 3
```

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

```
#Potencia(**)
(5**2)
```

```
## [1] 25
```

```
#Raíz
(25**(1/2))
```

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

```r
#La raíz también puede obtenerse por la función sqtr()
(sqrt(36))
```

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

```r
#Módulo(residuo de una división)(%%)
(5%%2)
```

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

```r
#### Jerarquía de Operaciones ####

(2 * 3) + 2
```

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

```r
2 * (3 + 2)
```

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

```r
2 * 3 + 2
```

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

```r
2 + 3 * 4 /2
```

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

También se puede hacer operación entre variables:

```r
x <- 5
y <- 2
(suma <- x+y)
```

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

```r
#otra forma:
(x+y)
```

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

```r
(resta<-x-y)
```

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

```r
(multiplicación <- x*y)
```

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

```
division <- x/y
print(division)
```

```
## [1] 2.5
```

```
(potencia <- x**y)
```

```
## [1] 25
```

```
#otra manera de usar la potencia es con ^
(potencia2 <- x^y)
```

```
## [1] 25
```

```
(modulo <- x%%y)
```

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

Operadores de comparación:

```
a <- 5
b <- 10
(igualdad <- x == y)
```

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

```
(desigualdad <- x != y)
```

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

```
(menor_que <- x < y)
```

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

```
(mayor_que <- x > y)
```

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

```
(menor_o_igual <- x <= y)
```

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

```
(mayor_o_igual <- x >= y)
```

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

Operadores lógicos

```
# and lógico
TRUE & TRUE
```

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

```
FALSE & FALSE
```

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

```
T & F
```

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

```
# or lógico
TRUE | TRUE
```

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

```
T | F
```

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

```
F | F
```

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

```
#not lógico

!F & !F
```

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

```
!T | !F
```

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

```
p <- TRUE
q <- FALSE
(and_logico <- p & q)
```

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

```
(or_logico <- p | q)
```

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

```
(not_logico <- !p)
```

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

Concatenación

En R se usa la función `paste()` para concatenar cadenas de texto.

```r
nombre <- "Juan"
apellido <- "Pérez"
nombre_completo <- paste(nombre, apellido)
print(nombre_completo)
```

```
## [1] "Juan Pérez"
```

Operador de existencia

Para verificar la existencia de un valor en un vector, usamos la función

```r
a <- c(1, 2, 3, 4, 5)
b <- 3
(resultado <- b %in% a)
```

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

Algunas otras operaciones comunes:

Otras funciones básicas muy utilizadas en estadística son: sin, cos, tan, asin, acos, atan, atan2, log, logb, log10, exp, sqrt, abs. A continuación algunos ejemplos de las anteriores funciones.

```r
sin(pi/2)
```

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

```r
cos(0)
```

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

```r
tan(pi/4)
```

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

```r
asin(0)
```

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

```r
acos(1)
```

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

```r
atan(0)
```

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

```r
atan2(2,0)
```

```
## [1] 1.570796
```

```r
log(0)
```

```
## [1] -Inf
```

```r
log(10,base = 2)
```

```
## [1] 3.321928
```

```r
log2(10)
```

```
## [1] 3.321928
```

```r
log10(10)
```

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

```r
exp(2)
```

```
## [1] 7.389056
```

```r
sqrt(4)
```

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

```r
abs(-4)
```

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

Capítulo 5: Vectores

Los vectores en En R son objetos de una sola dimensión que pueden contener datos del **mismo tipo**, ya sea numéricos, cadenas de carácter, o datos lógicos, entre otros. Esencialmente son uno de los elementos básicos en la estructura de los datos en R .

Creación de vectores

Para crear un vector usamos la función combine `c()` de la siguiente manera:

```
mi_vector <- c(1,7,5,10)
mi_vector
```

```
## [1]  1  7  5 10
```

```
class(mi_vector)
```

```
## [1] "numeric"
```

Ahora vemos como una variable puede almacenar más de un solo valor.

Si quisiera crear un vector más grande, por ejemplo una secuencia del 1 al 100. En lugar de escribir 100 numeros manualmente, lo hacemos de la siguiente manera:

```
(mi_vector2 <- 1:100)
```

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

Note que puede hacer lo mismo empezando desde cualquiere otro numero hasta el número que se le ocurra, lo importante es notar que los dos puntos nos ayudan a escribir números consecutivos:

```
8:11
```

```
## [1]  8  9 10 11
```

```
-5:1
```

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

```
10:1
```

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

```
4977:5000
```

```
##  [1] 4977 4978 4979 4980 4981 4982 4983 4984 4985 4986 4987 4988 4989 4990 4991
## [16] 4992 4993 4994 4995 4996 4997 4998 4999 5000
```

Aunque no es fácil escribrir esta secuencia de numeros consecutivos, a veces puede resultar algo impráctica. Sin embargo, podemos crear secuencias de números de una forma sencilla usando la función `seq`, la estructura de esta función es:

```
seq(from=0, to=1, length.out = 11)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Para ahora saber la longitud de mi vector podemos usar la función `length()` que me va a dar como respuesta la longitud de mi vector:

```
length(mi_vector)
```

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

```
length(mi_vector2)
```

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

Otros ejemplos de vectores son los siguientes:

```
vector_numerico <- c(1, 3, 5, 7)
class(vector_numerico)
```

```
## [1] "numeric"
```

```
vector_texto <- c("a", "b", "c", "d")
class(vector_texto)
```

```
## [1] "character"
```

También existe una manera de crear un vector de tipo `character` en orden alfabético con dos funciones:

```
minúsculas <- letters[1:10]
MAYÚSCULAS <- LETTERS[1:10]

minúsculas
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
MAYÚSCULAS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

```
vector_logico <- c(TRUE, FALSE, TRUE)
class(vector_logico)
```

```
## [1] "logical"
```

Otra manera de declarar vectores es utilizando la función `vector()`

```
(mi_vector3 <- vector(mode= "numeric", length = 8))
```

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

Notamos como nos da de respuesta un vector con valores nulos pero de longitud 8, esto es debido a que si especificamos la longitud pero no los valores. Sin embargo, si especificamos que sea un vector numérico.

Los tipos de vectores que puede haber son tantos como la función `typeof()`lo permita, y para ver que tipo de valores son listados en la función `typeof()` vease la función `help()` esta función nos va a ayudar para saber la estructura de cada función en R:

```
help("typeof")
```

La respuesta nos abrirá una pagina en internet con la descripción de cada función que le pongamos así como su estructura.

Regresando al tema, también puede haber vectores mixtos:

```
mixto <- c(7, "a", T, 49, "carro", F)
mixto
```

```
## [1] "7"     "a"     "TRUE"  "49"    "carro" "FALSE"
```

```
#si quisieramos saber si seguimos trabajando con un vector utilizamos la siguiente función:

is.vector(mixto)
```

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

Vemos como efectivamente seguimos trabajando con vectores.

Una cosa importante es diferenciar entre la estructura de dato y el tipo de dato.

```
ej_vec <- c(10,5)

is.vector(ej_vec) #Es la estructura de dato
```

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

```
is.numeric(ej_vec) #Es el tipo de dato
```

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

```
is.character(ej_vec)
```

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

```
is.character(mixto)
```

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

Nótese como R en el caso de datos mixtos, todos los va a guardar como character.

Ejemplo 1

Si tengo las siguientes calificaciones de un examen de R que se aplicó el día de ayer: 4, 10, 7, 8,8 No presentó, 9 y 5. ¿Cómo escribo esta información como un vector de tal manera que las calificaciones faltantes se representen adecuadamente y no quede fuera ningún alumno del conteo?

```
(vecNA <- c( 4,10, 7, 8, 8, NA, 9, 5))
```

```
## [1]  4 10  7  8  8 NA  9  5
```

```r
length(vecNA)
```

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

```r
class(vecNA)
```

```
## [1] "numeric"
```

Note que el vector sigue siendo numeric a pesar de que tiene un NA, esto es porque NA no es texto, ni siquiera esta entre comillas, NA es un valor faltante del cual hablaremos más adelante.

```r
is.na(vecNA)
```

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

Ejemplo 2

Si se sabe que el alumno que no presentó el examen, lo presentó al dia siguiente y obtuvo una calificación de 7. ¿Cuál fue el promedio de ese examen? Para responder esta pregunta utilizaré las funciones `replace()` y `mean()`

```r
califs <- replace(vecNA, 6, 7 )
califs
```

```
## [1]  4 10  7  8  8  7  9  5
```

```r
mean(califs)
```

```
## [1] 7.25
```

La estructura de la función replace nos da en el primer lugar, el objeto en el cual vamos a reemplazar el valor, en este caso era el 'vecNA', el 6 es la posición del elemento a reemplazar y el 7 es el objeto por el cual se va a sustituir el valor NA.

Mientras que mean, es el promedio del nuevo vector ya sin ningún valor faltante. Pero si quisieramos hacerlo manual de sumar todos los elementos de un vector y después dividirlos entre el número de elementos del mismo vector, lo haremos utilizando operaciones entre vectores:

Operaciones de vectores

Las operaciones entre vectores van a funcionar igual que las operaciones normales, sin embargo se tienen que cumplir algunas condiciones:

```r
y <- c(1,2,3,4,5,6)
print(y)
```

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

```r
# Si los vectores no son de la misma longitud:
  y + c(-1, 1, -1, 1,-1, 1, -1, 1)
```

```
## Warning in y + c(-1, 1, -1, 1, -1, 1, -1, 1): longer object length is not a
## multiple of shorter object length
```

```
## [1] 0 3 2 5 4 7 0 3
```

```
# Los vectores mas cortos se repiten hasta tener la longitud del vector mas largo

  y * c(1, 2, 3)
```

```
## [1]  1  4  9  4 10 18
```

Una vez con esto en mente, fijémonos en el siguiente ejemplo:

```
op_vec <- c(1,5)
op_vec +1
```

```
## [1] 2 6
```

```
op_vec2 <- op_vec +1
op_vec2 -1
```

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

```
(op_vec2 -1) == op_vec
```

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

```
op_vec3 <- c(8,9)

op_vec3 + op_vec2 + op_vec
```

```
## [1] 11 20
```

```
op_vec/5
```

```
## [1] 0.2 1.0
```

```
- op_vec
```

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

Funciones sobre vectores

En R podemos destacar las siguientes funciones básicas sobre vectores numéricos.

- ‘min‘: para obtener el mínimo de un vector.
- ‘max‘: para obtener el máximo de un vector.
- ‘length‘: para determinar la longitud de un vector.
- ‘range‘: para obtener el rango de valores de un vector, entrega el mínimo y máximo.
- ‘sum‘: entrega la suma de todos los elementos del vector.
- ‘prod‘: multiplica todos los elementos del vector.
- ‘which.min‘: nos entrega la posición en donde está el valor mínimo del vector.
- ‘which.max‘: nos da la posición del valor máximo del vector.
- ‘rev‘: invierte un vector.

Ejemplo:

```r
fun_vec <- c(1,2,3)
min(fun_vec)
```

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

```r
max(fun_vec)
```

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

```r
length(fun_vec)
```

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

```r
range(fun_vec)
```

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

```r
sum(fun_vec)
```

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

```r
prod(fun_vec)
```

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

```r
which.min(fun_vec)
```

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

```r
which.max(fun_vec)
```

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

```r
rev(fun_vec)
```

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

Por último, cada elemento del vector puede tener un 'nombre' y se le asigna de la siguiente manera:

```r
(vec2 <- c(elemento1=47484, elemento2= "akfaj"))
```

```
## elemento1 elemento2
##   "47484"   "akfaj"
```

Ejercicio

El siguiente vector contiene la cantidad de Zinc en la sangre de una muestra de x personas,

```
zinc <- c(3, 5.8, 5.6, 4.8, 5.1, 3.6, 5.5, 4.7, 5.7, 5, 5.9, 5.7, 4.4, 5.4, 4.2, 5.3, NA)
```

Conteste lo siguiente:

- ¿Cuántas personas son?
- ¿Cuál es el promedio de zinc en la sangre de la muestra?
- ¿Cuál es la desviación estándar?
- ¿Cuál es la mediana?
- Ordene los elementos de menor a mayor y de mayor a menor.
- Haga un resumen estadístico con la función 'summary()'.
- Haga una gráfica de caja, un histograma y grafique la densidad del vector.

Respuestas

- ¿Cuántas personas son?

```
length(zinc)
```

```
## [1] 17
```

- ¿Cuál es el promedio de zinc en la sangre de la muestra?

```
#mean

mean(zinc, na.rm = T) #el na.rm va a quitar todos los valores NA del vector
```

```
## [1] 4.98125
```

```
#funciones sobre vectores

(zinc2 <- zinc[-17])
```

```
##  [1] 3.0 5.8 5.6 4.8 5.1 3.6 5.5 4.7 5.7 5.0 5.9 5.7 4.4 5.4 4.2 5.3
```

```
#note cómo elimine el elemento 17 con la estructura:
## <<nombre_del_vector>>[-<<valor no deseado>>] ##
sum(zinc2)/length(zinc2)
```

```
## [1] 4.98125
```

Volviendo a la parte de funciones sobre vectores, la estructura que usamos para quitar el valor NA del vector zinc es solo un uso de esa estructura, en general la estructura «`nombre_del_vector`»[...] nos va a ayudar a filtrar información del vector. Por ejemplo, hay otra manera de remover elementos que no queremos analizar:

```
is.na(zinc)
```

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

```
!is.na(zinc) #notese el ! antes de la función is.na
```

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

```
zinc[!is.na(zinc)]
```

```
##  [1] 3.0 5.8 5.6 4.8 5.1 3.6 5.5 4.7 5.7 5.0 5.9 5.7 4.4 5.4 4.2 5.3
```

- ¿Cuál es la desviación estándar?

```
sd(zinc, na.rm = T)
```

```
## [1] 0.8320407
```

Recordar que la desviación estándar es la variación de los datos respecto a su media

- ¿Cuál es la mediana?

```
median(zinc, na.rm = T)
```

```
## [1] 5.2
```

- Ordene los elementos de menor a mayor y de mayor a menor.

```
sort(zinc2)
```

```
##  [1] 3.0 3.6 4.2 4.4 4.7 4.8 5.0 5.1 5.3 5.4 5.5 5.6 5.7 5.7 5.8 5.9
```

```
sort(zinc2,  T)
```

```
##  [1] 5.9 5.8 5.7 5.7 5.6 5.5 5.4 5.3 5.1 5.0 4.8 4.7 4.4 4.2 3.6 3.0
```

note cómo no tueve que poner en el argumento de la función decreasing = T, nada más con poner T basta

- Haga un resumen estadístico con la función 'summary()'.

```
summary(zinc)
```

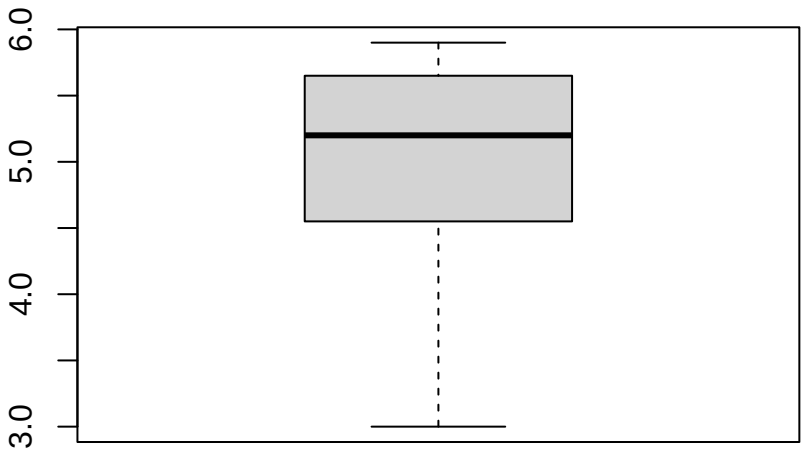
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   3.000   4.625   5.200   4.981   5.625   5.900       1
```

```
summary(zinc2)
```

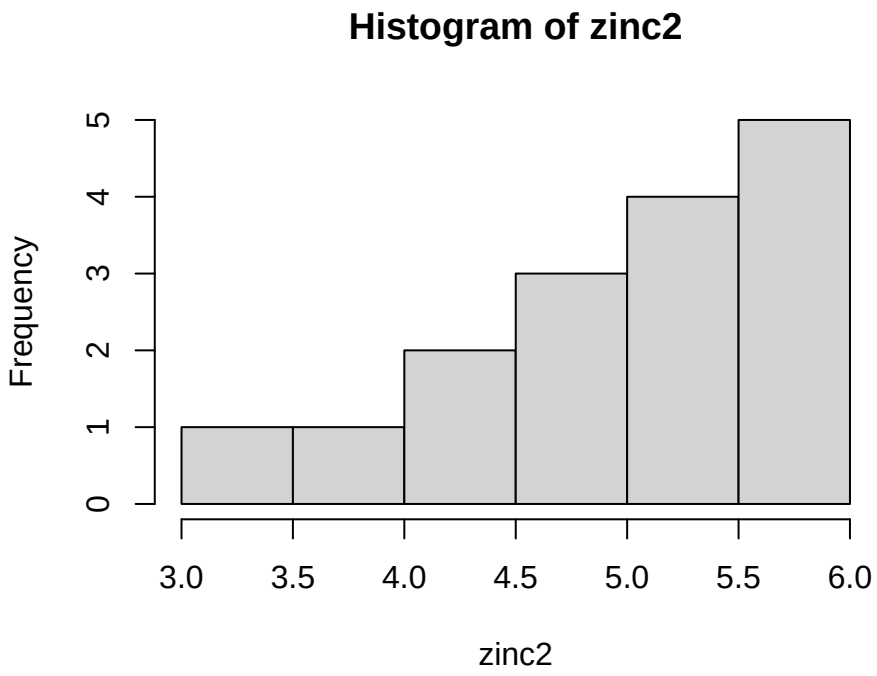
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.000   4.625   5.200   4.981   5.625   5.900
```

- Haga una grafica de caja, un histograma y grafique la densidad del vector.

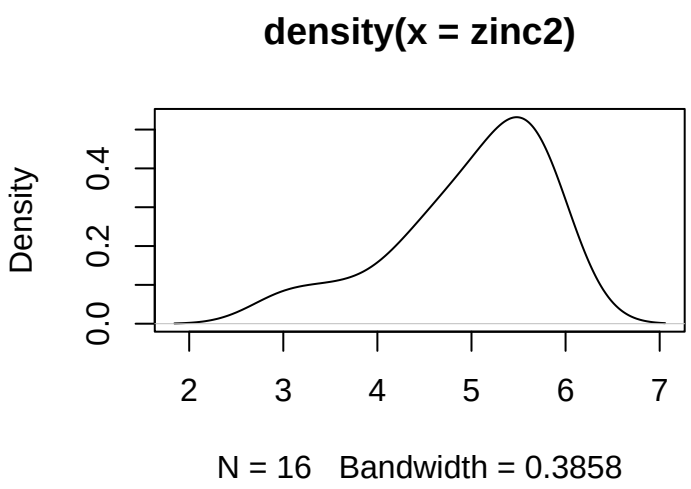
```r
boxplot(zinc2)
```




```r
hist(zinc2)
```



```
plot(density(zinc2))
```



Estas formas de gráficar así como cosas más avanzadas las retomaremos más adelante.

Extras

```
c(1:3, 99, 800)
```

```
## [1]   1   2   3  99 800
```

```
c(c(1:3, 99, 800),777)
```

```
## [1]   1   2   3  99 800 777
```

```
(re<- rep(10,times = 3))
```

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

```
is.vector(re)
```

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

```
rep(c(1,2), times = 3) #repite el vector 3 veces
```

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

```
rep(c(1,2), each = 3) # repite cada elemento del vector 3 veces
```

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

```
seq(-3,1, length=3) # número de elementos en ese intervalo
```

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

```r
seq(-3,1, by=0.5)
```

```
## [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0
```

Posiciones en vectores

```r
x <- 7:14
x
```

```
## [1]  7  8  9 10 11 12 13 14
```

```r
x[3] #posición 3
```

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

```r
x[3:6]
```

```
## [1]  9 10 11 12
```

```r
#sustituir valores específicos de un vector
x[2] <- 1000
x
```

```
## [1]    7 1000    9   10   11   12   13   14
```

Recuerde que los corchetes inmediatamente después del vector sirven para consultar el elemento en esa posición. Esto es muy útil en el caso en el que queramos consultar condiciones más complicadas

```r
x[x>10 & x<13]
```

```
## [1] 11 12
```

Por último, la función `is.element()`nos va a servir para verificar si algún valor es parte de un vector:

```r
is.element(10,x)
```

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

Capítulo 6: Matrices, Arreglos.

Matrices

Las matrices son arreglos rectangulares de filas y columnas con información numérica, alfanumérica o lógica. Para construir una matriz se usa la función matrix( ). Por ejemplo, para crear una matriz de 3x3:

```
mimatriz <- matrix(data=1:9, nrow=3, ncol=3, byrow = FALSE)
mimatriz
```

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

Notar como las entradas de la matriz se crean a partir de `data` y definimos la secuencia de 1 a 9, el siguiente parámetro `nrow` nos da el número de filas, `ncol` nos da el número de columnas y el último parámetro `byrow` nos da el orden con el que queremos que se acomoden los datos.

Si por ejemplo pusiéramos el último parámetro en TRUE:

```
mimatriz2 <- matrix(data=1:9, nrow=3, ncol=3, byrow = TRUE)
mimatriz2
```

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

Vease que el orden con el que me da los números cambia.

Al igual que en el caso de los vectores, para extraer elementos almacenados dentro de una matriz se usan los corchetes [,] y dentro, separado por una coma, el número de fila(s) y el número de columna(s) que nos interesan. Ejemplo:

```
mimatriz[1,2]
```

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

Si quisieramos un renglón completo, dejamos el lado de la fila vacío:

```
mimatriz[1,]
```

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

Note que la salida es en forma de vector.

Análogamente, para extraer una columna:

```
mimatriz[,2]
```

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

Note que la salida igual es en forma de vector.

Si quisieramos ver la matriz sin un renglón en específico, por ejemplo el 2:

```
mimatriz[,-c(2)]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

Note el signo - antes del vector.

Si quisieramos la matriz sin la fila 3:

```
mimatriz[-c(1),]
```

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

También se van a poder crear matrices diagonales de la siguiente forma:

```
D <- diag(c(3, 5))
D
```

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

Vemos como en la respuesta es el vector pero en forma de matriz diagonal y las entradas ($a_{1,1}$, $a_{2,2}$, ..., $a_{n,n}$) dependen de la posición del elemento en el vector.

De igual forma se va a poder crear la matriz identidad

```
I <- diag(2)
I
```

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

```
I2 <- diag(4)
I2
```

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

Álgebra lineal de matrices.

Las operaciones con matrices que se ven en álgebra lineal se van a poder replicar en R con los siguientes comandos:

Cuadro 1: Operaciones de matrices en R

Operación	Sintaxis
Transpuesta	`t()`
Diagonal	`diag()`
Traza	`sum(diag())`
Determinante	`det()`
Inversa	`solve()`
Descomposición QR	`qr()`
Rango	`qr()$rank`
Descomposición de Cholesky	`chol()`
Varianza	`var()`

Veamos un ejemplo usando algunas de estas operaciones:

Primero vamos a crear las matrices que vamos a usar:

```
A <- matrix(c(2, 1, 1, 3), nrow = 2, byrow = TRUE)  # Matriz 2x2
A
```

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

```
B <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)   # Matriz 2x3
B
```

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

```
t(A)        # Transpuesta de A
```

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

```
A + A       # Suma de matrices
```

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

```
A - A       # Resta de matrices
```

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

```r
A %*% A     # Producto de matrices
```

```
##      [,1] [,2]
## [1,]    5    5
## [2,]    5   10
```

```r
2 * A       # Multiplicación por un escalar
```

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

```r
det(A)      # Determinante de A
```

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

```r
solve(A)    # Inversa de A (si es invertible)
```

```
##      [,1] [,2]
## [1,]  0.6 -0.2
## [2,] -0.2  0.4
```

```r
qr(A)$rank  # Rango de A
```

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

Notar la importancia de que la matriz sea de la misma dimensión.

Tambien vamos a poder calcular los eigenvalores y eigenvectores con la función `eigen()`

```r
eig <- eigen(A)   # Calcula autovalores y autovectores
eig$values        # Autovalores
```

```
## [1] 3.618034 1.381966
```

```r
eig$vectors       # Autovectores (columnas)
```

```
##           [,1]       [,2]
## [1,] 0.5257311 -0.8506508
## [2,] 0.8506508  0.5257311
```

Además, se puede diagonalizar la matriz y elevarla a una potencia. Por ejemplo, si A es diagonalizable, se expresa como $A = PDP^{-1}$.

```r
P <- eig$vectors   # Matriz de autovectores
P
```

```
##           [,1]       [,2]
## [1,] 0.5257311 -0.8506508
## [2,] 0.8506508  0.5257311
```

```r
D <- diag(eig$values)  # Matriz diagonal de autovalores
D
```

```
##          [,1]     [,2]
## [1,] 3.618034 0.000000
## [2,] 0.000000 1.381966
```

```r
P_inv <- solve(P)   # Inversa de P
P_inv
```

```
##            [,1]      [,2]
## [1,]  0.5257311 0.8506508
## [2,] -0.8506508 0.5257311
```

```r
(A_diag <- P %*% D %*% P_inv)  # Debería ser igual a A
```

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

Ahora que vimos que nuestros cálculos son correctos, podemos elevar a una potencia la matriz:

```r
n <- 3   # Exponente deseado
D_n <- D^n  # Elevar la matriz diagonal a la n
D_n
```

```
##          [,1]    [,2]
## [1,] 47.36068 0.00000
## [2,]  0.00000 2.63932
```

```r
A_n <- P %*% D_n %*% P_inv  # A^n
A_n
```

```
##      [,1] [,2]
## [1,]   15   20
## [2,]   20   35
```

Si la matriz no fuese diagonalizable se puede usar la multiplicación repetida:

```r
n <- 3
A_n <- A
for (i in 2:n) {
  A_n <- A_n %*% A  # Multiplicación iterativa
  print(A_n)        # Muestra la matriz en cada iteración
}
```

```
##      [,1] [,2]
## [1,]    5    5
## [2,]    5   10
##      [,1] [,2]
## [1,]   15   20
## [2,]   20   35
```

El uso de `for` se verá más adelante

Arreglos

Un arreglo es una matriz de varias dimensiones con información numérica, alfanumérica o lógica. Para construir un arreglo se usa la función `array( )`. Por ejemplo, para crear un arreglo de 3×4×2 con las primeras 24 letras minúsculas del alfabeto se escribe el siguiente código.

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

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "d"  "g"  "j"
## [2,] "b"  "e"  "h"  "k"
## [3,] "c"  "f"  "i"  "l"
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,] "m"  "p"  "s"  "v"
## [2,] "n"  "q"  "t"  "w"
## [3,] "o"  "r"  "u"  "x"
```

El argumento `data` de la función sirve para indicar los datos que se van a almacenar en el arreglo y el argumento `dim` sirve para indicar las dimensiones del arreglo.

Para obtener elementos almacenados en un arreglo se usan también corchetes, y dentro de los corchetes, las coordenadas del objeto de interés. Ejemplo:

Si queremos extraer la letra almacenada en la fila 1 y columna 3 de la segunda capa de miarray usamos el siguiente código.

```
miarray[1, 3, 2]  # Notar que el orden es importante
```

```
## [1] "s"
```

Si queremos extraer la segunda capa completa usamos el siguiente código.

```
miarray[,, 2]  # Notar que no se coloca nada en las primeras posiciones
```

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

Si queremos extraer la tercera columna de todas las capas usamos el siguiente código.

```
miarray[, 3,]  # No se coloca nada en las primeras posiciones
```

```
##      [,1] [,2]
## [1,] "g"  "s"
## [2,] "h"  "t"
## [3,] "i"  "u"
```