Chapter 1

Programación en R

Introducción

R es un dialecto del lenguaje *S* y esta a su vez es un lenguaje que fue desarrollado por John Chambers en Bell Labs en los 70's, fue pensado como un ambiente para el análisis estadístico dentro de la compañía.

Inicialmente se implementó como una serie de bibliotecas de Fortran, además de que las primeras versiones no contenían funciones para análisis estadístico. El sistema fue reescrito en los 80's en C y es entonces que Chambers y Hastie documentan en "*Statistical Models in S*" la funcionalidad del lenguaje para el análisis estadístico. En 1998, sale la versión 4 que conocemos hoy en día, y aparece el libro "*Programming with Data*" de John Chambers que detalla las minucias del lenguaje.

Básicamente, *S* fue pensado como un ambiente interactivo que no necesariamente se puede pensar como un lenguaje de programación, pero mientras el usuario va avanzando utilizando las características de este, entra de manera fácil en prácticas de programación.

Provée un acceso relativamente simple a una gran variedad de técnicas estadística y gráficas. Para los usuarios avanzados se ofrece un lenguaje de programación completo con el que añadir nuevas técnicas por medio de la definición de funciones.

Un poco de la historia de *R*:

- ♢ 1996, *R-help*, *R-devel*
- ♢ 1997, *R Core Group*
- ♢ 2000, *R versión* 1.0
- ♢ 2020, *R versión* 4.0

Características:

- ♢ Sintaxis y semántica similares a *S*
- ♢ Funciona en casi cualquier plataforma
- ♢ Tiene cambio de versiones muy rápido

- ♢ Muy pequeño, la funcionalidad se le agrega utilizando "paquetes"
- ♢ Capacidad de graficación bastante madura
- ♢ Modo interactivo para probar ideas pero es un lenguaje para desarrollar nuevas herramientas

Ventajas de R:

- ♢ Una comunidad muy activa; las listas R-help, R-devel, y en Stack Overflow.
- ♢ Software libre

Desventajas

- ♢ Sin sistema de gráficos en 3D integrado
- ♢ Funcionalidad basada en sus usuarios
- ♢ Los objetos deben estar en la memoria RAM
- ♢ No es ideal para todo

¿Por qué usarlo?

- ♢ Tiene muchas funciones estadísticas
- ♢ Podemos implementar de manera rápida procedimientos estadísticos y de gráficos
- ♢ Buenos gráficos base y avanzados

¿Quién está detrás de R?

- ♢ Desde 1997 se mantiene con un equipo llamado el "*R Development Core Team*"
- ♢ Financiado por la *R Foundation, non-profit*, basada en Viena
- ♢ Más información en: http://www.r-project.org

Hay varias maneras de encontrar ayuda cuando hay problemas en R:

- ♢ Utilizar el sistema de ayuda de R
- ♢ Listas de distribución
- ♢ *Stackoverflow*

El sistema de ayuda

El sistema de ayuda del sistema es bastante simple y se puede resumir a estas funciones:

- ♢ help( )
- ♢ example( )
- ♢ help.search( )
- ♢ library(help = " ")
- ♢ vignette( )

https://rdrr.io/snippets/

1.1 Objetos

Casi todo es un objeto, incluyendo las funciones y estructuras de datos; R incluye ciertos tipos de objetos predefinidos, entre ellos:

- ♢ Propios del lenguaje: llamadas, expresiones, nombres
- ♢ Expresiones: Colecciones de expresiones correctas no evaluadas
- ♢ Funciones: Constan de lista de argumentos, código, entorno

Al escribir el nombre de un objeto nos regresa su contenido.

R tiene 5 tipos de objetos básicos o "atómicos":

- ♢ *character*
- ♢ *numeric*
- ♢ *integer*
- ♢ *complex*
- ♢ *logical*

1.1.1 Atributos

Los objetos en R pueden tener diferentes atributos; entre los más comúnes:

- ♢ *names*, *dimnames*
- ♢ *dimensions* (matrices, arreglos)
- ♢ *class*
- ♢ *length*
- ♢ Otros atributos definidos por el usuario

Se pueden acceder usando la función *attributes*( )

1.1.2 Números

Los números en R se tratan generalmente como objetos numéricos (números de punto flotante de doble precisión). Si queremos un entero de manear explícita, se necesita usar el sufijo *L*. Los valores *NaN* e *Inf* son numéricos.

Entrada de comandos

- ♢ En el *prompt* o línea de comando de R podemos escribir expresiones, como si fuera una calculadora
- ♢ $<-$ es el símbolo de asignación
- ♢ La gramática del lenguaje determina si una expresión está completa o no
- ♢ Con # podemos hacer comentarios al código

Las expresiones se evalúan al presionar *Enter* y el prompt devuelve el resultado, en ocasiones se imprime.

Ejemplos:

```
> x <- 1
> x
[1] 1
> mensaje <- "hola mundo" #comentario
> mensaje
[1] "hola mundo"
```

El [1] indica un vector y el primer elemento que contiene.

Con el operador : se crean secuencias de enteros:

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

1.2 Operadores básicos

R incluye muchos operadores comúnes:

♢ Aritméticos:

- suma +, resta −
- multiplicación ∗, división /
- exponenciación ^, ∗∗
- módulo %%
- división entera %/%

```
> 5+2
[1]
7
> 5-2
[1] 3
> 5*2
[1] 10
> 5/2
[1] 2.5
> 5^2
[1] 25
> 5%%2
[1] 1
> 5%/%2
[1] 2
```

♢ Lógicos:

- negación !
- conjunción &, &&
- disyunción |, ||

```
> ! TRUE
[1] FALSE
> T & T
[1] TRUE
> T && F
[1] FALSE
> T | F
[1] TRUE
> F || F
[1] FALSE
```

♢ De comparación:

- $<, <=, >, >=$
- $==, != $

```
> 3<4
[1] TRUE
> 3<=4
[1] TRUE
> 3>4
[1] FALSE
> 3>=4
[1] FALSE
> 3==4
[1] FALSE
> 3!=4
[1] TRUE
```

1.2.1 Reglas de precedencia y asociatividad

La precedencia de operadores define el orden en el que se evalúan las expresiones; cuando incluyen múltiples operadores se siguen ciertas reglas que indican el orden en el que se evaluarán cada una de las subexpresiones que la componen, dichas reglas se conocen *precedencia de operadores*. Los operadores con precedencia alta se evaluarán primero.

Cuando se tienen operadores con la misma precedencia se siguen las llamadas *reglas de asociatividad* para ese grupo de operadores. En R los operadores pueden no tener asociatividad o bien, tenerla por la izquierda o la derecha. Los operadores con asociatividad izquierda se evalúan de izquiera a derecha, aquellos que tienen asociatividad derecha se evalúan de derecha a izquierda y los que no tienen asociatividad no siguen un orden predefinido.

La tabla 1.1 muestra la precedencia de opearodes en R, así como su asociatividad; la precedencia disminuye de arriba a abajo.

Operador	Descripción	Asociatividad
$\hat{}$	exponenciación	derecha
$+x, -x$	*más* y *menos* unarios	izquierda
$\%\%$	módulo	izquierda
$*, /$	multiplicación, y división	izquierda
$+, -$	suma y resta	izquierda
$<, <=, >, >=, ==, !=$	comparaciones	izquierda
!	negación	izquierda
$\&, \&\&$	conjunción	izquierda
$\|, \|\|$	disyunción	izquierda
$->, ->>$	asignación a la derecha	izquierda
$<-, <<-$	asignación a la izquierda	derecha
$=$	asignación a la izquierda	derecha

Table 1.1: Precedencia y asociatividad de operadores

1.3 Estructuras de control

Al igual que en otros lenguajes de programación, permiten controlar el flujo de ejecución de un programa dependiendo de ciertas condiciones en el mismo; también permiten que se lleven a cabo tareas de manera repetida:

- ♢ Condicional: *if ... else*
- ♢ Ciclo de repetición: *for*
- ♢ Ciclo de condición: *while*
- ♢ Ciclo *infinito*: *repeat*
- ♢ Romper un ciclo: *break*
- ♢ Salta una iteración: *next*

1.3.1 Condicional

Ejecuta cierta sección de código dependiendo de la evaluación de su condición, su sintaxis es:

```
if(cond1){
  # operaciones cuando se cumple cond1
}else if(cond2){
  # operaciones cuando se cumple cond2
}else{
  # operaciones cuando no se cumple ninguna
}
```

Ejemplo:

```
> x <- 5
> y <- 7
> if(x>y){
    print('X es mayor')
  }else if(x<y){
    print('Y es mayor')
  }else{
    print('son iguales')
  }
[1] "Y es mayor"
```

1.3.2 Ciclo *for*

Permite iterar los elementos de una estructura dada, su sintaxis es:

```
for(iterador){
  # operaciones
}
```

Ejemplos:

```
> letras <- c('a','b','c','d')
> for(l in letras){
    print(l)
  }
[1] "a"
[1] "b"
[1] "c"
[1] "d"

> for(i in 1:length(letras)){
    print(letras[i])
  }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Se puede usar directamente sobre vectores:

```
> x <- seq(from=1, to=5, by=0.5)
> for(i in x){
    print(i)
  }
[1] 1
[1] 1.5
[1] 2
[1] 2.5
[1] 3
[1] 3.5
[1] 4
[1] 4.5
[1] 5
```

1.3.3 Ciclo condicional

Se utiliza para repetir cierta sección del código mientras una condición sea verdadera, su sintaxis es:

```
while(cond){
  # operaciones
}
```

Ejemplo:

```
> while(cont<10){
    print(cont)
    cont <- cont+2
}
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

1.3.4 *repeat* y *break*

Se recomienda su uso combinado, aunque no es buena idea utiizarlos porque fácilmente se podría obtener un ciclo infinito, pueden simular el ciclo *while*, su sintaxis es:

```
repeat{
  # operaciones
  if(cond){
    # operaciones
    break
  }else{
    # operaciones
  }
}
```

1.3.5 Salto de iteración

Se utiliza cuando se desea que bajo cierta condición, no se ejecute una iteración del código, su sintaxis es:

```
for(iterador){
  if(cond){
    # si se cumple cond, salta
    # la iteración actual
    next
  }
  # operaciones
}
```

Ejemplo:

```
> for(i in 1:10){
    if(i%%2 == 0){
       next
    }
    print(i)
  }
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

1.4 Vectores, matrices y otros tipos de datos

1.4.1 Creación de vectores

Para crear vectores de objetos se utiliza la función *c*( ), también se puede usar la función *vector*( ):

```
> x <- c(0.3, 0.9) # numérico
> class(x)
[1] "numeric"
> length(x)
[1] 2
> names(x)
NULL
> dimnames(x)
NULL
```

```
> x <- c(TRUE, FALSE) # lógico
> class(x)
[1] "logical"
> x <- c(T, F) # lógico equivalente
```

```
> x <- c('a', 'b', 'c') # caracteres
> class(x)
[1] "character"
```

```
> x <- c(1+1i, 2-3i)
> class(x)
[1] "complex"
```

```
> x <- vector('numeric', 10)
> class(x)
[1] "numeric"
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

1.4.1.1 Mezcla de objetos

Cuando se combinan diferentes tipos de objetos, se lleva a cabo una "*coerción*", lo que genera que todos los objetos de un vector sean de la misma clase:

```
> x <- c(1+1i, 2)
> class(x)
[1] "complex"
> x <- c(1+1i, 'a')
> class(x)
[1] "character"
> x <- c('a',T)
> class(x)
[1] "character"
> x <- c(2, T)
> class(x)
[1] "numeric"
```

Los objetos pueder ser explícitamente coercionados de una clase a otra, utilizando las funciones as.*:

- ♢ as.numeric( )
- ♢ as.logical( )
- ♢ as.character( )
- ♢ as.integer( )

```
> x <- 0:10
> a <- as.numeric(x)
> class(a)
[1] "numeric"
> b <- as.logical(x)
> class(b)
[1] "logical"
> c <- as.character(x)
> class(c)
[1] "character"
> d <- as.integer(x)
> class(d)
[1] "integer"
```

Mostrando el contenido de cada vector:

```
> a
 [1]  0  1  2  3  4  5  6  7  8  9 10
> b
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
TRUE
> c
 [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
> d
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

Puede suceder que se intente coercionar un vector a un tipo de objeto que puede resultar en algo que no tiene sentido:

```
> x <- c('a', 'b', 'c')
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

1.4.2 Creación de matrices

Las matrices es el siguiente tipo de objeto que tenemos, estos son vectores con un atributo de dimensión. La dimensión es un atributo que es un vector entero de tamaño 2: (*nrow*, *ncol*), este vector indica el número de renglones y columnas de la matriz.

Se crean con la función *matrix*:

```
> m <- matrix(nrow=2, ncol=3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

```
> class(m)
[1] "matrix"
> attributes(m)
$dim
[1] 2 3
> dim(m)
[1] 2 3
> class(dim(m))
[1] "integer"
```

A diferencia de otros lenguajes, las matrices se rellenan por columnas, esto significa que los valores de un vector que formaría una matriz van rellenando a la matriz columna por columna de izquierda a derecha:

```
> m <- matrix(1:6, nrow=2, ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

También pueden ser creadas directamente de vectores si les añadimos un atributo de dimensión:

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m)
NULL
> dim(m) <- c(2,5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

También se puede crear matrices utilizando *cbind* y *rbind*, estas funciones "pegan" columnas o renglones:

```
> x <- 6:9
> y <- 11:14
> cbind(x,y)
      x  y
[1,] 6 11
[2,] 7 12
[3,] 8 13
[4,] 9 14
> rbind(x,y)
  [,1] [,2] [,3] [,4]
x    6    7    8    9
y   11   12   13   14
```

Y se pueden añadir filas/columnas con las mismas operaciones:

```
> x <- 6:9
> y <- 11:14
> m <- cbind(x,y)
> z <- 1:4
> cbind(m,z)
      x  y z
[1,] 6 11 1
[2,] 7 12 2
[3,] 8 13 3
[4,] 9 14 4

> m <- rbind(x,y)
> rbind(m,z)
  [,1] [,2] [,3] [,4]
x    6    7    8    9
y   11   12   13   14
z    1    2    3    4
```

Para obtener la transpuesta de una matriz:

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

1.4.3 Listas

Las listas son un tipo especial de vector, que puede contener elementos de diferentes clases:

```
> x <- list(1, 'a', T, 1+1i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+1i
```

1.4.4 Factores

Los factores son otro tipo especial de vectores, se usan para representar datos *categóricos*, que pueden estar ordenados o sin orden:

```
> x <- factor(c('si','no','no','si'), level=(c('si','no')))
> x
[1] si no no si
Levels: si no
```

En general, si no se indican los niveles, R los infiere de los datos:

```
> x <- factor(c('si','no','no','si'))
> x
[1] si no no si
Levels: no si
> table(x)
x
 no si
  2  2
 > unclass(x)
[1] 2 1 1 2
attr(,"levels")
[1] "no" "si"
```

El orden de los niveles puede establecerse utilizando el argumento *level* a la función *factor*( ). Esto es importante en modelado lineal, porque el primer nivel siempre es usado como el nivel base:

```
> x <- factor(c('si','no','no','si'), level=(c('si','no')))
> x
[1] si no no si
Levels: si no
```

1.4.4.1 Valores faltantes

Los valores faltantes se denotan con *NA* o *NaN* para operaciones indefinidas. Podemos probar si estos valores existen en nuestros objetos utilizando las funciones:

- ♢ *is.na*( )
- ♢ *is.nan*( )

Los valores *NA* también tienen una clase, hay *enteros NA*, *character NA*, etc. Un valor *NaN* es al mismo tiempo un *NA* pero no al contrario.

```
> x <- c(5,6,NA,NaN,9)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE  TRUE FALSE
```

1.4.4.2 Agregar elementos

Para añadir elementos en un vector o lista, se puede utilizar la función *append*:

```
> x <- list(1, 'a', T, 1+1i)
> append(x,6)

> x <- c(1,2,3)
> append(x,4)
```

Se puede indicar la posición en la que se añadirá:

```
> x <- list(1, 'a', T, 1+1i)
> append(x,6,after=3)

> x <- c(1,2,3)
> append(x,0,after=2)
```

También es posible, y recomendable cuando es al final, utilizar la función *c*:

```
> x <- list(1, 'a', T, 1+1i)
> c(x,6)

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

Esta función no permite indicar una posición para insertar:

```
> x <- c(1,2,3)
> c(x,4,after=2)
```

1.4.5 Dataframes

Este es un tipo en R que nos sirve para guardar datos tabulares. Es uno de los tipos de datos más importantes, ya que con éste podemos llevar a cabo todos los análisis que necesitemos de manera fácil.

Se pueden crear utilizando *data.frame*( ), *read.table*( ) ó *read.csv*( ). La función *data.matrix*( ) sirve para convertir un dataframe a matriz.

Se representan como un tipo especial de lista donde cada elemento de la lista tiene la misma longitud. Cada elemento de la lista se puede ver como una columna, y la longitud de cada elemento de la lista es el número de renglones.

Contrario a las matrices, los dataframes pueden guardar diferentes tipos de datos en cada columna. Tiene un atributo llamado *row.names*.

```
> x <- data.frame(col1=1:4, col2=c(T,F,T,F))
> x
  col1  col2
1    1  TRUE
2    2 FALSE
3    3  TRUE
4    4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Si se intenta generar un *data.frame* con diferente número de filas en las columnas se gerena un error:

```
> x <- data.frame(col1=1:5, col2=c(T,F,T,F))
Error in data.frame(col1 = 1:5, col2 = c(T, F, T, F)) :
   arguments imply differing number of rows: 5, 4
```

Se puede indicar nombres para los objetos con el fin de tener código más limpio y fácil de leer porque los objetos se describen directamente. También es posible usar nombres también con listas.

```
> x <- 12:15
> names(x)
NULL
> names(x) <- c('doce','trece','catorce','quince')
> x
   doce   trece catorce  quince
     12      13      14      15
```

```
> y <- list(1,3,5)
> names(y) <- c('uno','dos','tres')
> y
$uno [1]
1
$dos [1]
3
$tres [1]
5
```

Las matrices también pueden tener nombres:

```
> m <- matrix(5:8, nrow=2, ncol=2)
> dimnames(m) <- list(c('x','y'), c('i','j'))
> m
  i j
x 5 7
y 6 8
```

1.4.6 Operaciones vectorizadas

Al ser uno de tipos de datos más importantes dentro de R, es natural definir operaciones sobre ellos de forma que los códigos sean más legibles y eficientes.

```
> x <- 6:9
> y <- 10:13
> x+y
[1] 16 18 20 22
> x > 8
[1] FALSE FALSE FALSE  TRUE
> y==10
[1]  TRUE FALSE FALSE FALSE
> x*y
[1]  60  77  96 117
> x/y
[1] 0.6000000 0.6363636 0.6666667 0.6923077
> x%*%y
     [,1]
[1,]  350
```

1.4.7 Subconjuntos

1.4.7.1 Vectores

R tiene operadores para hacer *subsetting*; para un vector, [ ] extrae elementos similar al uso de índices; los índices en R inician en 1.

```
> x <- c('a', 'b', 'c', 'd', 'e')
> x[1]
[1] "a"
> x[2:5]
[1] "b" "c" "d" "e"
> index <- x > 'b'
> x[index]
[1] "c" "d" "e"
> x[x > 'b'] # se puede escribir directamente la condición
[1] "c" "d" "e"
```

1.4.7.2 Matrices

Para matrices existen:

- ♢ $m\,[x, y]$, un elemento particular
- ♢ $m\,[x,]$, un renglón completo
- ♢ $m\,[, y]$, una columna completa

```
> x <- matrix(1:10,2,5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x[2,4]
[1] 8
> x[1,]
[1] 1 3 5 7 9
> x[,4]
[1] 7 8
> x[3,4]
Error in x[3, 4] : subscript out of bounds
```

1.4.7.3 Listas

Para listas puede usarse:

- ♢ *lista*[ ]
- ♢ *lista*[[ ]]
- ♢ *lista*$*nombre*, sólo puede usarse con nombres literales

```
> x <- list(hola=1:10, mundo=1+1i)
> x
$hola
[1]  1  2  3  4  5  6  7  8  9 10
$mundo
[1] 1+1i
> x[1]
$hola
 [1]  1  2  3  4  5  6  7  8  9 10
> x[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10
> x$hola
[1]  1  2  3  4  5  6  7  8  9 10
> x$mundo
[1] 1+1i
> x['mundo']
$mundo
[1] 1+1i
```

Se puede usar pasando un vector como parámetro:

```
> x <- list(hola=1:10, mundo=1+1i, cruel='puente?')
> x[c(1,3)]
$hola
 [1]  1  2  3  4  5  6  7  8  9 10
$cruel
[1] "puente?"
```

lista[[ ]] es equivalente a *lista*$*nombre* cuando se utiliza el nombre del elemento:

```
> x <- list(hola=1:10, mundo=1+1i, cruel='puente?')
> x[['cruel']]
[1] "puente?"
> x$cruel
[1] "puente?" >
```

1.4.7.4 Elementos anidados

Se puede acceder a elementos dentro de otros elementos:

```
> x <- list(b=list(10,11), b=c(9.8,2.33))
> x[[c(1,2)]]
[1] 11
> x[[1]][[2]] # resultado igual al anterior
[1] 11
> x[[c(2,1)]]
[1] 9.8
```

Notar la diferencia:

```
> x <- list(b=list(10,11), b=c(9.8,2.33))
> x
$b
$b[[1]]
[1] 10
$b[[2]]
[1] 11
$b [1]
9.80 2.33

> x <- c(b=list(10,11), b=c(9.8,2.33))
> x
$b1
[1] 10
$b2
[1] 11
$b1
[1] 9.8
$b2
[1] 2.33
```

1.4.8 Valores faltantes

Es común encontrar valores *NA* (faltantes), necesitamos poder trabajar a pesar de eso.

```
> x <- c(1234,NA,NA,42,NA)
> faltantes <- is.na(x)
> faltantes
[1] FALSE  TRUE  TRUE FALSE  TRUE
```

Para quedarnos con aquellos que sí contienen valor:

```
> x <- c(1234,NA,NA,42,NA,64)
> x[!is.na(x)]
[1] 1234   42   64
```

Para el caso de *data.frame* se puede utilizar la función *complete.cases*()

```
> x <- data.frame(x1 = c(7, 2, 1, NA, 9),
+                 x2 = c(1, 3, 1, 9, NA),
+                 x3 = c(NA, 8, 8, NA, 5))
> x
  x1 x2 x3
1  7  1 NA
2  2  3  8
3  1  1  8
4 NA  9 NA
5  9 NA  5
> complete.cases(x)
[1] FALSE  TRUE  TRUE FALSE FALSE
> x[complete.cases(x),]
  x1 x2 x3
2  2  3  8
3  1  1  8
```

1.5 Funciones

En R se utiliza *function*() para crear y definir una función y deben almacenarse con algún nombre; su sintaxis es:

```
nombre <- function(params){
  # operaciones
}
```

Ejemplo (es mejor escribir el código de las funciones como un *script* de R):

```
doble <- function(n){
  2*n
}
```

```
> doble(5)
[1] 10
```

Las funciones en R son objetos y, por tanto, se tratan como cualquier otro tipo:

♢ Se pueden usar como parámetros de otras funciones

♢ Se pueden definir funciones dentro de funciones

♢ El valor que regresan es la última expresión evaluada

1.5.1 Parámetros

Los parámetros formales son los que se indican en la definición de la función. La función *formals*() devuelve la lista de parámetros de una función:

```
> formals(doble)
$n

> formals(matrix)
$data
[1] NA
$nrow
[1] 1
$ncol
[1] 1
$byrow
[1] FALSE
$dimnames
NULL

> formals(formals)
$fun
sys.function(sys.parent())
$envir
parent.frame()
```

Es posible definir valores por omisión para los argumentos de una función:

Ejemplo:

```
multiplo <- function(n, m=1){
  n*m
}
```

```
> multiplo(2,5)
[1] 10
```

Es posible hacer coincidencia por posición y también por nombre del parámetro:

```
> multiplo(m=5,n=2)
[1] 10
```

Las siguientes llamadas de la función *matrix* son equivalentes:

```
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> matrix(ncol=3, nrow=2, data=1:6)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

1.5.1.1 Parámetros *perezosos*

El paso de parámetros reales sólo será obligado cuando se utilicen realmente.

```
suc <- function(x,y){
  x+1
}
```

```
> suc(1)
[1] 2
```

Un poco más interesante:

```
inc <- function(x,y){
  if(x<5){
    x+1
  }else{
    x+y
  }
}
```

```
> inc(4)
[1] 5
> inc(6)
Error in inc(6) : argument "y" is missing, with no default
> inc(6,2)
[1] 8
```

1.5.1.2 Parámetro "..."

Se utiliza para indicar que la función puede recibir un número de parámetros desconocido en principio

```
f_var <- function(...){
  print(substitute(alist(...))) # capturar la lista de valores
}
```

Si se desean pasar otros argumentos despues de "..." debe llevar explícitamente su nombre.

1.5.2 Ejemplos

♢ Función que recibe un *vector*, un *umbral* y devuelve los elementos del vector que sean mayores a ese umbral:

```
mayores <- function(vec, umb=0){
  ind <- vec > umb
  vec[ind]
}
```

```
> mayores(1:25,15)
[1] 16 17 18 19 20 21 22 23 24 25
```

♢ Función que suma los valores de cada columna en una matriz:

```
suma_cols <- function(matriz){
  n_col <- ncol(matriz)
  n_fil <- nrow(matriz)
  suma_col <- numeric(n_col)
  for(j in 1:n_col){
    for(i in 1:n_fil){
      suma_col[j] <- suma_col[j] + matriz[i,j]
    }
  }
  suma_col
}
```

```
> m <- matrix(1:20, 4,5)
> suma_cols(m)
[1] 10 26 42 58 74
```

Otra versión:

```
suma_cols <- function(matriz){
  n_col <- ncol(matriz)
  suma_col <- numeric(n_col)
  for(j in 1:n_col){
    suma_col[j] <- sum(matriz[,j])
  }
  suma_col
}
```

```
> m <- matrix(1:20, 4,5)
> suma_cols(m)
[1] 10 26 42 58 74
```

¿Es mejor, nos *ahorramos* un ciclo?
R incluye las funciones *colSums*, *rowSums*, *colMeans* y *rowMeans*.

1.5.3 ¿Qué sucede si llamo un elemento igual a otro existente?

R tiene espaciones de nombres distintos para funciones y para otros elementos, esto quiere decir que se puede definir un objeto llamado c y no habrá problemas con la función $c()$:

```
> c <- 10
> c
[1] 10

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

¿Pero qué sucede si escribo una función con el nombre $c()$?

```
c <- function(){
  print('mismo nombre!')
}
```

```
> c()
[1] "mismo nombre!"
> c(1,2,3)
```

¿Ya no podemos crear más vectores?
Sí, pero debemos poner atención a los ambientes; por omisión, R utiliza los elementos del llamado *.GlobalEnv* que es el del usuario y al final el del ambiente *base*, para saber todos los ambientes disponibles al momento, se puede utilizar *search*().
Si se desea utilizar una función de algún ambiente particular, se utiliza la sentaxis *paquete*::*funcion*; por ejemplo:

```
> base::c(1,2,3)
[1] 1 2 3
```

1.6 Lectura y escritura

Una operación muy importante en R es la lectura de datos que pueden provenir de distintas fuentes y en diversos formatos. R incluye varias funciones para leer datos, entre otras:

- ♢ *read.table, read.csv*
- ♢ *readLines*
- ♢ *source*
- ♢ *dget*
- ♢ *load*
- ♢ *unserialize*

read.table es la más usada comúnmente para leer datos, por ejemplo, *read.csv, read.csv2, read.fortran, read.socket* son formas especializadas de *read.table* que ya tienen parámetros preestablecidos.

Los parámetros más importantes son:

- ♢ *file*: nombre del archivo que almacena los datos, puede ser una URL
- ♢ *header*: valor lógico que indica si el archivo contiene los nombres de las columnas
- ♢ *sep*: indica el separador de valores en las filas
- ♢ *colClasses*: vector que indica el tipo de datos que se espera en cada columna
- ♢ *nrows*: número máximo de filas a leer
- ♢ *skip*: número de filas de datos que serán ignoradas antes de comenzar la lectura
- ♢ *comment.char*: el carácter que se indicará filas de comentarios y que serán ignoradas en la lectura

Por default, *read.table* ignorará las líneas que comienzan con "#", supondrá que no hay renglón para encabezados, determinará el número de filas del archivo e inferirá el tipo de datos que tiene cada columna.

```
> read.table('datos.txt')
> datos
           V1          V2            V3            V4
1 FamilyNames FamilyAges FamilyGenders FamilyWeights
2         Dad          43          Male           188
3         Mom          42        Female           136
4         Sis          12        Female            83
5         Bro           8          Male            61
6         Dog           5        Female            44
```

Aunque siempre es recomendable indicar la mayor cantidad de parámetros posible en la lectura. Por ejemplo, nuestros datos incluyen encabezado:

```
> datos <- read.table('datos.txt', header=TRUE)
> datos
  FamilyNames FamilyAges FamilyGenders FamilyWeights
1         Dad         43          Male           188
2         Mom         42        Female           136
3         Sis         12        Female            83
4         Bro          8          Male            61
5         Dog          5        Female            44

> class(datos$FamilyNames)
[1] "factor"
> class(datos$FamilyAges)
[1] "integer"
```

Los nombres no son un *factor*, es mejor que sean de tipo *character*:

```
> cols <- c('character','integer','factor','integer')
> datos <- read.table('datos.txt', header=TRUE, colClasses=cols)
> class(datos$FamilyNames)
[1] "character"
```

Además, R también incluye funciones para escritura:

- ♢ *write.table*, *write.csv*
- ♢ *writeLines*
- ♢ *dump*
- ♢ *dput*
- ♢ *save*
- ♢ *serialize*

Una forma simple de escribir y leer datos propios es con *dump* y *source*:

```
> x <- 'cad'
> y <- data.frame(a=2, b=1+1i)
> dump(c('x','y'), file='ejemplo.txt')
> rm(x,y) # elimina los objetos del entorno actual
> source('ejemplo.txt')
> x
[1] "cad"
> y
  a    b
1 2 1+1i
```

Por supuesto que también se puede escribir información con *write.table* o *write.csv*; su uso es similar a sus contrapartes de lectura.

Por ejemplo, podemos añadir un renglón a los datos familiares y posteriormente volver a almacenarlos como texto:

```
> cat <- c('Cat',2,'Male',10)
> datos <- rbind(datos,cat)
> write.table(datos, file='datos2.txt', quote=FALSE, row.names=FALSE)
```

Debe ponerse atención en los parámetros *quote* y *row.names*.

1.7 Fecha y hora

R tiene clasese especiales para estos dos tipos de objetos, estos tipos permiten realizar operaciones aritméticas y estadísticas.

1.7.1 POSIXct

Es un tipo de dato que almacena fecha y hora como el número de segundos transcurridos desde el 1 de septiembre de 1970; número negativos representan los segundos antes de dicha fecha. También existe el tipo POSIXlt.

Para obtener el tiempo actual se utiliza la función *Sys.time*( ); las funciones *as.POSIXct*( ) y *as.POSIXlt*( ) sirven para covertir un valor de tiempo al formato correspondiente:

```
> t <- Sys.time()
> class(t)
[1] "POSIXct" "POSIXt"
 > typeof(t)
[1] "double"
> cat(t)
1564822046
> t
[1] "2019-08-03 03:47:26 CDT"
```

```
> c <- as.POSIXct(t)
> typeof(c)
[1] "double"
> c
[1] "2019-08-03 03:47:26 CDT"
> cat(c)
1564822046
```

En ocasiones se requiere escribir un objeto de tiempo con un formato distinto alq ue utiliza R, por ejemplo por compatibilidad con algún otro sistema; para esto se puede usar:

```
> tiempo <- strftime(t, '%Y-%m-%d %H:%M:%S')
> tiempo
[1] "2019-08-03 03:47:26"
```

Si se necesita hacerlo en sentido inverso:

```
> tiempo <- '2019-08-03'
> typeof(tiempo)
[1] "character"
> t <- strptime(t, '%Y-%m-%d')
> t
[1] "2019-08-03 CDT"
> t <- as.POSIXct(t)
> cat(t)
1564808400
```

Es posible realizar operaciones entre fechas:

```
> time1 <- '2017-08-03 10:25:14'
> t1 <- strptime(time1, '%Y-%m-%d %H:%M:%S')
> t2 <- Sys.time()
> t2-t1
Time difference of 729.7237 days
```

Una vez que un dato tipo tiempo se almacena en el formato interno de R se pueden realizar operaciones y comparaciones de manera fácil:

```
> dif <- as.difftime('00:30:00', '%H:%M:%S')
> dif
Time difference of 30 mins
> ahora <- Sys.time()
> al_rato <- ahora + dif
> ahora < al_rato
[1] TRUE
```

1.7.2 Tipo *Date*

Adicionalmente, R incluye el tipo de dato *Date*, la diferencia con tiempo es que este tipo lleva la cuenta de la diferencia de días en vez de segundos, se puede hacer un *casting* de una cadena de caracteres en fecha usando la función *as.Date*( ):

```
> fecha1 <- as.Date('2014-01-01')
> fecha2 <- as.Date('2021-08-23')
> fecha2-fecha1
Time difference of 2042 days
```

Para obtener la fecha del sistema se utiliza *Sys.Date*():

```
> ini <- as.Date('1980/02/05')
> hoy <- Sys.Date()
> hoy-ini
Time difference of 14435 days
```

Una función muy útil es la función *format*( ); permite modificar la salida de manera que se muestren en el orden que nosotros queremos:

```
> tiempo <- Sys.time()
> format(tiempo, '%H:%M')
[1] "21:57"
```

```
> fecha <- as.Date('2021-08-23')
> format(fecha, '%y/%m/%d')
[1] "19/08/05"
```

1.8 Gráficas básicas

La función más básica para traficar es la función *plot*( ), ésta forma la base de una gráfica a partir de la cual se pueden construir gráficas más complejas añadiendo líneas, puntos, leyendas, etc.

plot( ) es una función genérica, es decir que la función que lleva a cabo depende del tipo de objeto con el que es llamado. Por ejemplo, puede recibir dos vectores:

```
> plot(c(1,2,3,4), c(5,2,6,1))
```

Se puede cambiar la marca de puntos con:

```
> plot(c(1,2,3,4), c(5,2,6,1), pch='x')
```

Etiquetas en los ejes:

```
> plot(c(1,2,3,4), c(5,2,6,1), pch='*', xlab='X', ylab='Y')
```

Una vez creada una gráfica con *plot*, se pueden añadir elementos:

```
> plot(c(1,2,3,4), c(5,2,6,1), pch='*', xlab='X', ylab='Y')
> points(c(1,4), c(1,5))
> lines(c(1,4), c(1,5), col='red')
```

Como se mencionó, *plot* es una función genérica, por lo que también puede graficar un *dataframe* completo, por ejemplo iris:

```
> plot(iris)
```

O indicar las columnas deseadas:

```
> plot(iris$Sepal.Length, iris$Sepal.Width)
```