Capítulo 1

Los datos y sus tipos

Todas las cosas que manipula R se llaman objetos. En general, éstos se construyen a partir de objetos más simples. De esta manera, se llega a los objetos más simples que son de cinco clases a las que se denomina *atómicas* y que son las siguientes:

- `character` (cadenas de caracteres)
- `numeric` (números reales)
- `integer` (números enteros)
- `complex` (números complejos)
- `logical` (lógicos o booleanos, que sólo toman los valores True o False)

En el lenguaje, sin embargo, cada uno de estas clases de datos no se encuentran ni se manejan de manera aislada, sino encapsulados dentro de la clase de objeto más básica del lenguaje: el `vector`. Un `vector` puede contener cero o más objetos, pero todos de la misma clase. En contraste, la clase denominada `list`, permite componer objetos también como una secuencia de otros objetos, pero, a diferencia del `vector`, cada uno de sus componentes puede ser de una clase distinta.

1.1. Los datos numéricos

Probablemente el principal uso de R es la manipulación de datos numéricos. El lenguaje agrupa estos datos en tres categorías, a saber: `numeric, integer` y `complex`, pero cuando se introduce algo que puede interpretar como un número, su inclinación es tratarlo como un dato de tipo `numeric`, es decir, un número de tipo real, a no ser que explícitamente se indique otra cosa. Veamos algunos ejemplos:

```
x <- 2  # Se asigna el valor 2 a x
print(x)  # Se imprime el valor de x

## [1] 2

class(x)  # Muestra cual es la clase de x

## [1] "numeric"

x <- 6/2  # Se asigna el valor de la operacion dividir 6/2 a x
print(x)

## [1] 3

class(x)

## [1] "numeric"
```

Aparentemente las dos asignaciones que se hacen, mediante el operador de asignación, `<-`, a la *variable* `x`, es de los enteros 2 y 3 respectivamente. Sin embargo, al preguntar, mediante la *función* `class()`, cuál es la clase de `x`, la respuesta es `numeric`, esto es, un número real. Para asignar explícitamente un entero, `integer`, a una variable, se agrega la letra L al final del número, como sigue:

```
x <- 23L; print(x)

## [1] 23

class(x)

## [1] "integer"
```

Aquí la variable `x` tendrá como valor el entero 23. Como una nota adicional del lenguaje, nótese que se han escrito dos expresiones de R en un mismo renglón. En este caso, las expresiones se separan mediante ';'.

Para lograr que una expresión, como la operación de división `6/2`, arroje como resultado un entero, se tiene que hacer una *conversión*; ello se logra mediante la función `as.integer`, como sigue:

```
x <- as.integer(6/2); print(x)

## [1] 3

class(x)

## [1] "integer"
```

Por su parte, los números complejos, `complex` en el lenguaje, tienen una sintaxis muy particular; misma que se tiene que emplear para indicar explícitamente que un número introducido corresponde a ese tipo:

```
x <- 21 + 2
y <- 2 + 21 # El mismo valor que x
z <- -1 + 0 # Corresponde a -1
tt <- sqrt(z) # raiz cuadrada de -1
print(x); print(y); print(z); print(tt)

## [1] 21+2i
## [1] 21+2i
## [1] -1+0i
## [1] 0+1i

class(tt)

## [1] "complex"
```

En los ejemplos anteriores a la variable `tt` se le asigna el resultado de una función, `sqrt()`, que es la raíz cuadrada de el número `-1`. Nótese que ésta es la forma correcta de calcular esa raíz, por ejemplo, `sqrt(-1)`, hubiera arrojado como resultado un error.

También, existe un valor numérico especial, `Inf`, que representa el infinito y que puede resultar en algunas expresiones, por ejemplo:

```
x <- 1/0  # Division por cero
x

## [1] Inf

# Tambien dividir un número por Inf da cero:
y <- 1/Inf
y

## [1] 0
```

Finalmente, algunas operaciones pueden resultar en algo que no es un número, esto se representa por el valor `NaN`. Veamos un ejemplo:

```
x <- 0/0
x

## [1] NaN
```

1.2. Vectores

Se ha dicho con anterioridad que las clases atómicas de datos no se manejan de manera individual. En efecto, en todos los ejemplos anteriores, el lenguaje ha creado implícitamente vectores de longitud 1, y son esos los que se han asignado a las variables. Tomemos el caso más sencillo:

```
x <- 2  # Se asigna el valor 2 a x
print(x)  # Se imprime el valor de x

## [1] 2
```

Aquí, la impresión del valor de `x` tiene una forma muy particular: "`[1] 2`". El `'[1]'` que precede al valor, indica que se trata del primer elemento y único, en este caso, del vector que se muestra.

Hay diversas maneras de crear vectores de otras longitudes, que, como se ha dicho antes, son secuencias de objetos de la misma clase atómica. En las siguientes secciones se verán algunos casos.

1.2.1. El uso de la función `c()` para crear vectores

La primer manera de crear vectores es a partir de los elementos individuales que compondrán el vector. Para esto se utiliza la función `c()` como se muestra a continuación.

```
c(4,2,-8)  # Creacion de un vector sin asignarlo a una variable

## [1]  4  2 -8

## ----------------
## Distintas formas de asignar un vector a una variable
u <- c(4,2,-8) # Usando el operador <-
c(4, 2, -8) -> v # Usando el operador ->
# Usando la funcion assign:
assign("w", c(4, 2, -8))
p = c(4, 2, -8) # Usando el operador =
print(u); print(v); print(w); print(p)

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

La función `c()` sirve para concatenar varios elementos del mismo tipo. En todos los ejemplos mostrados, la impresión del vector se hace en un renglón que comienza con el símbolo `'[1]'`, indicando con ello que el primer elemento del renglón corresponde al primer elemento del vector.

Un caso muy particular de asignación, es el de la función `assign()`. A diferencia de los otros casos vistos en el ejemplo anterior, el nombre de la variable aparece entre comillas.

Más adelante, en la página 7, se verá como la función `c()` también se puede utilizar para la creación de vectores a partir de otros vectores.

1.2.2. Creación de vectores a partir de archivos de texto - la función `scan()`

Otra manera de crear un vector es a partir de un archivo de texto. Sea, por ejemplo, el caso del archivo UnVec.txt, que se contiene la siguiente información:

```
  12 15.5 3.1
  -2.2 0 0.0007
```

Supóngase ahora que a partir de esos datos se quiere crear un vector. Para eso se usa la función `scan()`, como se muestra a continuación:

```
vec <- scan("UnVec.txt")
print(vec)

## [1] 12.0000 15.5000  3.1000 -2.2000  0.0000  0.0007
```

Desde luego que hay otras funciones para lectura de archivos, más complejas, pero baste por el momento con el uso de esta función, tal como se muestra, para permitir la creación de un vector a partir de los datos contenidos en un archivo de texto. Por el ahora, la única nota adicional es que la función `scan()` ofrece la posibilidad de indicarle explícitamente el tipo de vector que se quiere crear. Así por ejemplo, la creación de un vector de enteros se hace de la siguiente manera:

```
vec <- scan("IntVec.txt", integer())
print(vec); class(vec) # El vector y su clase

## [1]   4   3  -2   1   0 200  -8  20
## [1] "integer"
```

Por supuesto que en este caso, se debe prever que el archivo leído contenga datos que puedan ser interpretados como números enteros.

La función inversa, en este caso, de la función scan(), es la función write. Así, un vector cualquiera fácilmente se puede escribir en un archivo de texto, como se muestra a continuación:

```
vv <- c(5, 6.6, -7.7)
write(vv, "OtroArchivo.txt")
# Ahora recuperemos el contenido del archivo
v1 <- scan("OtroArchivo.txt")
v1
```

```
## [1]  5.0  6.6 -7.7
```

1.2.3. Creación de vectores a partir de secuencias y otros patrones

Un vector, inicializado en ceros, o `FALSE`, y de longitud determinada, se puede crear con la función `vector()`. Es esta misma función la que permite crear vectores sin elementos. En seguida se muestran algunos ejemplos:

```
v <- vector("integer", 0)
v  # Un vector de enteros sin elementos

## integer(0)

w <- vector("numeric", 3)
w  # Un vector de tres ceros

## [1] 0 0 0

u <- vector("logical", 5)
u  # Un vector de 5 FALSE

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

El operador `':'` permite generar un vector entero a partir de una secuencia creciente o decreciente de enteros, cuyos extremos se indican, tal como se muestra en seguida:

```
1:3

## [1] 1 2 3

v <- 40:13
print(v)

##  [1] 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23
## [19] 22 21 20 19 18 17 16 15 14 13

class(v)  # El vector y su clase

## [1] "integer"
```

Nótese que el desplegado o impresión del vector `v`, se ha tenido que hacer en dos renglones. Cada uno de esos renglones comienza, indicando entre corchetes `[ ]`, el índice del primer elemento en el renglón. Por otra parte, el operador `':'` es un caso particular de la función `seq()` que permite generar una mayor variedad de secuencias numéricas. Veamos aquí algunos ejemplos:

```
v <- seq(from = 5, to = 15, by = 2)
print(v)  # secuencia desde 5 hasta 15 de 2 en 2

## [1]  5  7  9 11 13 15
```

Debe notarse aquí, no obstante, que la clase del resultado de esta secuencia es `numeric` y no `integer`; esto es, el vector resultante es de números reales, que puede, a conveniencia, ser convertido a enteros, mediante la función `as.integer()`, como se vio anteriormente, en la página 2.

```
class(v)

## [1] "numeric"
```

La función `seq()` tiene varios argumentos más cuya documentación se puede consultar mediante `?seq` o `help('seq')` en el intérprete de R. En seguida se muestra sólo otra forma bastante común de utilizar esta función, que tiene que ver con la producción de un vector o una secuencia de una longitud determinada.

```
v <- seq(from = 4, by = 2, length.out = 8)
print(v)  # secuencia de 8 números iniciando desde  4 y de 2 en 2

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

Algunas veces es necesario repetir una secuencia de números varias veces para generar un vector deseado. La función `rep()` sirve para ese propósito. Supóngase, por ejemplo, que se desea crear un vector con la repetición de la secuencia 4, 8, -3, cinco veces. Eso se logra como se muestra a continuación:

```
v <- c(4, 8, -3)
w <- rep(v, times = 5)
print(w)

##  [1]  4  8 -3  4  8 -3  4  8 -3  4  8 -3  4  8 -3
```

Finalmente, aveces se requiere construir un vector a partir de dos o más vectores ya existentes. La forma simple de lograr esto es con la función `c()` como se muestra a continuación:

```
u <- c(3, 4, 5)
v <- c(5, 4, 3)
w <- c(u, v)
print(w)  # La concatenacion de u y v

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

1.2.4. Acceso a los elementos individuales de un vector

Aunque este tema está comprendido dentro de la selección de subconjuntos o porciones, que se verá más adelante en el capítulo **??**, se dará aquí un adelanto para permitir operar con los elementos individuales de los vectores. Dentro de un vector, sus elementos se pueden identificar mediante un *índice* entero, que en el caso de este lenguaje empieza con el 1. Así, por ejemplo:

```
v <- c(8, 7, -3, 2, 182)
v[5] # El quinto elemento

## [1] 182

print(v[1]); print(v[3])

## [1] 8
## [1] -3

v[4]+v[2] # La suma del cuarto y segundo elementos de v

## [1] 9
```

Primeramente se ha accedido al quinto elemento, mediante `v[5]` ; si el intérprete de R se está usando interactivamente, el valor de ese elemento, 182 , se imprime implícitamente. Luego se manda imprimir explícitamente, los elementos 1 y 3 del vector. Finalmente, se suman los elementos 4 y 2 del vector; el resultado de esa operación se imprime implícitamente, es decir, de manera automática, si la operación se solicita al intérprete en su modo interactivo.

El acceso a los elementos individuales de un vector no solamente es para *consulta* o *lectura*, sino también para su *modificación* o *escritura*. Por ejemplo:

```
v[1] <- v[2] - v[5]
v  # Note que el resultado de la operacion se ha guardado en v[1]

## [1] -175    7   -3    2  182
```

Esta misma operación puede hacer crecer un vector. Por ejemplo, el vector `v` tiene 5 elementos. Si se asigna un valor al elemento 8, el vector *crecerá* hasta esa longitud, de la manera siguiente:

```
v[8] <- 213
v  # v tiene ahora 8 elementos con espacios vacios: NA

## [1] -175    7   -3    2  182   NA   NA  213
```

La nota aquí es que para aumentar el vector a esa longitud se tuvieron que introducir elementos ausentes o vacíos que se indican con el valor `NA` (del inglés: *Not Available*) en los espacios correspondientes.

Otra característica interesante de este lenguaje, es que permite dar nombre y acceder por medio de ese nombre a los elementos individuales de un vector. Supóngase por ejemplo que se tiene el registro de cantidades de ciertas frutas en un vector:

```
frutas <- c(15, 100, 2, 30)
frutas

## [1]  15 100   2  30
```

Supóngase ahora que se quiere asociar esos valores con el nombre de la fruta correspondiente:

```
names(frutas) <- c("naranja", "pera", "manzana", "durazno")
```

Si ahora se manda desplegar el vector:

```
frutas

## naranja    pera manzana durazno
##      15     100       2      30
```

Otra manera más directa de nombrar los elementos de un vector, es en el momento mismo de la creación con la función `c()`, con una sintaxis semejante a la siguiente:

```
frutas <- c(naranja = 15, pera = 100, manzana = 2, durazno = 30)
```

Además se puede acceder a los elementos individuales del vector mediante su nombre:

```
frutas["durazno"]

## durazno
##      30

frutas["manzana"] <- 8
frutas

## naranja    pera manzana durazno
##      15     100       8      30

# El acceso a traves de indices se sigue permitiendo:
frutas[2]

## pera
##  100
```

1.2.5. Operaciones sencillas con vectores

Las operaciones aritméticas más comunes están definidas para vectores: la suma, la resta, la división y la exponenciación, todas ellas se definen elemento a elemento entre dos vectores. Por ejemplo:

```
v <- 2 + 3  # Resulta en un vector de longitud 1
v

## [1] 5

v <- c(2, 3) - c(5, 1)  # Resulta en un vector de longitud 2
v

## [1] -3  2

v <- c(2, 3, 4) * c(2, 1, 3)  # Resulta en un vector de longitud 3
v

## [1]  4  3 12

v <- c(2, 3, 4)^(3:1)  # Eleva a potencias 3,2,1
v

## [1] 8 9 4
```

En todos los casos, la operación indicada se aplica elemento a elemento entre los dos vectores operandos. En el último ejemplo, debido al orden de precedencia de aplicación de los operadores, es necesario encerrar entre paréntesis la expresión `3:1`.

En muchas ocasiones es necesario saber la longitud de una vector. La función `length()` aplicada a un vector regresa precisamente ese valor:

```
u <- 2:33
v <- c(4, 5, 6)
w <- c(u, v)
w

##  [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  4  5  6

length(w)

## [1] 35
```

Aprovecharemos el vector `w`, creado en el ejemplo anterior, para ilustrar también el uso de las operaciones lógicas. ¿Qué pasa si probamos este vector para saber cuáles de sus elementos son menores o iguales a 10?

```
w <= 10  # Prueba elementos menores o iguales a 10

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [10] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [28] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

El resultado es un vector de lógicos, de la misma longitud que el original y *paralelo* a ese, en el que se indica, elemento a elemento cuál es el resultado de la prueba lógica: *"menor o igual que diez"*, en este caso. Otros operadores lógicos son: `<`, `>`, `>=`, `==`, y `!=`.

En el asunto de las operaciones aritméticas que se han ilustrado anteriormente, surge una pregunta: ¿qué pasa cuando los vectores operandos no son de la misma longitud? En esos casos, el intérprete del lenguaje procede a completar la operación *reciclando* los elementos del operador de menor longitud. Así, por ejemplo:

```
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2)

## Warning: longitud de objeto mayor no es múltiplo de la longitud
de uno menor

v

## [1]  4 10  6 14  8 18 10
```

es lo mismo que:

```
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2, 1, 2, 1, 2, 1)
v

## [1]  4 10  6 14  8 18 10
```

Notemos, sin embargo, que en el primer caso el sistema ha arrojado un mensaje de advertencia, *Warning*, indicando la diferencia en las longitudes de los operandos. La eliminación de estos mensajes se hace por medio de la función `options()`, como sigue:

```
options(warn = -1)
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2)
v

## [1]  4 10  6 14  8 18 10
```

Es esta funcionalidad la que permite hacer de manera muy simple algunas operaciones vectoriales, como por ejemplo:

```
v <- c(2, -3, 4)
w <- 2 * (v^2)  # Dos veces el cuadrado de v
w

## [1]  8 18 32
```

Además, algunas funciones pueden recibir como argumento un vector y producir a su salida un vector de la misma longitud que el de entrada. Tal es el caso de las funciones trigonométricas como `sin()`, `cos()`, y la raíz cuadrada: `sqrt()`. Por ejemplo:

```
# Se desea la raiz cuadrada de los siguientes valores:
v <- c(9, 8, 31)
sqrt(v)

## [1] 3.000 2.828 5.568

# El sin de 30, 45 y 60 grados: Primero se hace la conversion
# a radianes:
angulos <- c(30, 45, 60) * (pi/180)
angulos  # En radianes

## [1] 0.5236 0.7854 1.0472

senos <- sin(angulos)
senos

## [1] 0.5000 0.7071 0.8660
```

Para ilustrar la utilidad de estos conceptos en los siguientes párrafos se da un ejemplo de aplicación.

Ejemplo de aplicación

De un edificio, a una altura de 15 m, se ha lanzado con un ángulo de 50 grados, un proyectil a una velocidad de 7 m/s. ¿Cuáles serán las alturas (coordenadas y) del proyectil a cada 0.5 m de distancia horizontal desde donde se lanzó y hasta los 11 m?

Las ecuaciones que gobiernan este fenómeno son las siguientes:

$$x = v_{0x}t + x_0$$

$$y = -\tfrac{1}{2}gt^2 + v_{0y}t + y_0$$

Aquí, g es la aceleración de la gravedad, el parámetro t se refiere al tiempo, y la velocidad está descompuesta en sus componentes: v_{0x} y v_{0y}. Tal como se muestra en la Fig. 1.1, éstas se pueden obtener a partir de la velocidad inicial y el ángulo, usando las funciones trigonométricas `sin()` y `cos()`, y considerando

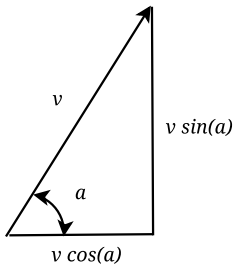

Figura 1.1: Las componentes de la velocidad

que en R, los argumentos de esas funciones deben estar dados en radianes, y por tanto el ángulo debe convertirse a esa unidad. Así, los datos de partida son como sigue:

```
g <- 9.81  # aceleracion gravedad
x0 <- 0  # x inicial
y0 <- 15  # y inicial
vi <- 7  # velocidad inicial
alphaD <- 50  # angulo-grados
```

y para encontrar las componentes de la velocidad:

```
# Se convierte a radianes
alpha <- (pi/180) * alphaD  # angulo-radianes
vox <- vi * cos(alpha)  # componente x de velocidad inicial
vox

## [1] 4.5

voy <- vi * sin(alpha)  # componente y de velocidad inicial
voy

## [1] 5.362
```

Con esto es suficiente para proceder con el problema. Primeramente obtenemos las x para las que se desea hacer el cálculo, como sigue:

```
# desde 0 hasta 11 de 0.5 en 0.5:
las.x <- seq(from = 0, to = 11, by = 0.5)
```

En este ejemplo, la secuencia de valores de x se ha guardado en una variable de nombre "`las.x`". En este lenguaje, en los nombres de las variables, el punto (.), así como el guión bajo (_), se pueden utilizar simplemente como separadores, para darles mayor claridad.

Nótese que en las fórmulas que gobiernan el fenómeno, dadas anteriormente, no se tiene y en función de x, sino que las dos coordenadas dependen del parámetro t, esto es, del tiempo. Para resolver este asunto simplemente se despeja en parámetro t, en la ecuación de x, y obtenemos:

$$t = (x - x_0)/v_{0x}$$

Así, obtenemos los valores de t correspondientes a las x, usando esta fórmula:

```
las.t <- (las.x - x0)/vox
```

Finalmente, encontramos las y correspondientes a las t, justamente encontradas, aplicando la fórmula para y:

```
las.y <- -(g/2) * las.t^2 + voy * las.t + y0
# Los resultados:
las.x

##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
## [12]  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5
## [23] 11.0

las.y

##  [1] 15.0000 15.5353 15.9495 16.2425 16.4144 16.4652 16.3948
##  [8] 16.2033 15.8906 15.4568 14.9019 14.2258 13.4286 12.5103
## [15] 11.4708 10.3102  9.0285  7.6256  6.1015  4.4564  2.6901
## [22]  0.8026 -1.2059
```

Se han encontrado los valores buscados, y en la Fig. 1.2 se muestra un gráfico con la trayectoria del proyectil.

1.2.6. Otras clases de datos basadas en vectores

Los vectores sirven como base para la definición de otras clases de datos, a saber: las matrices y los arreglos. En la sección siguiente se da una breve introducción al tema de las matrices. El tema de los arreglos, sin embargo, se abordará en un capítulo posterior.

1.3. Matrices

Desde el punto de vista del lenguaje, una matriz es un vector con un atributo adicional: `dim`. Para el caso de las matrices, este atributo es un vector entero de dos elementos, a saber: el número de renglones y el número de columnas que componen a la matriz.

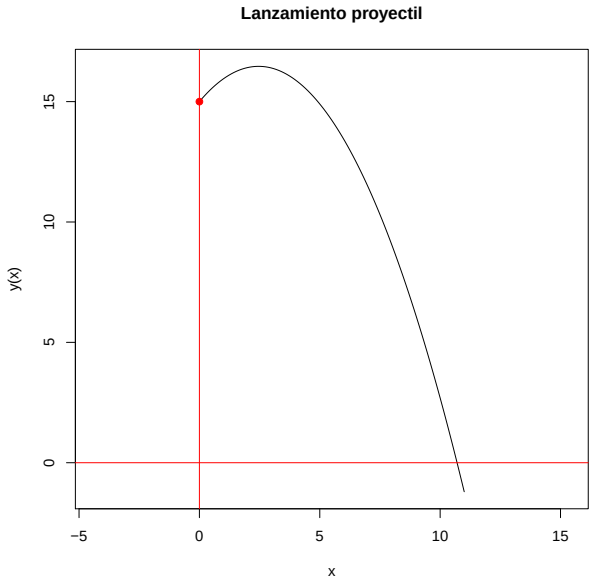

Figura 1.2: Gráfico de la trayectoria del proyectil lanzado desde una altura de 15 m.

1.3.1. Construcción de matrices

Una de las formas de construir una matriz es a partir de un vector, como sigue:

```
(m <- 11:30)  # Un vector con 20 numeros
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
## [19] 29 30
```

```
# Para convertirla en matriz simplemente se especifica el
# atributo dim
dim(m) <- c(4, 5)  # 4 renglones y 5 columnas
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   15   19   23   27
## [2,]   12   16   20   24   28
## [3,]   13   17   21   25   29
## [4,]   14   18   22   26   30
```

```
class(m)
```

```
## [1] "matrix"
```

Debe notarse que, mediante la construcción mostrada, el armado de la matriz se hace por columnas. Por otra parte, las dimensiones de la matriz pueden cambiarse en cualquier momento, y el acceso a un elemento particular de la matriz se hace ahora mediante dos índices: el renglón y la columna, aunque, el acceso a los elementos de la matriz como un vector, es decir, con un solo índice, sigue siendo posible, como se muestra en seguida:

```
dim(m) <- c(5, 4)  # ahora 5 renglones y 4 columnas
m

##      [,1] [,2] [,3] [,4]
## [1,]   11   16   21   26
## [2,]   12   17   22   27
## [3,]   13   18   23   28
## [4,]   14   19   24   29
## [5,]   15   20   25   30

# Y el elemento en el renglon 3 y columna 2 es:
m[3, 2]

## [1] 18

m[8]  # acceso al mismo elemento, como vector, con un solo indice

## [1] 18
```

Una ventaja del lenguaje es que permite hacer referencia a una columna o a un renglón de la matriz, como si se tratara de un sólo objeto, o sea como un vector. Para ello, se omite alguno de los dos índices en la expresión de acceso a la matriz, como se muestra más adelante. En el ejemplo que se viene examinando, esos vectores estarían compuestos por números enteros, aunque los componentes de una matriz pueden ser también reales (`numeric`) o complejos (`complex`).

```
# El renglon 3 y la columna 2 de la matriz:
m[3, ]

## [1] 13 18 23 28

m[, 2]

## [1] 16 17 18 19 20

# La clase las columnas o renglones:
class(m[3, ])

## [1] "integer"
```

Las matrices también se pueden crear de manera flexible por medio de la función primitiva `matrix()`, que permite alterar la secuencia por *default* de armado de la matriz; esto es, ahora, si se quiere, se puede armar la matriz por renglones en vez de columnas:

```
(m <- matrix(11:30, nrow = 5, ncol = 4, byrow = TRUE))

##      [,1] [,2] [,3] [,4]
## [1,]   11   12   13   14
## [2,]   15   16   17   18
## [3,]   19   20   21   22
## [4,]   23   24   25   26
## [5,]   27   28   29   30
```

Adicionalmente, a los renglones y las columnas de una matriz se les pueden asignar nombres, que pueden ser después consultados o usados como índices:

```
rownames(m) <- c("uno", "dos", "tres", "cuatro", "cinco")
colnames(m) <- c("UNO", "DOS", "TRES", "CUATRO")
m

##        UNO DOS TRES CUATRO
## uno     11  12   13     14
## dos     15  16   17     18
## tres    19  20   21     22
## cuatro  23  24   25     26
## cinco   27  28   29     30

# Consulta de los nombres de las columnas
colnames(m)

## [1] "UNO"    "DOS"    "TRES"   "CUATRO"

# Una columna:
m[, "DOS"]

##    uno    dos   tres cuatro  cinco
##     12     16     20     24     28
```

Las funciones `rbind()` y `cbind()`, son otras que se pueden utilizar para construir matrices, dando, ya sea los renglones individuales o las columnas individuales, respectivamente.

```
m1 <- rbind(c(1.5, 3.2, -5.5), c(0, -1.1, 60))
m1

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

```
## [1,]  1.5  3.2 -5.5
## [2,]  0.0 -1.1 60.0

class(m1[1, ])  # ahora compuesta de numeros reales

## [1] "numeric"

(m2 <- cbind(c(1.5, 3.2, -5.5), c(0, -1.1, 60)))

##      [,1] [,2]
## [1,]  1.5  0.0
## [2,]  3.2 -1.1
## [3,] -5.5 60.0
```

1.3.2. Acceso a los elementos individuales de una matriz

Como en los casos anteriores, el lenguaje también provee de mecanismos para acceder a los elementos individuales de una matriz. Para ello se emplea el operador `[]`. Supongamos que en la matriz `m`, del ejemplo anterior se quiere tener acceso al elemento que se encuentra en el renglón 2 y en la columna 1 de la matriz. Eso se logra de la siguiente manera:

```
m[2, 1]

## [1] 15
```

Y también se pueden utilizar los nombres de renglón y columna, si es que la matriz los tiene:

```
m["dos", "UNO"]

## [1] 15
```

Otras formas para tener acceso a porciones de la matriz, se verán con detalle más adelante, en el capítulo **??**.

1.3.3. Operaciones sencillas con matrices

Todas las operaciones aritméticas válidas para vectores, son validas para las matrices, siempre y cuando, las matrices operando tengan las mismas dimensiones y se aplican elemento a elemento, esto es, la operación se aplica entre cada columna, con su correspondiente, como si fueran vectores. (véase la sección correspondiente: 1.2.5). En seguida, se muestra un ejemplo con la multiplicación, que no debe ser confundido con la multiplicación matricial.

```
(m <- matrix(1:15, nrow = 5, ncol = 3))

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

(mm <- rbind(1:3, 3:1, c(1, 1, 1), c(2, 2, 2), c(3, 3, 3)))

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

m * mm

##      [,1] [,2] [,3]
## [1,]    1   12   33
## [2,]    6   14   12
## [3,]    3    8   13
## [4,]    8   18   28
## [5,]   15   30   45
```

La multiplicación matricial se hace con el operador `%*%`. Para entender esta operación, pondremos un ejemplo con dos matrices, como sigue:

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

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

(B <- rbind(7:9, 10:12))

##      [,1] [,2] [,3]
## [1,]    7    8    9
## [2,]   10   11   12
```

En el ejemplo, la matriz `A` será multiplicada por la matriz `B`, y debe notarse que, en este caso, el número de columnas de la matriz `A`, es igual al número de renglones de la matriz `B`. La multiplicación de estas dos matrices la podemos visualizar en la Fig. 1.3. En esta figura, la matriz `A` se pone a la izquierda y

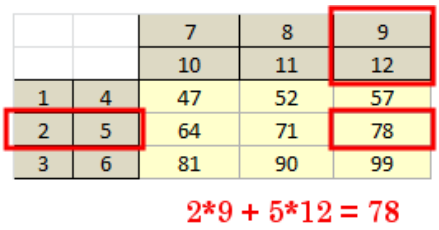

Figura 1.3: La multiplicación matricial

la matriz `B` se pone en la parte superior. Los elementos de la matriz producto, estarán en las intersecciones de un renglón de la matriz `A` con una columna de la matriz `B`, y se calculan como se muestra en el ejemplo: el primer elemento del renglón de `A` por el primer elemento de la columna de `B` más el segundo elemento del renglón de `A` por el segundo elemento de la columna de `B`, etc. Este procedimiento es igual, para dimensiones mayores, siempre y cuando coincida el número de columnas de `A` con el número de renglones de `B`. En R, esta operación se hace así:

```
A %*% B

##      [,1] [,2] [,3]
## [1,]   47   52   57
## [2,]   64   71   78
## [3,]   81   90   99
```

Otra operación muy utilizada e implementada en R como una función, `t()`, es la traspuesta de una matriz. Esta es una operación en la que los renglones se cambian a columnas y viceversa, tal como se muestra en el siguiente ejemplo:

```
# Se usa la misma matriz A del ejemplo anterior:
A

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

t(A)

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

Hay otras operaciones matriciales, pero baste con éstas en el presente capítulo, que las otras se introducirán más adelante en el texto.

Ejemplo de aplicación

Las transformaciones lineales se representan por medio de matrices, y su aplicación involucra la multiplicación matricial de la matriz que representa la transformación por el vector o secuencia de vectores que representan el punto o puntos en el espacio que se quieren transformar. Como un ejemplo, la rotación en dos dimensiones es una transformación lineal: si se quiere rotar el punto (x, y) por un ángulo α, la operación está dada por:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

donde el punto (x', y'), es el punto transformado, es decir, al que se ha aplicado la rotación. Si la operación se quiere hacer a una secuencia de puntos que pudieran representar los vértices de alguna figura geométrica, bastará con armar la matriz de puntos correspondiente y aplicar a ella la transformación, de la siguiente manera:

$$\begin{bmatrix} x'_1 & x'_2 & \dots & x'_n \\ y'_1 & y'_2 & \dots & y'_n \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \end{bmatrix}$$

Supóngase ahora, que se tiene un triángulo, cuyos vértices son (1.0, 0.0), (2.0, 1.0), y (1.0, 1.0), y se quieren encontrar los vértices del triángulo resultante de una rotación de 32^{o}. Tómese en cuenta que el lenguaje R, provee de las funciones trigonométricas `sin()`, `cos()`, así como del número `pi`.

```
# Triangulo original:
m <- cbind(c(1, 0), c(2, 1), c(1, 1))
# Se convierte el angulo a radianes
alpha <- 32 * pi/180
# La matriz para esa rotacion es:
tr <- rbind(c(cos(alpha), -sin(alpha)), c(sin(alpha), cos(alpha)))
# El triangulo transformado
mt <- tr %*% m  # multiplicacion matricial
# Los vertices del triangulo transformado
mt
```

```
##         [,1]  [,2]   [,3]
## [1,] 0.8480 1.166 0.3181
## [2,] 0.5299 1.908 1.3780
```

En la Fig. 1.4 se muestran tanto el triangulo original, como el triangulo resultante, en rojo, después de la rotación.

1.4. Factores y vectores de caracteres

Los caracteres, o más apropiadamente, las *cadenas de caracteres*, se utilizan para nombrar cosas u objetos del *mundo*. Igual que en el caso de los números, en

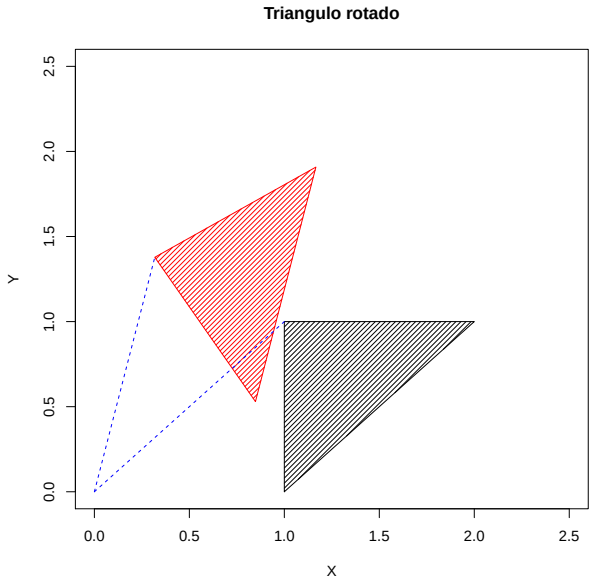

Figura 1.4: Rotación de un triángulo; el triángulo rotado se muestra en rojo

R la clase `character` no se refiere a una cadena de caracteres aislada sino a un vector que contiene cero o más cadenas de caracteres. De este modo podríamos tener por ejemplo, una lista (o vector) con los nombres de personas, y otra, paralela a la primera, con sus meses de nacimiento:

```
persona <- c("Hugo", "Paco", "Luis", "Petra", "Maria", "Fulano",
             "Sutano", "Perengano", "Metano", "Etano", "Propano")
mes.nacimiento <- c("Dic", "Feb", "Oct", "Mar", "Feb", "Nov",
                    "Abr", "Dic", "Feb", "Oct", "Dic")
persona

##  [1] "Hugo"      "Paco"      "Luis"      "Petra"
##  [5] "Maria"     "Fulano"    "Sutano"    "Perengano"
##  [9] "Metano"    "Etano"     "Propano"

mes.nacimiento

##  [1] "Dic" "Feb" "Oct" "Mar" "Feb" "Nov" "Abr" "Dic" "Feb"
## [10] "Oct" "Dic"
```

Así, si se quiere imprimir el nombre de la persona 7 con su mes de nacimiento se puede hacer con:

```
print(persona[7]); print(mes.nacimiento[7])

## [1] "Sutano"
## [1] "Abr"

# De una manera mas "pulcra":
print( c(persona[7], mes.nacimiento[7]) )

## [1] "Sutano" "Abr"
```

La función `paste()` permite concatenar cadenas de caracteres y por medio de ella se puede dar incluso una mejor apariencia a la salida:

```
paste(persona[7], "nacio en el mes de", mes.nacimiento[7])

## [1] "Sutano nacio en el mes de Abr"
```

1.4.1. Los factores y su estructura

Los dos vectores anteriores pueden considerarse como una *estructura de información*, a la que se puede someter a algún tipo de procesamiento estadístico. El lenguaje tiene muchas herramientas para ese propósito. Considérese, por ejemplo, el problema de determinar la frecuencia de aparición de ciertos meses en el vector `mes.nacimiento`. En este caso, el lenguaje provee de una clase que facilita este tipo de análisis, a saber: la clase `factor`. Para entender esta clase, procedamos primeramente a transformar el vector `mes.nacimiento` a un factor, mediante la función de conversión `as.factor()`, como sigue:

```
Fmes.nacimiento <- as.factor(mes.nacimiento)
Fmes.nacimiento

##  [1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
## Levels: Abr Dic Feb Mar Nov Oct

# y generamos la impresion ahora con el factor:
paste(persona[7], "nacio en el mes de", Fmes.nacimiento[7])

## [1] "Sutano nacio en el mes de Abr"
```

Si se compara la impresión del factor `Fmes.nacimiento` con la del vector `mes.nacimiento`, se podría pensar que "no ha pasado mucho". De hecho, la impresión *bonita* con la función `paste()`, ha resultado igual. Sin embargo, el factor exhibe una estructura adicional denominada `Levels`, en la que se han registrado e identificado los elementos del vector sin repetición; esto es, los nombres únicos de los meses, en este caso. La estructura interna de esta clase se puede descubrir:

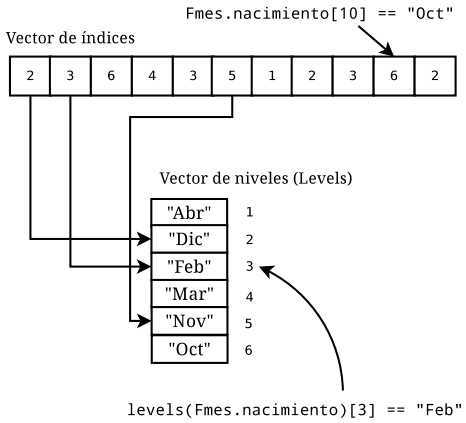

Figura 1.5: Estructura interna de los factores

```
unclass(Fmes.nacimiento)

##  [1] 2 3 6 4 3 5 1 2 3 6 2
## attr(,"levels")
## [1] "Abr" "Dic" "Feb" "Mar" "Nov" "Oct"
```

Como se puede ver, el núcleo de la clase son dos vectores. El primero, es un vector de índices enteros, que sustituye al vector de caracteres original, y el segundo es un vector de caracteres, que contiene los niveles (*Levels*) o categorías, a los que hace referencia el primer vector. La Fig. 1.5 muestra esta disposición, en la que, con motivo de no tener un desplegado confuso, se grafican sólo tres de las referencias del vector de índices al vector de niveles.

Abordemos ahora el problema que motivó la presente discusión: la frecuencia de aparición de ciertos elementos en un vector. La función `table()` toma típicamente como argumento un factor y regresa como resultado justamente la frecuencia de aparición de los niveles en el vector de índices:

```
table(Fmes.nacimiento)

## Fmes.nacimiento
## Abr Dic Feb Mar Nov Oct
##   1   3   3   1   1   2
```

La interpretación de estos resultados en el contexto de la estructura de información original, es que, por ejemplo, 3 personas del vector `persona`, nacieron en el mes de Dic. En el ejemplo mostrado, los niveles o `Levels` aparecen ordenados alfabéticamente. La creación de factores en los que se establezca un orden determinado en los niveles, se puede hacer con la función `factor()`, como se muestra:

```
meses <- c("Ene","Feb","Mar","Abr","May","Jun","Jul","Ago",
           "Sep","Oct","Nov","Dic")
# Se incluyen meses que no estan el el vector original
FFmes.nacimiento <- factor(mes.nacimiento, levels=meses)
FFmes.nacimiento

##  [1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic

# Ahora la tabla de frecuencias es:
table(FFmes.nacimiento)

## FFmes.nacimiento
## Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
##   0   3   1   1   0   0   0   0   0   2   1   3
```

Debe notarse que la función `table()` pudiera haber recibido como argumento directamente el vector de caracteres original, y hubiera producido el resultado deseado, como se muestra:

```
table(mes.nacimiento)

## mes.nacimiento
## Abr Dic Feb Mar Nov Oct
##   1   3   3   1   1   2
```

La razón es simple: el intérprete del lenguaje *sabe* que la función está esperando recibir un factor y en consecuencia trata de convertir, en automático, el argumento que recibe, a esa clase. Como la conversión de vectores de caracteres a factores es trivial, la función no tiene ningún problema en desarrollar su tarea.

1.4.2. Acceso a los elementos de un factor

El acceso a cada uno de los dos vectores que le dan estructura al factor se hace como se muestra a continuación y se ha ilustrado también en la Fig. 1.5:

```
# Un elemento individual del factor:
Fmes.nacimiento[10]

## [1] Oct
## Levels: Abr Dic Feb Mar Nov Oct

# Un elemento individual de los niveles:
levels(Fmes.nacimiento)[3]

## [1] "Feb"
```

Incluso es posible modificar todos o algunos de los niveles del factor. Por ejemplo:

```
levels(Fmes.nacimiento)[3] <- "febrero"
Fmes.nacimiento

##  [1] Dic     febrero Oct     Mar     febrero Nov     Abr
##  [8] Dic     febrero Oct     Dic
## Levels: Abr Dic febrero Mar Nov Oct
```

Si se quiere tener acceso al factor como un vector de índices, se convierte a entero:

```
as.integer(Fmes.nacimiento)

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

1.5. Listas

Una lista, de la clase `list`, es una clase de datos que puede contener cero o más elementos, cada uno de los cuales puede ser de una clase distinta. Por ejemplo, se puede concebir una lista para representar una familia: la mamá, el papá, los años de casados, los hijos, y las edades de los hijos, de la manera siguiente:

```
familia <- list("Maria", "Juan", 10, c("Hugo", "Petra"), c(8,
    6))
familia

## [[1]]
## [1] "Maria"
##
## [[2]]
## [1] "Juan"
##
## [[3]]
## [1] 10
##
## [[4]]
## [1] "Hugo"  "Petra"
##
## [[5]]
## [1] 8 6
```

Nótese que la lista contiene cinco elementos; los tres primeros son a su vez de un sólo elemento: el nombre de la mamá, el nombre del papá, y los años

de casados. Los siguientes dos, son dos vectores de dos elementos cada uno: los hijos y sus respectivas edades.

Al igual que en el caso de los vectores, como se vio en la sección 1.2.4 en la página 9, los elementos de las listas pueden ser nombrados, lo que añade mayor claridad a su significado dentro de la lista. La forma de hacer esto se muestra a continuación:

```
familia <- list(madre="Maria", padre="Juan", casados=10,
                hijos=c("Hugo", "Petra"), edades=c(8, 6))
familia

## $madre
## [1] "Maria"
##
## $padre
## [1] "Juan"
##
## $casados
## [1] 10
##
## $hijos
## [1] "Hugo"  "Petra"
##
## $edades
## [1] 8 6
```

1.5.1. Acceso a los elementos individuales de una lista

Al igual que en el caso de los vectores, las listas no serían de mucha utilidad sin la posibilidad de tener acceso a sus elementos individuales. El lenguaje, provee de este acceso mediante tres operadores, a saber: `[ ]`, `[[ ]]`, y `$`. El primero de estos operadores se revisará a detalle en el capítulo **??**. Aquí se explicarán los otros dos operadores en su forma de uso más simple.

Cuando los elementos de la lista tienen nombre, se puede acceder a ellos con cualquiera de los dos operadores. Usando los ejemplos anteriores, esto se puede hacer de la manera siguiente:

```
# Acceso de lectura
familia$madre

## [1] "Maria"

familia[["madre"]]

## [1] "Maria"
```

```
# Acceso de escritura
familia[["padre"]] <- "Juan Pedro"
familia$padre  # para checar el nuevo valor

## [1] "Juan Pedro"
```

Nótese que al emplear el operador `$`, no se han usado las comillas para mencionar el nombre del elemento, pero, este operador también admite nombres con comillas. Por otra parte, el operador `[[ ]]`, sólo admite los nombres de elementos con comillas, o de cualquier expresión que al evaluarse dé como resultado una cadena de caracteres. En seguida se muestran algunos ejemplos:

```
familia$"madre" <- "Maria Candelaria"
mm <- "madre"
familia[[mm]]

## [1] "Maria Candelaria"

familia[[ paste("ma", "dre", sep="") ]]

## [1] "Maria Candelaria"
```

En el último caso, el operador ha recibido como argumento la función `paste()`, que, como se ha dicho anteriormente, en la página 23, sirve para concatenar cadenas de caracteres. Esta función supone de inicio que las cadenas irán separadas por un espacio en blanco. Por ello es que, en el ejemplo se indica que el separador es vacío mediante `sep=""`. Alternativamente, para este último caso, se puede usar la función `paste0()`, que de entrada supone que tal separador es vacío.

1.6. *Data frames*

Un *data frame*[1] es una lista, cuyos componentes pueden ser vectores, matrices o factores, con la única salvedad de que las longitudes, o número de renglones, en el caso de matrices, deben coincidir en todos los componentes. La apariencia de un *data frame* es la de una tabla y una forma de crearlos es mediante la función `data.frame()` . Veamos un ejemplo:

```
(m <- cbind(ord=1:3, edad=c(30L, 26L, 9L)) )

##      ord edad
## [1,]   1   30
## [2,]   2   26
## [3,]   3    9
```

[1] *Usamos aquí el témino anglosajón, "data frames", y no su traducción al castellano, "marco o estructura de datos", dado que estos nombres sólo introducirían confusión, pues ninguno de ellos da una pista de lo que es. Probablemente un término apropiado sería algo como "tabla de datos".*

```
(v <- c(1.80, 1.72, 1.05) )

## [1] 1.80 1.72 1.05

ff <- data.frame(familia=c("Padre", "Madre", "Hijo"),
                 m, estatura=v)
ff

##   familia ord edad estatura
## 1   Padre   1   30     1.80
## 2   Madre   2   26     1.72
## 3    Hijo   3    9     1.05
```

Una gran ventaja de los *data frames*, es que R tiene diversas funciones para leer y guardar las tablas que representan, en archivos de texto, y otros formatos. Como un ejemplo, supongamos que se tiene un archivo, denominado "Rtext.txt", la siguiente información:

	Precio	Piso	Area	Cuartos	Edad	Calentador
01	52.00	111.0	830	5	6.2	no
02	54.75	128.0	710	5	7.5	no
03	57.50	101.0	1000	5	4.2	no
04	57.50	131.0	690	6	8.8	no
05	59.75	93.0	900	5	1.9	si

La lectura de esta tabla hacia un *data frame*, es muy sencilla y se hace mediante la función `read.table()`[2], como sigue:

```
mi.tabla <- read.table("Rtext.txt")
mi.tabla

##    Precio Piso Area Cuartos Edad Calentador
## 01  52.00  111  830       5  6.2         no
## 02  54.75  128  710       5  7.5         no
## 03  57.50  101 1000       5  4.2         no
## 04  57.50  131  690       6  8.8         no
## 05  59.75   93  900       5  1.9         si
```

Nótese que el primer renglón y la primera columna no son parte de los datos de la tabla; ellos son, respectivamente, los nombres de las columnas y renglones de la tabla o *data frame*, lo que podemos constatar mediante las funciones `colnames()` y `rownames()`:

[2] Aparte de la función `read.table()`, existen otras funciones que permiten leer datos de algún tipo de archivo y vaciarlos en una estructura de tipo data frame. Probablemente, una de las más útiles es la función `read.csv()`, que permite hacer esta operación a partir de archivos que contienen valores separados por comas, uno de los formatos de intercambio de información entre manejadores de hojas de cálculo, como Excel, más usados.

```
colnames(mi.tabla)

## [1] "Precio"     "Piso"       "Area"       "Cuartos"
## [5] "Edad"       "Calentador"

rownames(mi.tabla)

## [1] "01" "02" "03" "04" "05"
```

Como se mencionó anteriormente, un *data frame* es una lista muy particular, pero, ¿cuáles son los elementos de esa lista? Los elementos de la lista, y que obedecen a todas las reglas sintácticas dadas anteriormente (ver sección 1.5.1), son las columnas de la tabla. Así, por ejemplo, al segundo elemento de la lista podemos tener acceso de las siguientes formas:

```
mi.tabla$Piso

## [1] 111 128 101 131  93

mi.tabla[[2]]

## [1] 111 128 101 131  93

mi.tabla[2]

##    Piso
## 01  111
## 02  128
## 03  101
## 04  131
## 05   93
```

En el último caso, los datos se despliegan junto con el nombre de la columna y cada uno de los nombres de los renglones. Ello se debe a que en realidad, el operador `[]` extrae una *rebanada* del dato o variable sobre la cuál opera, un *data frame* en este caso, y que podríamos denominarlo como un *sub-data frame* aquí; esto es, se trata otra vez de un *data frame* pero más *chiquito* que el original. Los detalles de este operador se discutirán a detalle más adelante en el texto.

Para tener acceso a un elemento individual de un data frame, se utiliza el operador `[]`, con la misma sintaxis que se utilizó para las matrices. Por ejemplo, el elemento en el renglón 3 y la columna 2, se puede revisar, o incluso cambiar con:

```
mi.tabla[3, 2]

## [1] 101
```

```
# modificamos el elemento con:
mi.tabla[3, 2] <- 106
mi.tabla

##    Precio Piso Area Cuartos Edad Calentador
## 01  52.00  111  830       5  6.2         no
## 02  54.75  128  710       5  7.5         no
## 03  57.50  106 1000       5  4.2         no
## 04  57.50  131  690       6  8.8         no
## 05  59.75   93  900       5  1.9         si
```

Otra característica importante de los *data frames* es que, salvo que se indique otra cosa, las columnas de tipo `character` se convierten automáticamente a tipo `factor`:

```
mi.tabla$Calentador

## [1] no no no no si
## Levels: no si

class(mi.tabla$Calentador)

## [1] "factor"
```

La posibilidad de operar con *rebanadas* de los data frames, es una de las cosas que hacen más atractivas a esta estructura. Si, por ejemplo, se quiere añadir, una nueva columna o componente del *data frame*, y ésta calcularla como el resultado de multiplicar el Precio por el Area, se puede hacer de la siguiente manera:

```
mi.tabla$Total <- mi.tabla$Precio * mi.tabla$Area
mi.tabla

##    Precio Piso Area Cuartos Edad Calentador Total
## 01  52.00  111  830       5  6.2         no 43160
## 02  54.75  128  710       5  7.5         no 38872
## 03  57.50  106 1000       5  4.2         no 57500
## 04  57.50  131  690       6  8.8         no 39675
## 05  59.75   93  900       5  1.9         si 53775
```

1.7. Funciones

A diferencia de otros lenguajes de programación procedurales, como C, Java, y PHP, en R las funciones constituyen una *clase*. Por ejemplo, los objetos de esa *clase* pueden ser asignados a variables; podría darse el caso, incluso, de armar una lista cuyos elementos fueran funciones.

Aunque la escritura de funciones es parte de la programación que se verá más adelante, se indicará aquí la forma de crear funciones como una herramienta para agrupar varias operaciones.

La sintaxis para la creación de una función es como sigue:

variable <- **function**(*arg_ 1*, *arg_ 2*, ..., *arg_ n*) *expresion*

Como se puede ver, se trata de una asignación de un valor: la función, a una variable. A partir de esa definición, la variable se puede utilizar como el *nombre* de la función. En R, toda expresión tiene un valor, así que el valor de la *expresión* será lo que la función regresará cuando se aplique. En seguida se muestra un ejemplo.

```
hipotenusa <- function(x, y) {
    sqrt(x^2 + y^2)
}
class(hipotenusa)

## [1] "function"
```

En este caso, la función de biblioteca `sqrt()`, entrega la raíz cuadrada, y el operador `^`, eleva un valor a la potencia indicada como segundo argumento. La función entrega como resultado el último valor calculado que encuentre, aunque esta entrega se pude hacer explícita mediante la instrucción `return`, con lo cual la función anterior podría alternativamente ser codificada como:

```
hipotenusa <- function(x, y) {
    return(sqrt(x^2 + y^2))
}
```

Para utilizar esta función lo hacemos con:

```
hipotenusa(3, 4)

## [1] 5
```

Los argumentos de la función tienen nombres, y esos se pueden usar en el llamado a la función, cambiando incluso el orden en que aparecen en la definición de la función.

```
hipotenusa(y = 4, x = 3)

## [1] 5
```

Otra característica es que las funciones, en su definición, pueden tener valores asignados por defecto o en ausencia cuando es llamada la función:

```
hipotenusa <- function(x=3, y=4) { # valores por ausencia
    return( sqrt( x^2 + y^2 ) )
}
# Llamamos a la funcion con argumentos "ausentes"
hipotenusa()

## [1] 5
```

Las funciones toman sus datos de los argumentos dados o de las variables que "le están al alcance" a la función, así por ejemplo, la siguiente función:

```
ff <- function(r) {
    return(PI * r^2)
}
```

si se ejecuta esta función, por ejemplo, con `ff(3)`, puede disparar un error, ya que no se ha definido el valor de PI. Pero si se ejecuta en la siguiente secuencia, se obtendrá un valor, aun cuando la función se haya definido con anterioridad:

```
PI <- 3.1416
ff(3)

## [1] 28.27
```

La noción de función que se ha presentado aquí es muy básica. En un capítulo posterior se presenta la creación de funciones enriquecida por las estructuras de control que se discutirán también más adelante.

1.8. Coerción

Se han abordado en este capítulo, no de una manera exhaustiva, pero sí para tener una idea clara de su potencial, los principales tipos de datos del lenguaje R. Estos tipos de datos son el fundamento para la construcción de otras *clases* de datos más complejas. Algunos de los tipos de datos admiten su conversión a otros tipos; para ello, el lenguaje provee de un conjunto de funciones de la forma: `as.<tipo>()`. En seguida se muestran algunos ejemplos.

Distintas conversiones entre datos numéricos:

```
x <- 1.03
x

## [1] 1.03

y <- as.integer(x)  # conversion a entero
y
```

```
## [1] 1

z <- as.complex(y)  # conversion a complejo
z

## [1] 1+0i

a <- c("1000", "2013.1", "0")
class(a)

## [1] "character"

b <- as.numeric(a)  # conversion de character a otro tipo
b

## [1] 1000 2013    0

class(b)

## [1] "numeric"

c <- as.logical(b)  # conversion a logico
# 0 es FALSE y distinto de 0 es TRUE
c

## [1]  TRUE  TRUE FALSE
```

También, puede haber conversiones entre clases de datos más estructuradas. Una que se antoja inmediata es la conversión de una matriz a un *data frame*:

```
(m <- matrix(1:20, nrow = 5, ncol = 4))

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

ff <- as.data.frame(m)
ff

##   V1 V2 V3 V4
## 1  1  6 11 16
## 2  2  7 12 17
## 3  3  8 13 18
## 4  4  9 14 19
## 5  5 10 15 20
```

Nótese, en este último caso, que la función de conversión automáticamente ha asignado nombres a los renglones y las columnas del *data frame* creado. Para tener acceso a los elementos del *data frame*, lo podemos hacer mediante los nombres asignados a las columnas:

```
ff$V2

## [1]  6  7  8  9 10
```

Existen muchas más conversiones posibles; pero baste por ahora con las que se han visto, que ellas dan una idea de los mecanismos usados en el lenguaje para este tipo de operaciones.