Notas de R

Roberto Álvarez

2022-08-11

Contents

Chapter 1

Acá empieza

Chapter 2

Introducción a R

R es un elenguaje de programación. Además es un entorno integrado para el manejo de datos, el cálculo, la generación de gráficos y análisis estadísticos. Las principales ventajas del uso de R son:

1. Es software libre.
2. Su facilidad para el manejo y almacenamiento de datos.
3. Es un conjunto de operadores para el cálculo de vectores y matrices.
4. Es una colección extensa e integrada de herramientas intermedias para el análisis estadístico de datos.
5. Posee un multitud de facilidades gráficas de altísima calidad
6. Es un lenguaje de programación (muy) poderoso con muchas librerías , bibliotecas más porpiamente dicho, especializadas disponibles.
7. La mejor herramienta para trabajar con datos genómicos, proteomicos, redes, metabolómica, entre varias más.
8. **Casi todos podemos aprender por nuestra cuenta a usar excel (pero hay que pagar por la licencia...), sin embargo es más díficil aprender por nuestra cuenta R; y si lo hacemos nos da una ventaja sobre el resto.**

2.1 Paquetes o bibliotecas

Las funciones especializadas de R se guardan en bibliotecas *(packages)* que deben ser invocados ANTES de llamar a una función de la biblioteca

Una manera de instalar bibliotecas es mediante el repositorio por defecto de R que es CRAN.

Navega por CRAN y encuentra algunos paquetes que podrían interesarte. Hay miles y cada día aumentan.

Para saber qué paquetes se tienen instalados en tu máquina teclea la función library()

```
library()
```

Para cargar un paquete en particular deben teclear, siempre y cuando ya esté instalado

```
library(nombre_de_paquete)
```

Por ejemplo

```
library(gplots)
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

Para visualizar los paquetes ya cargados tecleamos

```
search()
```

```
##  [1] ".GlobalEnv"        "package:gplots"    "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

Para visualizar las funciones dentro de un paquete en particular se utiliza

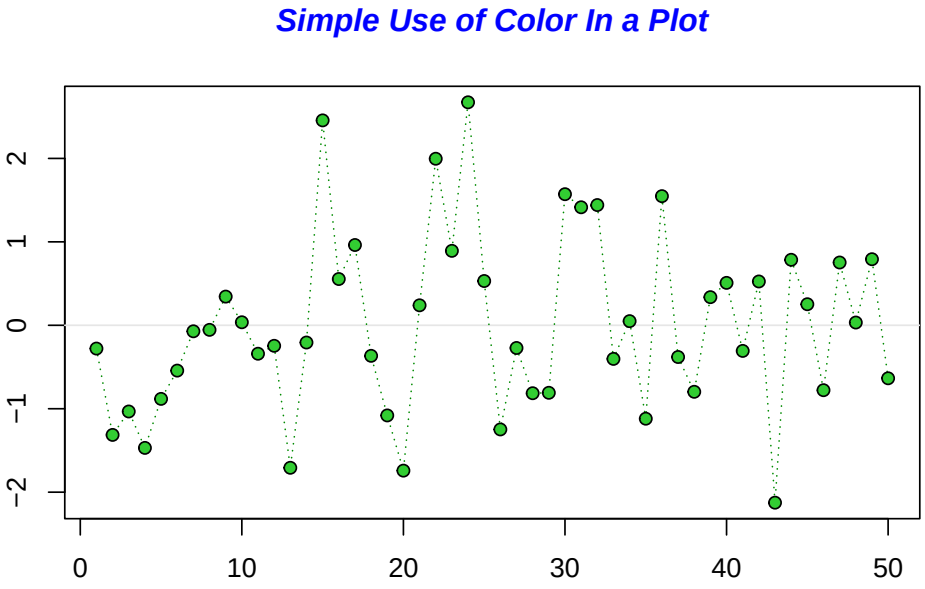
```
ls(2)
```

```
##  [1] "angleAxis"     "balloonplot"    "bandplot"      "barplot2"
##  [5] "bluered"        "boxplot2"       "ci2d"           "col2hex"
##  [9] "colorpanel"     "greenred"       "heatmap.2"      "hist2d"
## [13] "lmplot2"       "lowess"        "ooplot"        "overplot"
## [17] "panel.overplot" "plot.venn"    "plotCI"       "plotLowess"
## [21] "plotmeans"     "qqnorm.aov"    "redblue"       "redgreen"
## [25] "reorder.factor" "residplot"    "rich.colors"   "sinkplot"
## [29] "smartlegend"    "space"          "textplot"       "venn"
## [33] "wapply"
```

```
demo(graphics)
```

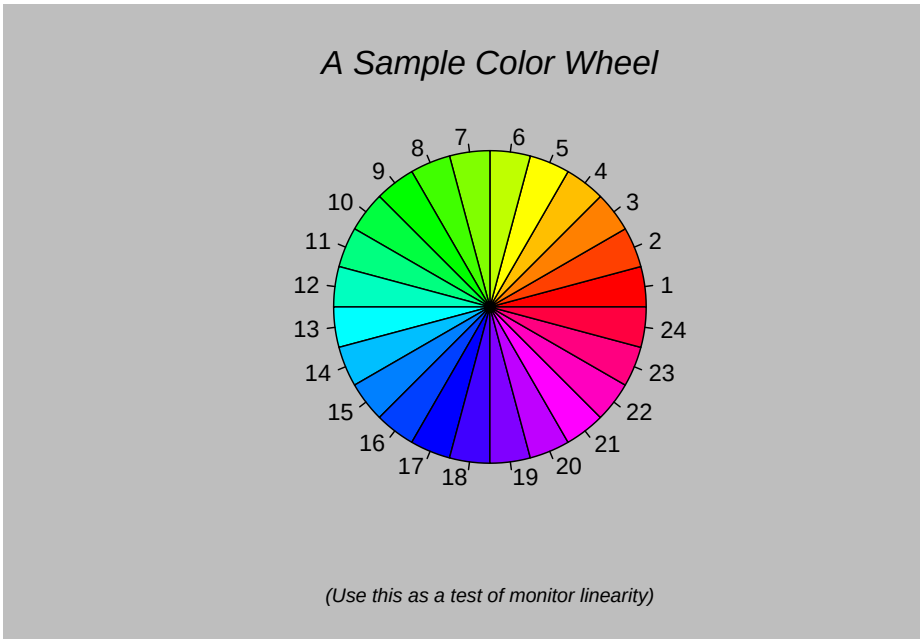
```
##
##
##  demo(graphics)
##  ---- ~~~~~~~~
##
## > #  Copyright (C) 1997-2009 The R Core Team
```

```
## >
## > require(datasets)
##
## > require(grDevices); require(graphics)
##
## > ## Here is some code which illustrates some of the differences between
## > ## R and S graphics capabilities.  Note that colors are generally specified
## > ## by a character string name (taken from the X11 rgb.txt file) and that line
## > ## textures are given similarly.  The parameter "bg" sets the background
## > ## parameter for the plot and there is also an "fg" parameter which sets
## > ## the foreground color.
## >
## >
## > x <- stats::rnorm(50)
##
## > opar <- par(bg = "white")
##
## > plot(x, ann = FALSE, type = "n")
```



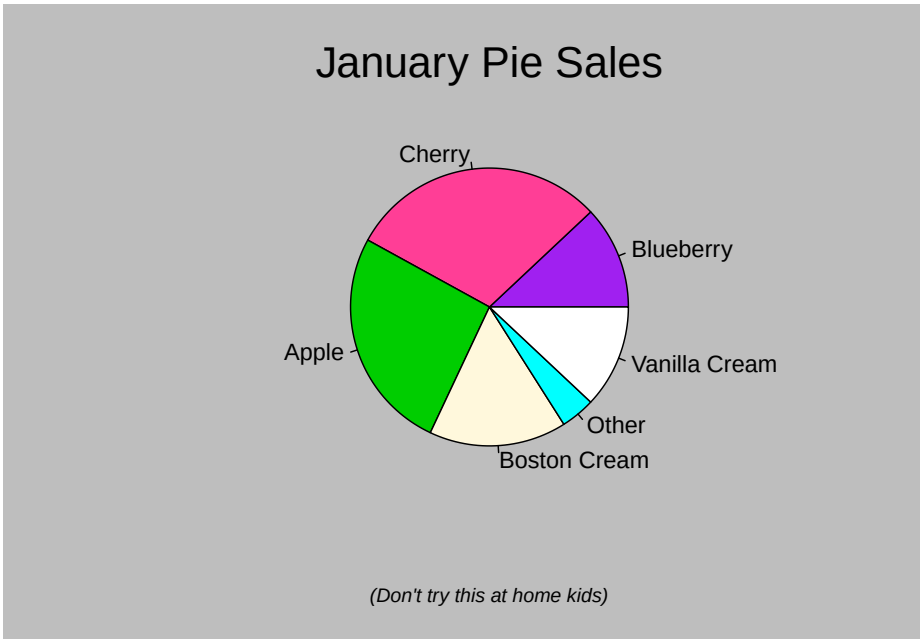

```
##
## > abline(h = 0, col = gray(.90))
##
## > lines(x, col = "green4", lty = "dotted")
##
## > points(x, bg = "limegreen", pch = 21)
```

```
##
## > title(main = "Simple Use of Color In a Plot",
## +       xlab = "Just a Whisper of a Label",
## +       col.main = "blue", col.lab = gray(.8),
## +     cex.main = 1.2, cex.lab = 1.0, font.main = 4, font.lab = 3)
##
## > ## A little color wheel.   This code just plots equally spaced hues in
## > ## a pie chart.   If you have a cheap SVGA monitor (like me) you will
## > ## probably find that numerically equispaced does not mean visually
## > ## equispaced.  On my display at home, these colors tend to cluster at
## > ## the RGB primaries.  On the other hand on the SGI Indy at work the
## > ## effect is near perfect.
## >
## > par(bg = "gray")
##
## > pie(rep(1,24), col = rainbow(24), radius = 0.9)
```



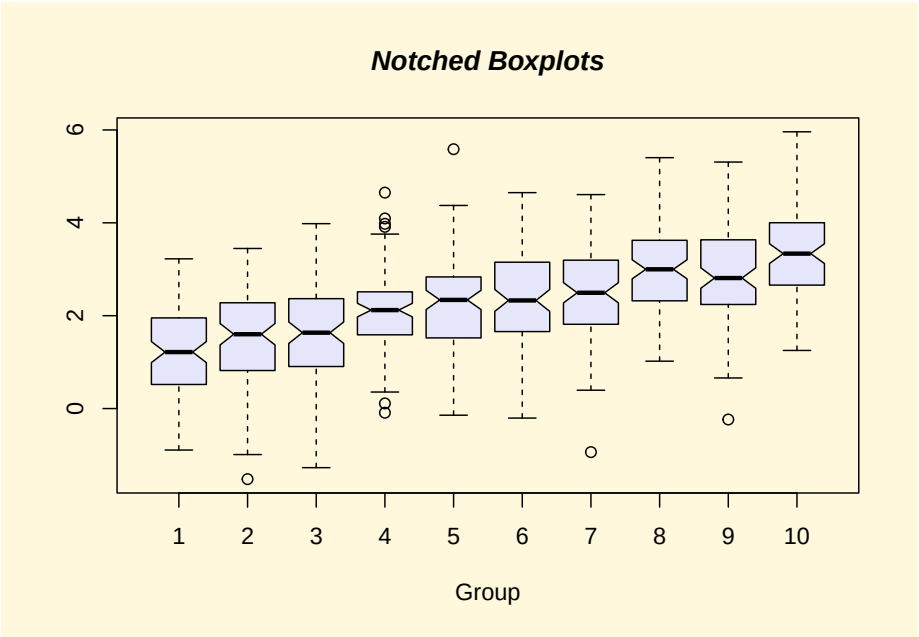

```
##
## > title(main = "A Sample Color Wheel", cex.main = 1.4, font.main = 3)
##
## > title(xlab = "(Use this as a test of monitor linearity)",
## +       cex.lab = 0.8, font.lab = 3)
##
## > ## We have already confessed to having these.  This is just showing off X11
## > ## color names (and the example (from the postscript manual) is pretty "cute".
```

```
## >
## > pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
##
## > names(pie.sales) <- c("Blueberry", "Cherry",
## +             "Apple", "Boston Cream", "Other", "Vanilla Cream")
##
## > pie(pie.sales,
## +   col = c("purple","violetred1","green3","cornsilk","cyan","white"))
```

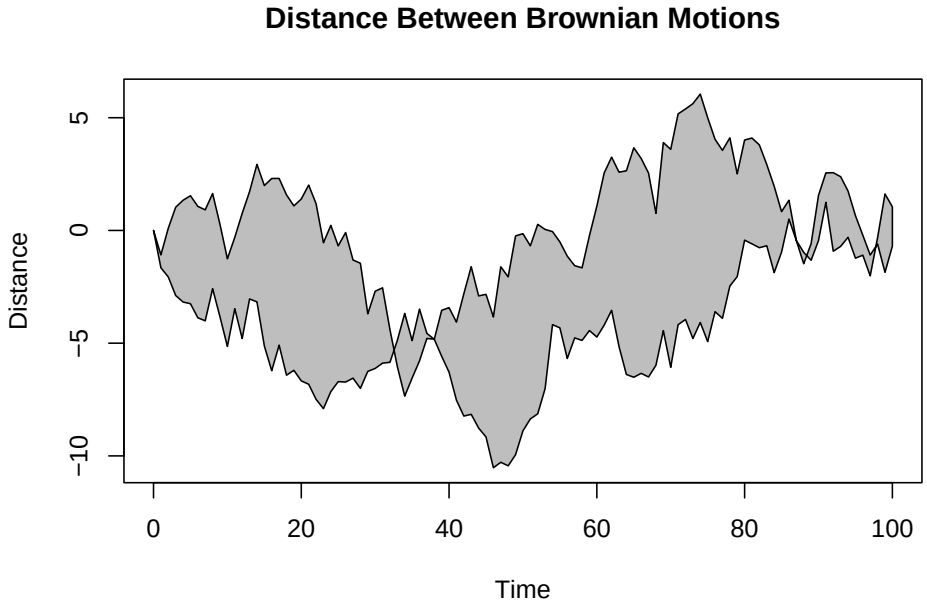



```
##
## > title(main = "January Pie Sales", cex.main = 1.8, font.main = 1)
##
## > title(xlab = "(Don't try this at home kids)", cex.lab = 0.8, font.lab = 3)
##
## > ## Boxplots:  I couldn't resist the capability for filling the "box".
## > ## The use of color seems like a useful addition, it focuses attention
## > ## on the central bulk of the data.
## >
## > par(bg="cornsilk")
##
## > n <- 10
##
## > g <- gl(n, 100, n*100)
##
## > x <- rnorm(n*100) + sqrt(as.numeric(g))
```

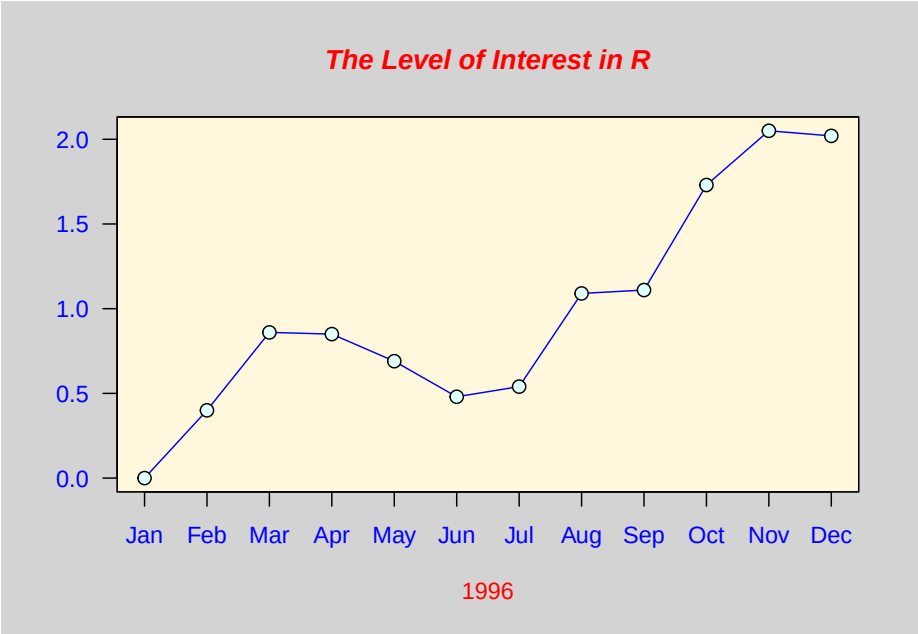
```
##
## > boxplot(split(x,g), col="lavender", notch=TRUE)
```




```
##
## > title(main="Notched Boxplots", xlab="Group", font.main=4, font.lab=1)
##
## > ## An example showing how to fill between curves.
## >
## > par(bg="white")
##
## > n <- 100
##
## > x <- c(0,cumsum(rnorm(n)))
##
## > y <- c(0,cumsum(rnorm(n)))
##
## > xx <- c(0:n, n:0)
##
## > yy <- c(x, rev(y))
##
## > plot(xx, yy, type="n", xlab="Time", ylab="Distance")
```

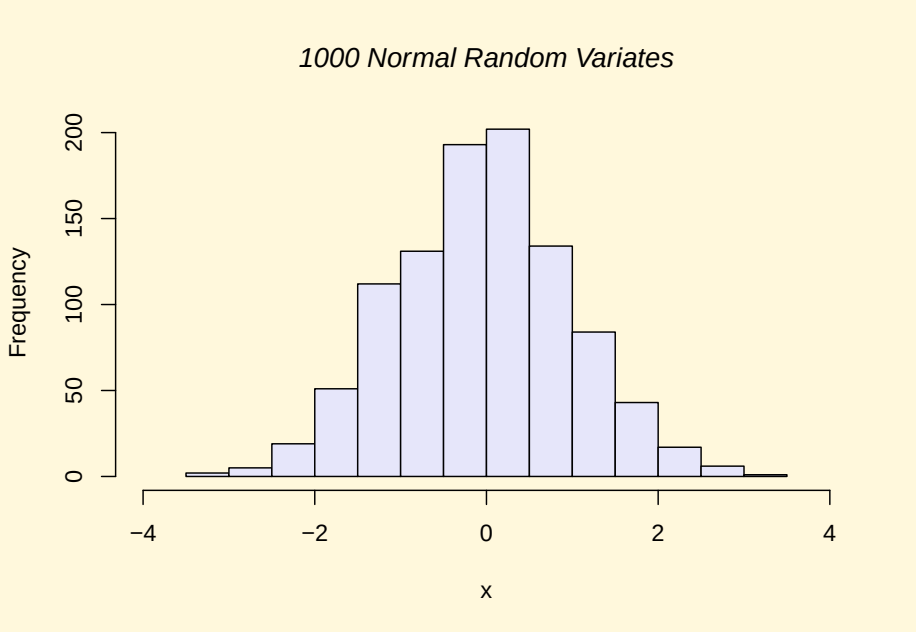



```
##
## > polygon(xx, yy, col="gray")
##
## > title("Distance Between Brownian Motions")
##
## > ## Colored plot margins, axis labels and titles.   You do need to be
## > ## careful with these kinds of effects.   It's easy to go completely
## > ## over the top and you can end up with your lunch all over the keyboard.
## > ## On the other hand, my market research clients love it.
## >
## > x <- c(0.00, 0.40, 0.86, 0.85, 0.69, 0.48, 0.54, 1.09, 1.11, 1.73, 2.05, 2.02)
##
## > par(bg="lightgray")
##
## > plot(x, type="n", axes=FALSE, ann=FALSE)
```

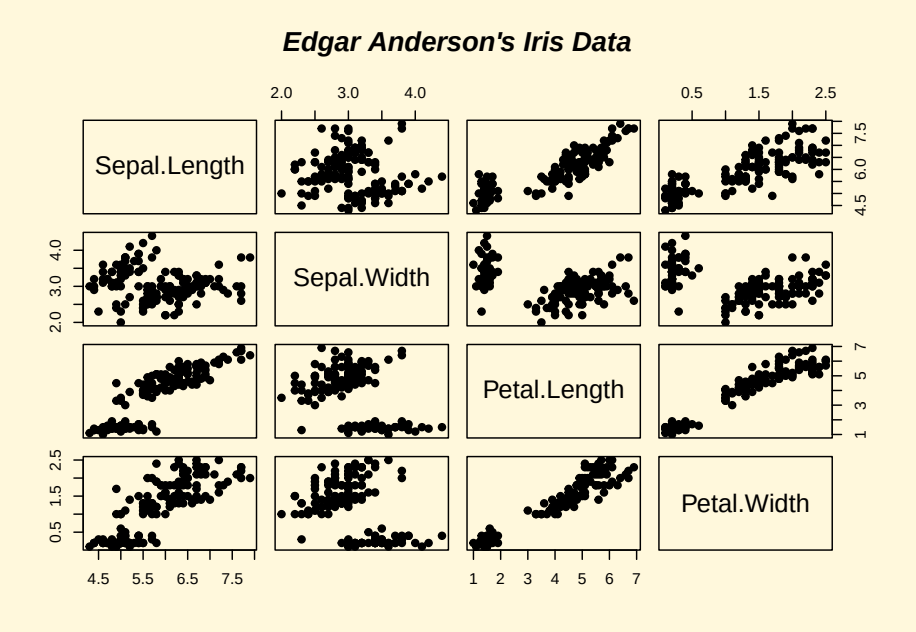



```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="cornsilk", border="black")
##
## > lines(x, col="blue")
##
## > points(x, pch=21, bg="lightcyan", cex=1.25)
##
## > axis(2, col.axis="blue", las=1)
##
## > axis(1, at=1:12, lab=month.abb, col.axis="blue")
##
## > box()
##
## > title(main= "The Level of Interest in R", font.main=4, col.main="red")
##
## > title(xlab= "1996", col.lab="red")
##
## > ## A filled histogram, showing how to change the font used for the
## > ## main title without changing the other annotation.
## >
## > par(bg="cornsilk")
##
## > x <- rnorm(1000)
```

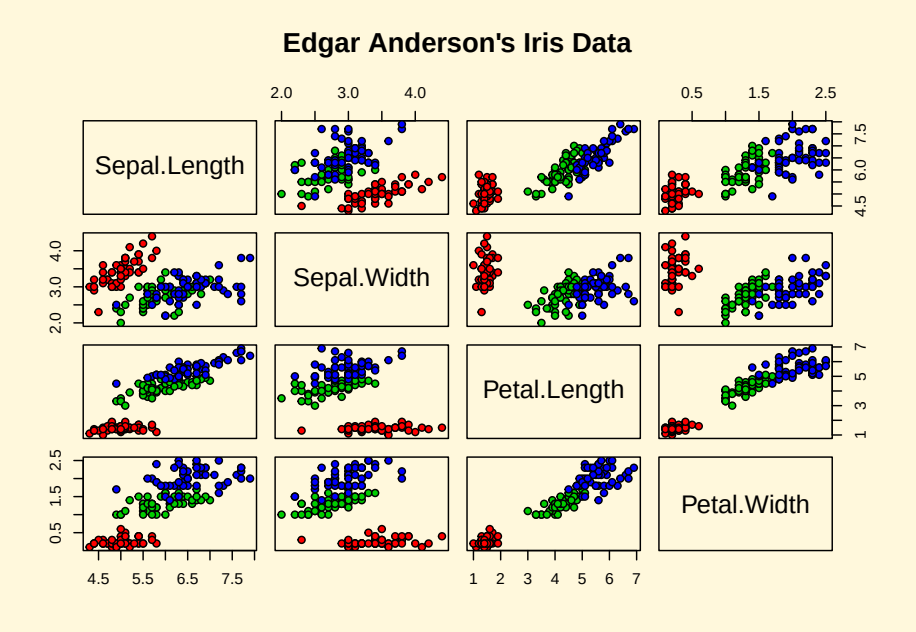
```
##
## > hist(x, xlim=range(-4, 4, x), col="lavender", main="")
```




```
##
## > title(main="1000 Normal Random Variates", font.main=3)
##
## > ## A scatterplot matrix
## > ## The good old Iris data (yet again)
## >
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", font.main=4, pch=19)
```

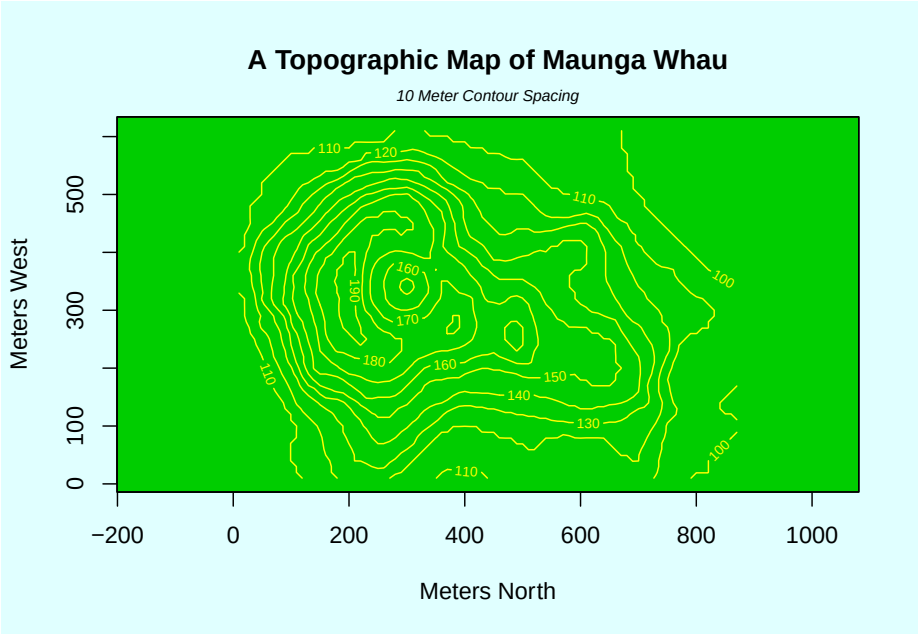



```
##
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", pch=21,
## +       bg = c("red", "green3", "blue")[unclass(iris$Species)])
```

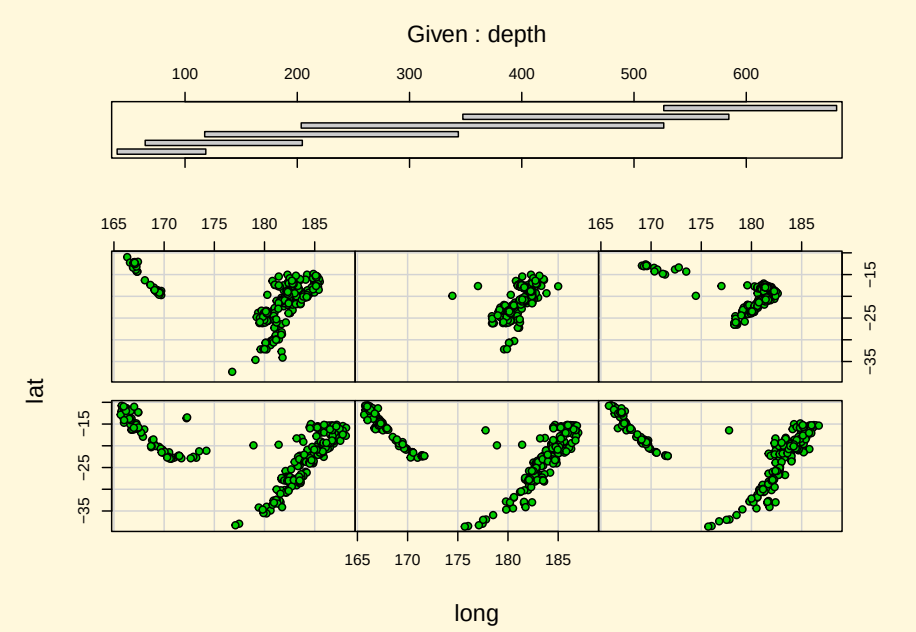



```
##
```

```
## > ## Contour plotting
## > ## This produces a topographic map of one of Auckland's many volcanic "peaks".
## >
## > x <- 10*1:nrow(volcano)
##
## > y <- 10*1:ncol(volcano)
##
## > lev <- pretty(range(volcano), 10)
##
## > par(bg = "lightcyan")
##
## > pin <- par("pin")
##
## > xdelta <- diff(range(x))
##
## > ydelta <- diff(range(y))
##
## > xscale <- pin[1]/xdelta
##
## > yscale <- pin[2]/ydelta
##
## > scale <- min(xscale, yscale)
##
## > xadd <- 0.5*(pin[1]/scale - xdelta)
##
## > yadd <- 0.5*(pin[2]/scale - ydelta)
##
## > plot(numeric(0), numeric(0),
## +     xlim = range(x)+c(-1,1)*xadd, ylim = range(y)+c(-1,1)*yadd,
## +       type = "n", ann = FALSE)
```




```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="green3")
##
## > contour(x, y, volcano, levels = lev, col="yellow", lty="solid", add=TRUE)
##
## > box()
##
## > title("A Topographic Map of Maunga Whau", font= 4)
##
## > title(xlab = "Meters North", ylab = "Meters West", font= 3)
##
## > mtext("10 Meter Contour Spacing", side=3, line=0.35, outer=FALSE,
## +       at = mean(par("usr")[1:2]), cex=0.7, font=3)
##
## > ## Conditioning plots
## >
## > par(bg="cornsilk")
##
## > coplot(lat ~ long | depth, data = quakes, pch = 21, bg = "green3")
```




```
##
## > par(opar)
```

2.2 Ayuda en R

help(nombre_comando) o ?nombre_comando

```
help(solve)
```

```
?solve
```

son equivalentes para buscar ayuda sobre el comando solve

Para buscar ayuda de funciones o palabra reservadas se utilizan comillas

```
help("for")
```

Para abrir la ayuda genral en un navegador (sólo si tenemos la ayuda en HTML instalada y tenemos conexión a la red) se utiliza

```
help.start()
```

```
## starting httpd help server ... done
```

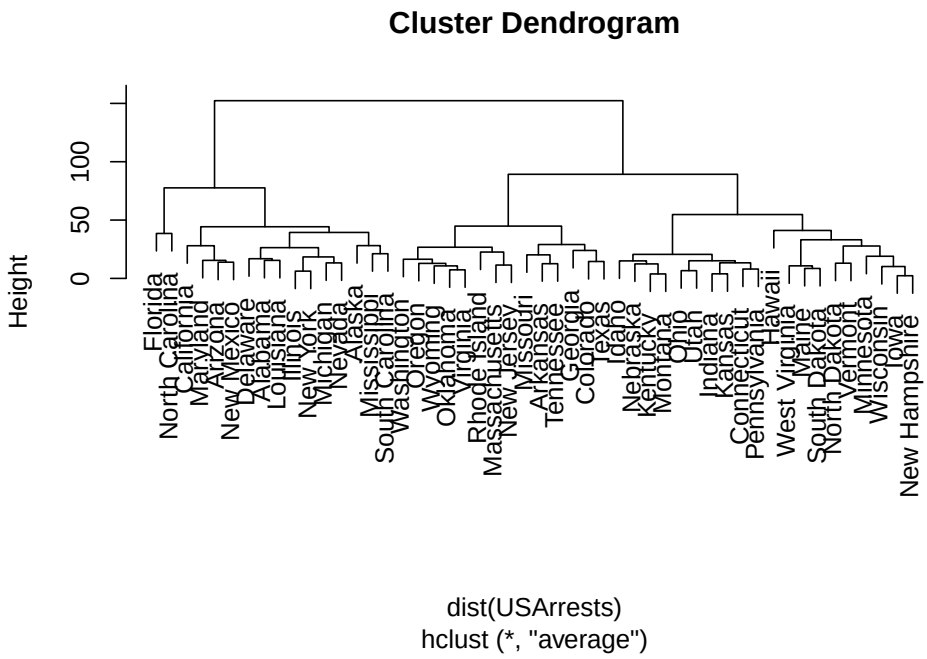
```
## If the browser launched by '/usr/bin/open' is already running, it is
##     *not* restarted, and you must switch to its window.
## Otherwise, be patient ...
```

```
help.search("clustering")
```

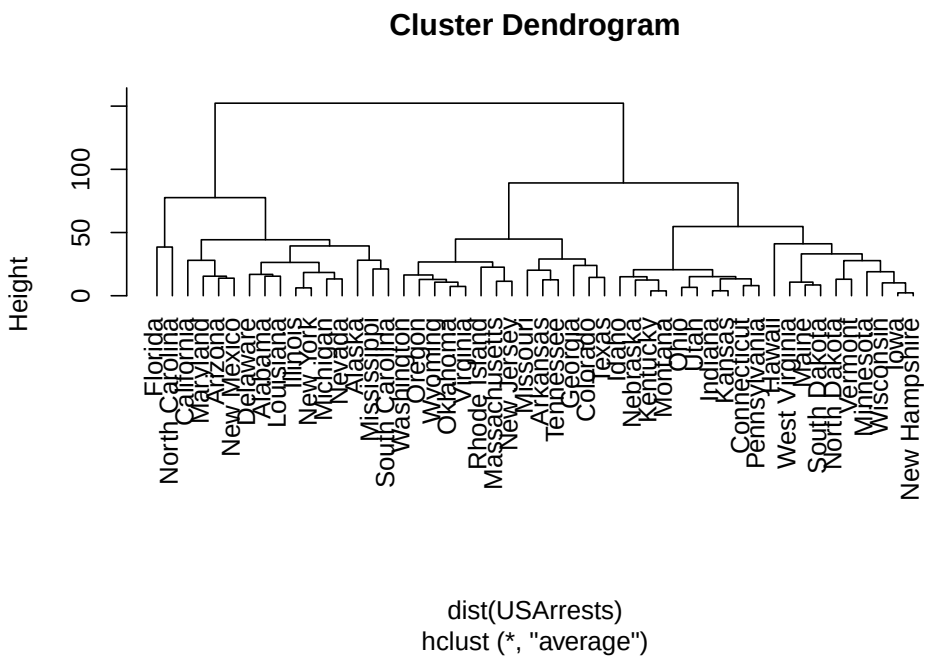
Si queremos ver ejemplos del uso de los comandos usamos la función ejemplo

```
example("hclust")
```

```
##
## hclust> require(graphics)
##
## hclust> ### Example 1: Violent crime rates by US state
## hclust>
## hclust> hc <- hclust(dist(USArrests), "ave")
##
## hclust> plot(hc)
```

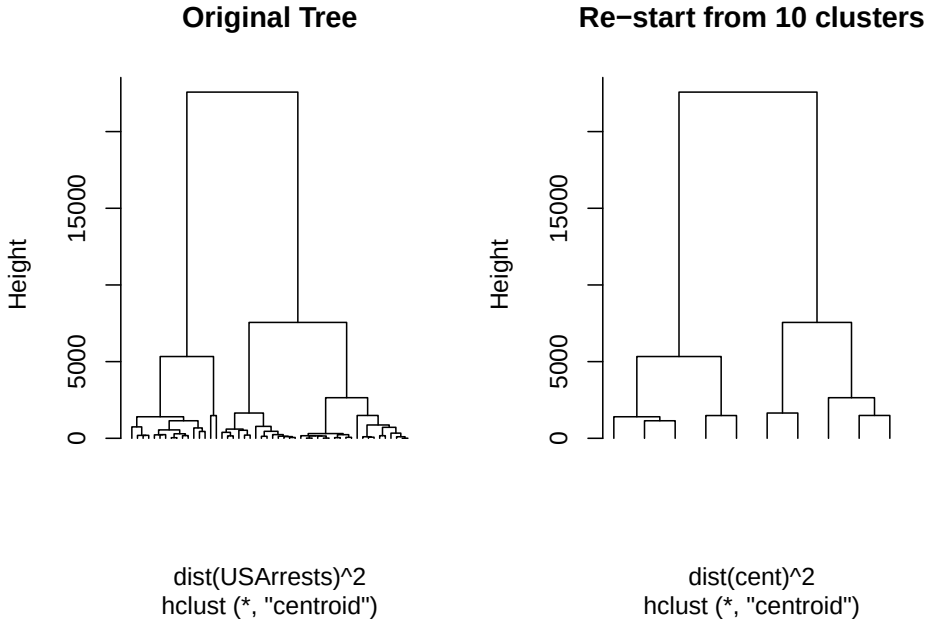



```
##
## hclust> plot(hc, hang = -1)
```




```
##
## hclust> ## Do the same with centroid clustering and *squared* Euclidean distance,
## hclust> ## cut the tree into ten clusters and reconstruct the upper part of the
## hclust> ## tree from the cluster centers.
## hclust> hc <- hclust(dist(USArrests)^2, "cen")
##
## hclust> memb <- cutree(hc, k = 10)
##
## hclust> cent <- NULL
##
## hclust> for(k in 1:10){
## hclust+  cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
## hclust+ }
##
## hclust> hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
##
## hclust> opar <- par(mfrow = c(1, 2))
##
## hclust> plot(hc,  labels = FALSE, hang = -1, main = "Original Tree")


##
## hclust> plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
```

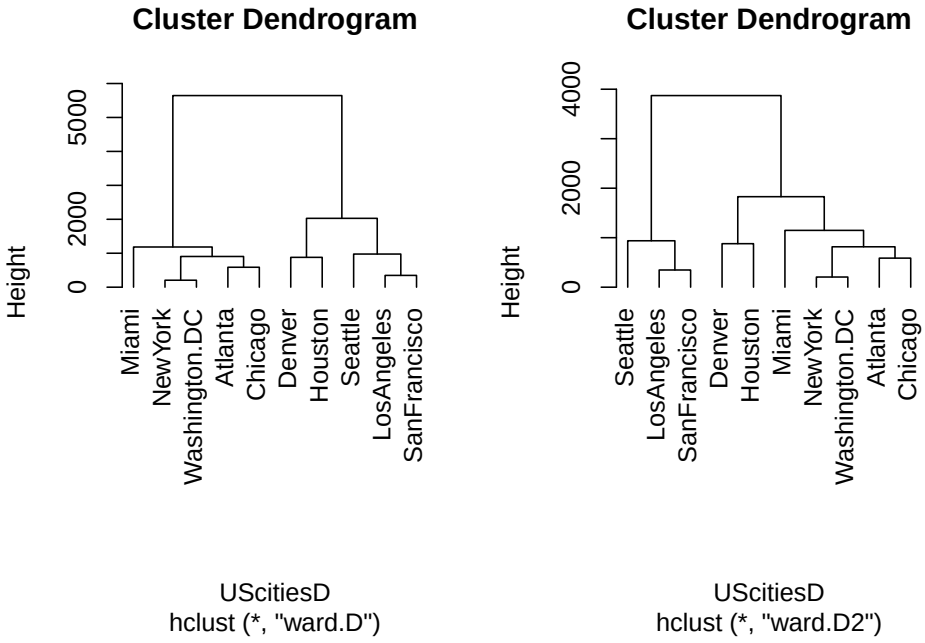



```
##
## hclust> par(opar)
##
## hclust> ### Example 2: Straight-line distances among 10 US cities
## hclust> ##  Compare the results of algorithms "ward.D" and "ward.D2"
## hclust>
## hclust> mds2 <- -cmdscale(UScitiesD)
##
## hclust> plot(mds2, type="n", axes=FALSE, ann=FALSE)
```


```
##
## hclust> text(mds2, labels=rownames(mds2), xpd = NA)
```

```
##
## hclust> hcity.D  <- hclust(UScitiesD, "ward.D") # "wrong"
##
## hclust> hcity.D2 <- hclust(UScitiesD, "ward.D2")
##
## hclust> opar <- par(mfrow = c(1, 2))
##
## hclust> plot(hcity.D,  hang=-1)

##
## hclust> plot(hcity.D2, hang=-1)
```




```
##
## hclust> par(opar)
```

Todo lo anterior requiere que conzocamos el nombre correcto del comando, pero ¿qué pasa si no lo sabemos? Podemos utilizar el comando `apropos()` para encontrar todo lo relacionado con algún término

```
apropos("solve")
```

```
## [1] "backsolve"     "forwardsolve"  "qr.solve"      "solve"
## [5] "solve.default" "solve.qr"
```

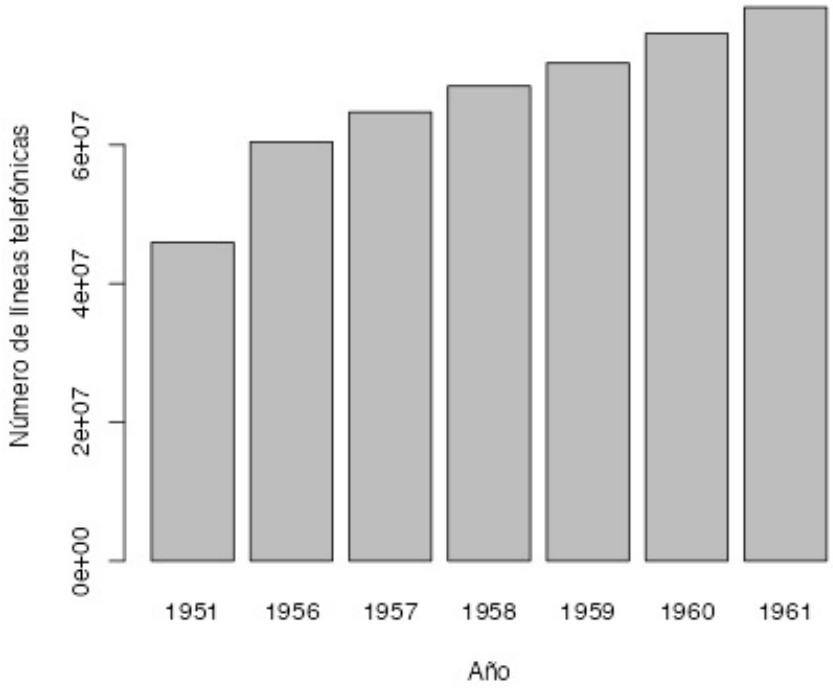
2.2.1 Sesiones interactivas

En R también podmeos generar elementos interactivos con la bilioteca **shiny**

```
##
## Listening on http://127.0.0.1:3354
```

Región:

N.Amer



2.3 Expresiones y asignaciones

Hay dos tipos de resultados en R: expresiones y asignaciones. Las primeras sólo se muestran a la salida estándar y NO se guardan en una variable; las segundas, se asignan y guardan en una variable

Expresión:

```
rnorm(10)
```

```
##  [1] -1.5722350  0.9601945 -0.1204486 -2.0042524  0.9381918 -0.1460520
##  [7]  1.3172614 -1.0901390 -1.0973922  0.1347918
```

Asignación

```
x <- rnorm(10)
```

```
x
```

```
##  [1] -0.3588845 -0.3139275 -0.5939401  2.0301381 -0.1833358  0.6998264
##  [7] -0.5824878 -0.5775879  0.9772605 -0.4070146
```

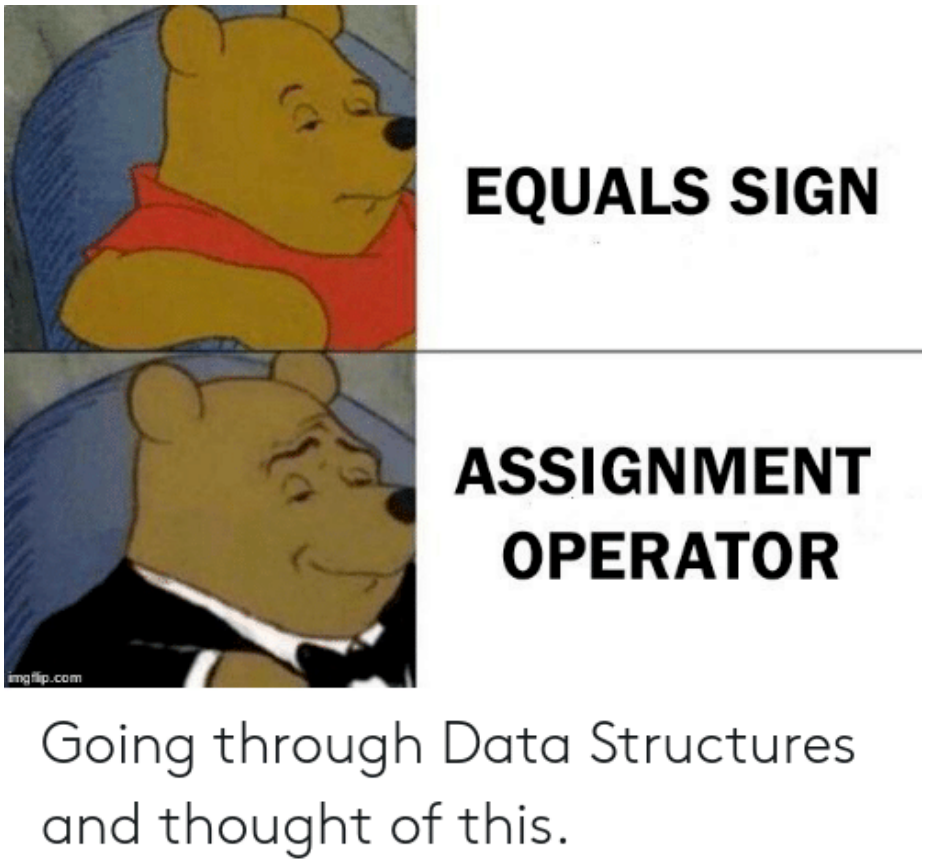


Figure 2.1: Operado de asignación. Evitar el uso del igual

R distingue entre mayúsculas y minúsculas, así las siguientes variables contienen valores distintos

```
a <- 3
```

```
A <- 6
```

Los comandos pueden separarse por ; o - mejor opción- por un salto de línea

```
a <- 3; b <-5
```

también pueden definirse asignaciones en más de una línea

```
a <-
  pi + 12
```

2.4 Movimiento entre directorios

Para saber en qué directorio estamos tecleamos

```
getwd()
```

```
## [1] "/Users/robertoalvarez/Dropbox/UAQ/Bioinformatica/Introduccion-a-R.github.io"
```

Para cambiar de directorio utilizamos `setwd("direccion_a_la_que_quieres_ir")`

```
setwd("~")
```

También podemos usar los comandos de bash dentro de R, utilizando la función `system()`

```
system("ls -la")
```

```
system("pwd")
```

2.5 Importante

Como regla general todos los nombres van entre comillas: nombre de carpetas, archivos, de columnas, de renglones,etc.

2.6 Operaciones aritméticas

Se puede sumar, restar, multiplicar,dividir, "exponenciar" y calcular la raíz cuadrada. Los operadores son, respectivamente: `+`,`-`,`*`, `/`,`**` o `^`, `sqrt()`

```
a + b
```

```
## [1] 20.14159
```

```
a - b
```

```
## [1] 10.14159
```

```
a * b
```

```
## [1] 75.70796
```

```
a ** b
```

```
## [1] 795898.7
```

```
a ^ b
```

```
## [1] 795898.7
```

```
sqrt(a)
```

```
## [1] 3.89122
```

2.7 Prioridad en las operaciones

Las operaciones se efectuan en el siguiente orden:

1. izquierda a derecha
2. sqrt() y ** , ^
3. "*" y /
4. "+" y -
5. <-

Este orden se altera si se presenta un paréntesis. En ese caso la operación dentro del paréntesis es la que se realiza primero.

Ejemplos

4 + 2 *3 = 4 + 6 = 10

4-15/3 +3^2 +sqrt(9)= 4-15/3 + 9 +3 = 4-5+12=13

4-(3+7)^2 + (2+3)/5=4-100+5/5=-95

2.7.1 Ejercicios

Resuelve en un pedazo de papel primero para saber cuál sería el resultado de las siguientes operaciones aritméticas. Después comprueba tu resultado tecleandolas en R

1. 1 + 2*3 +3 +15/3
2. 4-15/3 +3^2 +3*sqrt(81)
3. 40-(4+3)^2 + (10-5)/3

2.8 Tipos de datos lógicos o booleanos

Estos tipos de datos **sólo** contienen información `TRUE` o `FALSE`. Sirven para evaluar expresiones de $=$, $<$, $>$ y pueden utilizarse para obtener los elementos de un vector que cumplan con la característica deseada.

```
1 < 5
```

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

```
10 == 0  # Es igual a
```

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

```
10 != 0    # NO es igual a
```

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

```
10 <= 0    #Menor o igual
```

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

Dentro de R un valor lógico `TRUE` equivale a 1 y `FALSE` equivale a 0, por lo tanto para contar cuántos `TRUE`s hay podemos hacer una suma:

Ejercicio utiliza una sola líndea de R para averiguar si el logaritmo base de 10 de 20 es menor que la raiz cuadrada de 4.

2.8.1 Caracter

Son *strings* de texto. Se distingue porque los elementos van entre comillas (cada uno). Puede ser desde un sólo caracter hasta oraciones completas. Puede parecer que contienen números, pero las comillaa indican que serán tratados como texto. Podemos subsetearlos por su índice o buscando literalmente el texto.

```
x<- "La candente mañana de febrero en que Beatriz Viterbo murió, después de una imperi
```

2.8.2 Enteros y números (numeric)

R por default representa los números como `numeric`, NO `integer`. Estos tipos son dos formas diferentes en las cuales las computadoras pueden guardar los números y hacer operaciones matemáticas con ellos. Por lo común esto no importa, pero puede ser relevante para algunas funciones de Bioconductor, por ejemplo ya que el tamaño máximo de un `integer` en R es ligeramente más chico que el tamaño del genoma humano.

¿Cómo revisar si un objeto es numeric o entero?

```
x <- 1
class(x)
```

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

```
x <- 1:3
class(x)
```

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

2.9 # Vectores en R

En R puedes guardar muchos elementos del mismo tipo en un sólo objeto mediante vectores.

Un vector es una colección de datos del mismo tipo. Siempre del **mismo tipo**. No es posible mezclarlos.

Chapter 3

Vectores

3.1 Definición

Para definir un vector se utiliza la función c(), que siginfica *combine*

```
x <- c(1,2,6,90,76.7)
```

3.1.1 Longitud de un vector

Para obtener la longitud de un vector, es decir el número de elementos que tiene se utiliza la función `length()`

```
length(x)
```

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

Como siempre, para mostrar el contenido de una variable sólo es necesario poner la variable y presionar enter en la sesión interactiva, si se está en un *script* es necesario usar la función `print()`

3.2 Uso de la función combine `c()` y el operador `:`

Un vector se puede definir de forma extensiva, es decir poniendo explícitamente todos los valores del vector.

```
y<-c("esto","es","un","vector")
z<-c(1,10,100,1000)
```

Esto es muy poco eficiente a menos que los vectores sean muy pequeños por lo que existen funciones para generar algunos casos particulares. Por ejemplo, si queremos tener un vector que tenga los primeros 100 números enteros podemos definirlo de la siguiente manera con el uso del operador `:` :

```
x<-1:100;
x
```

```
##  [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
```

También se puede usar de forma equivalente la función `seq()` que significa sequence. `seq()` es una generalización del operador `:`,

```
x<-seq(1,100)
x
```

```
##  [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
```

con ella podemos generar secuencias numéricas de distintas clase y espaciadas por diferentes rangos. Por ejemplo si queremos tener una secuencia de -12 a 30 en pasos de 3, es decir -12, -9,-6,..., 27,30 Teclearíamos

```
x<-seq(from=-12,to=30,by=3)
x
```

```
##  [1] -12  -9  -6  -3   0   3   6   9  12  15  18  21  24  27  30
```

Podemos omitir los nombres from, to,by si usamos el mismo orden, si queremos intercambiarlo debemos necesariamente ponerlos

```
y<-seq(0,1,0.1)
y
```

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

```
z<-seq(by=0.1, to =1, from=0.5)
z
```

```
## [1] 0.5 0.6 0.7 0.8 0.9 1.0
```

3.3 Acceder a elementos de un vector

3.3.1 Elementos consecutivos de un vector

Para acceder a elemetnos de un objeto con índices en R debemos usar los corchetes `[]` para indicarle que queremos seleccionar esos objetos

```
x<-c("Muchos", "años" ,"después" ,",", "frente", "al" ,"pelotón")
x[1:4]
```

```
## [1] "Muchos"  "años"    "después" ","
```

‘

```
x<-c(1,2,3,5,8,13,21)
x[3:6]
```

```
## [1]  3  5  8 13
```

3.3.2 Elementos no consecutivos de un vector

Para seleccionar elementos no consecutivos definimos un nuevo vector (con la función `c()`) de índices que seleccionará los elementos que quieres

```
x<-c("Muchos", "años" ,"después" ,",", "frente", "al" ,"pelotón")
x[c(1,3,5,7)]
```

```
## [1] "Muchos"  "después" "frente"  "pelotón"
```

No es necesario que estén en orden

```
x<-c(1,2,3,5,8,13,21)
x[c(2, 7, 4)]
```

```
## [1]  2 21  5
```

3.3.3 Excluir elementos de un vector

Para seleccionar algunos elementos **excepto** un conjunto de ellos usamos el signo menos `-`

```
x<-c(1,2,3,5,8,13,21)
x[-2]
```

```
## [1]  1  3  5  8 13 21
```

```
# Todos menos el segundo elemento
x[-c(2, 7, 4)]   # Todos menos el segundo , séptimo y cuarto elemento
```

```
## [1]  1  3  8 13
```

```
x[-length(x)]    # ¿Esto qué hace?
```

```
## [1]  1  2  3  5  8 13
```

Este comando **no elimina** elementos de un vector sólo los selecciona

```
x<-c(1,2,3,5,8,13,21)
x[-6]
```

```
## [1]  1  2  3  5  8 21
```

```
x     # Estoy intacto
```

```
## [1]  1  2  3  5  8 13 21
```

3.4 Agregar y quitar elementos de un vector

```
x <- c(88,5,12,13)
x <- c(x[1:3],168,x[4])
x
```

```
## [1]  88   5  12 168  13
```

Podemos, incluso, definir un vector vacío y luego "llenarlo"

```
x<-c()
x                  # Soy un vector vacío :(
```

```
## NULL
```

```
x[1]<- 2
x[2:5]<-c(56,78,90,12)
x                  # Ahora ya no :)
```

```
## [1]  2 56 78 90 12
```

3.5 Repetición de elementos de un vector con `rep()`

La función `rep()` que viene del inglés *repeat* nos permite repetir elementos en un vector dado. Por ejemplo, `rep(x,n veces)`

```
x<-rep(3,5)
x
```

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

```
y<-rep(c(1,2,3,5),3)
y
```

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

```
primos<-c(1,2,3,5,7,11)
z<-rep(primos,4)
z
```

```
##  [1]  1  2  3  5  7 11  1  2  3  5  7 11  1  2  3  5  7 11  1  2  3  5  7 11
```

También podemos usar la opción `each` para indicar la frecuencia de repetición

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

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

3.6 Uso de funciones `any()` y `all()`

Las funciones `any()` y `all()` determinan si alguno o todos los elementos de un vector cumplen cierta condición respectivamente. La respuesta siempre será un valor booleano es decir: **TRUE** o **FALSE**

```
x<- 1:15
any(x > 7.5)
```

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

```
any(x > 19.76)
```

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

```
any(x >= 15)
```

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

```
all(x> sqrt(100))
```

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

```
all(x>0)
```

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

3.7 Operaciones con vectores

Al igual que en álgebra podemos definir varias operaciones que nos dejan siempre otro vector:

1. Suma (resta) de vectores
2. Producto de vectores (término a término)
3. Producto de un escalar por un vector

```
x<-c(1,2,3)
y<-c(4,5,6)
x + y
```

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

```
x-y
```

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

```
x*x
```

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

```
y*y
```

```
## [1] 16 25 36
```

```
x*y
```

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

```
3*x                #Multiplicación por escalar: término a término
```

```
## [1] 3 6 9
```

```
sqrt(2)*y          # Sí, por cualquier escalar!
```

```
## [1] 5.656854 7.071068 8.485281
```

```
3*x + sqrt(2)*y    # Operaciones más complejas
```

```
## [1]  8.656854 13.071068 17.485281
```

También podemos aplicar funciones para calcular con una sola instrucción varias operaciones útiles, por ejemplo `min()`, `max()`, `range()` , `sum()`, `mean()`, `median()`, `sd()`,`quantile()`,`unique()`,`sort()` Si tienes duda de qué hace alguna de ellas busca en la ayuda

```
x<-rnorm(1000)
min(x)
```

```
## [1] -3.359911
```

```
max(x)
```

```
## [1] 3.871783
```

```
range(x)
```

```
## [1] -3.359911  3.871783
```

```
sum(x)
```

```
## [1] 45.23038
```

```
mean(x)
```

```
## [1] 0.04523038
```

```
median(x)
```

```
## [1] 0.0339923
```

```
sd(x)
```

```
## [1] 0.9882187
```

```
quantile(x)
```

```
##          0%         25%         50%         75%        100%
## -3.3599110 -0.6131977  0.0339923  0.6845514  3.8717825
```

Para `unique()` y `sort()` conviene tener elementos discretos más que continuos

```
x<- c(rep(3,5),1:15,rep(c(1,2,3),5))
unique(x)
```

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

```
x<-sample(10,10)
x
```

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

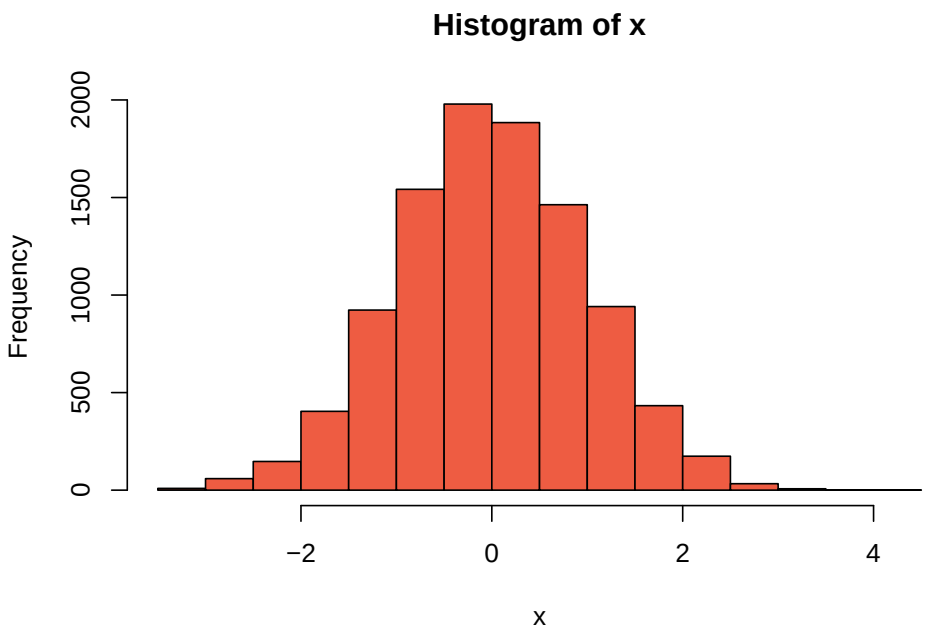
```
sort(x)
```

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

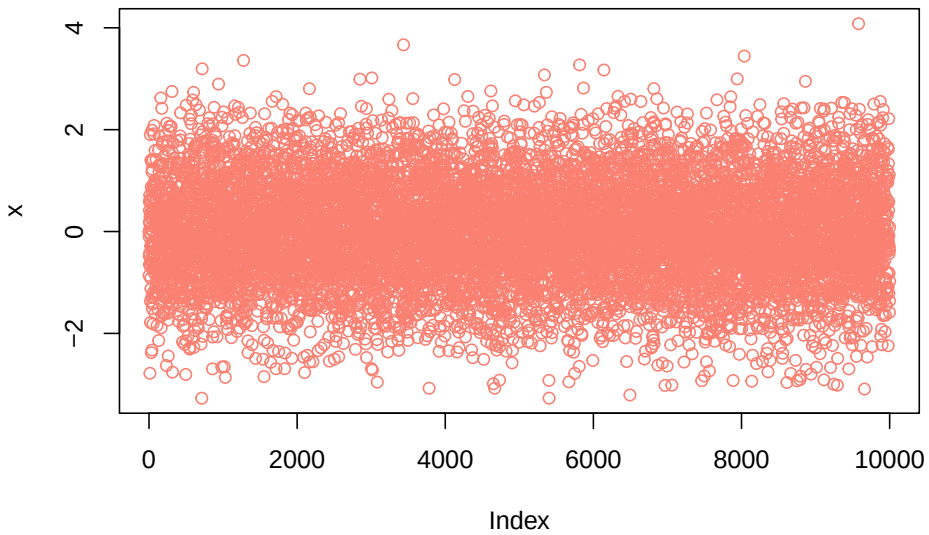
3.8 Gráficos con vectores

Podemos graficar los vectores de manera inmediata en `R`

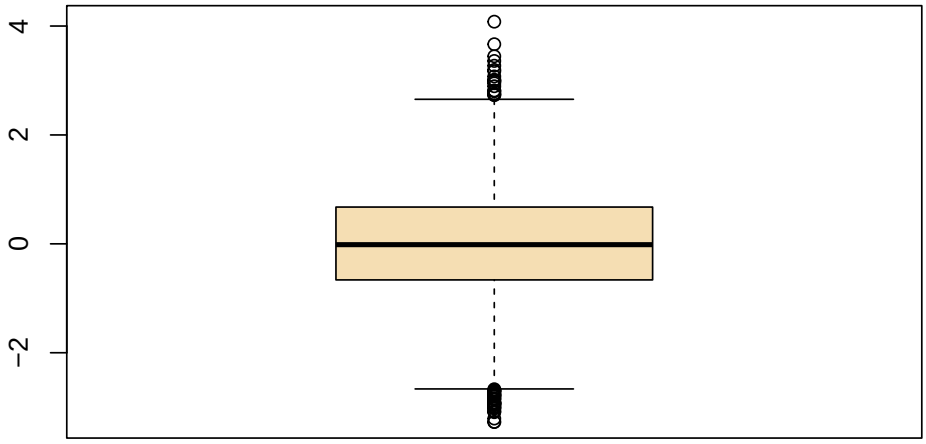
```
x<- rnorm(10000)
hist(x,col="tomato2")
```




```
plot(x,col="salmon")
```




```
boxplot(x,col="wheat")
```



3.9 Vectores con nombre

Definimos un vector llamado edades

```
edades<-c(35,35,70,17,14)
edades
```

```
## [1] 35 35 70 17 14
```

Podemos definir un vector del mismo tamaño que `edades` llamado nombres

```
nombres <-c("Jerry","Beth","Rick", "Summer","Morty")
nombres
```

```
## [1] "Jerry"  "Beth"   "Rick"   "Summer" "Morty"
```

Una de las cracterísticas de R es que podemos asignarles nombres a los vectores, para ello usamos la función `names()`

```
names(edades)<-nombres
```

Con ello ahora el vector edades tiene una nueva característica:

```
edades
```

```
##  Jerry   Beth   Rick Summer  Morty
##     35     35     70     17     14
```

Podemos seleccionar de la manera usual, por ejemplo, si quiero ver cuál es la edad de Rick, debo seleccionar el 3 elemento

```
edades[3]
```

```
## Rick
##   70
```

Esto es muy poco eficiente y propenso al error sobre todo con vectores muy grandes. Por ello podemos usar los nombres de los vectores

```
edades["Rick"]
```

```
## Rick
##   70
```

Recuerda que los nombres S-I-E-M-P-R-E van entre comillas

```
edades[c("Rick","Morty")]
```

```
##  Rick Morty
##    70    14
```

Ejercicios:

1. ¿Cuál es el promedio de las edades, sin contar el de Beth?
2. Quiten a Morty del vector, ordénenlo y guárdenlo como un nuevo objeto.
3. ¿Hay alguna edad que sea mayor de 75? ¿Menor de 12? ¿Entre 12 y 20?

3.9.1 Tamaños de genomas

Ahora veamos un ejemplo más "biológico"

```
genomeSizeM_BP<-c(3234.83,2716.97,143.73,0.014281,12.1)
```

Por ejemplo si quisierámos ver el tamño en bp simplemente multiplicamos por el valor del prefijo Mega = 1 millón

```
genomeSizeM_BP*1e6
```

```
## [1] 3234830000 2716970000  143730000      14281   12100000
```

```
organismo<-c("Human","Mouse","Fruit Fly","Roundworm","Yeast")
```

```
names(genomeSizeM_BP) <- organismo
```

```
genomeSizeM_BP
```

```
##       Human       Mouse   Fruit Fly   Roundworm       Yeast
## 3234.830000 2716.970000  143.730000    0.014281   12.100000
```

Se pueden seleccionar elementos de un vector utilizando corchetes

```
genomeSizeM_BP[1]
```

```
##   Human
## 3234.83
```

Para obtener elementos consecutivos

```
genomeSizeM_BP[1:4]
```

```
##       Human       Mouse   Fruit Fly   Roundworm
## 3234.830000 2716.970000  143.730000    0.014281
```

Para obtener elementos NO consecutivos

```
genomeSizeM_BP[c(1,2,5)]
```

```
##   Human   Mouse   Yeast
## 3234.83 2716.97   12.10
```

Para selecciona (no eliminar, ni quitar) elementos excepto algunos

```
genomeSizeM_BP[-c(1,3,5)]
```

```
##       Mouse   Roundworm
## 2716.970000    0.014281
```

Para referirnos a los elementos por su nombre

```
genomeSizeM_BP[c("Yeast","Human")]
```

```
##   Yeast   Human
##   12.10 3234.83
```

Además de algunas operaciones aritméticas, se pueden calcular la media, máximo, mediana, mínimo, suma y longitud de los vectores

Ejercicio

1. Generar un vector de las edades de 10 de tus compañeros
2. Asignales nombre.
3. Encuentra el mínimo,máximo, media, mediana, la desviación estándar, la longitud del vector y selecciona sólo los elementos impares.
4. Elimina el máximo y el mínimo y con el vector resultante realiza un histograma.
5. Crea un vector de caracteres con diez nombres de especies y asocialo con su número de acceso de NCBI para su genoma en nucleótidos.

3.10 ¿Cómo lidiar con las NAs ?

Es (muy) freceunte que en bases de datos se tengan valores `NA`, es decir medidas que no pudieron realizarse, medidas perdidas, etc. Para ello se utiliza `NA`. `R` trata de manera especial a las NAs

```
x <- c(88,NA,12,168,13)
```

Existe una fución para determinar si un elemento es o no una `NA`. La función es `is.na()`

```
x <- c(88,NA,12,168,13)
```

```
is.na(x)
```

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

Si queremos calcular ciertas funciones numéricas R no sabrá qué hacer

```
x <- c(88,NA,12,168,13)
mean(x)
```

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

Sin emabrgo, podemos decirle a R que las omita, indicando como argumento de la función `mean()` `na.rm=TRUE` que significa *na remove*

```
x <- c(88,NA,12,168,13)
mean(x,na.rm=TRUE)
```

```
## [1] 70.25
```

¿Qué otras funciones tienen esta opción?

3.11 Filtrado de elementos de un vector

Podemos generar vectores de que sean subconjuntos de vectores más grandes que cumplan cierta(s) condición(es)

```
un_vector<-c(1,2,3,5,7,11,13,17,19)
otro_vector <- un_vector[un_vector*un_vector > 10]    # Leeme de adentro hacia afuera
otro_vector
```

```
## [1]  5  7 11 13 17 19
```

Veamos paso a paso qué es lo que hace este proceso

```
un_vector
```

```
## [1]  1  2  3  5  7 11 13 17 19
```

```
un_vector*un_vector > 10    # Mira, de adentro hacia afuera
```

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

```
indices<-un_vector*un_vector > 10
un_vector[indices]
```

```
## [1]  5  7 11 13 17 19
```

```
un_vector[c(FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE)]
```

```
## [1]  5  7 11 13 17 19
```

la representación interna de los valores booleanos `FALSE` y `TRUE`son 0 y 1 respectivamente

```
un_vector[c(rep(0,3),rep(1,1))]
```

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

3.11.1 Filtrado con `subset()`

Podemos usar la función `subset()`para hacer lo mismo que en el caso anterior **excepto que omite los NA**

```
un_vector<-c(1,2,3,5,7,11,13,17,19)
otro_vector <- subset(un_vector,un_vector*un_vector > 10)
otro_vector
```

```
## [1]  5  7 11 13 17 19
```

Qué pasa si tenemos `NA`s. Si usamos el método anterior obtendríamos

```
un_vector<-c(1,2,3,5,7,11,NA,13,17,NA,19)
otro_vector <- un_vector[un_vector*un_vector > 10]   # Leeme de adentro hacia afuera
otro_vector  # Aquí salen las NAs
```

```
## [1]  5  7 11 NA 13 17 NA 19
```

En cambio con `subset()`

```
un_vector<-c(1,2,3,5,7,11,NA, 13,17,NA, 19)
otro_vector <- subset(un_vector,un_vector*un_vector > 10)
otro_vector  # Aquí ya no aparecen las NAs
```

```
## [1]  5  7 11 13 17 19
```

3.11.2 La función de selección `which()`

La función `which()` nos regresa los índices es decir nos dicen **quiénes** cumplen cierta condición

```
z <- c(5,2,-3,8)
which(z*z > 8)
```

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

Acá nos dicen quiénes

```
z[which(z*z > 8)]
```

```
## [1]  5 -3  8
```

3.12 ¿Cómo podemos ver si dos vectores son iguales?

Dos vectores son iguales si elemento a elemento son idénticos. Por lo tanto deben de ser del mismo tamaño. Para probar si dos elementos son iguales se utiliza el operador de comparación `==` son dos signos iguales juntos, sin espcaio. No confundir con el operador `=` que se puede usar como operador de asiganción (aunque no es recomendable su uso. De hecho está prohibido en este curso)

```
x<-c(1,4,9,16,25)
y<-1:5
y<-y*y
```

```
x==y
```

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

¿Qué pasaría si me confundo y escribo el operador de igualdad en lugar del de comparación?

```
y <-5:9
y
```

```
## [1] 5 6 7 8 9
```

```
x=y
```

```
x
```

```
## [1] 5 6 7 8 9
```

```
y
```

```
## [1] 5 6 7 8 9
```

Para vectores grandes puedo usar la función `all()` que ya vimos arriba

```
x <- seq(1,10000,1)
y <- seq(1,10000,1)
all(x==y)
```

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

¿Cómo podríamos corrobar que son iguales usando `any`?

También podríamos utilizar que `TRUE` es `1` y que `FALSE` es `0`

¿Por qué este código nos dice que sí son iguales?

```
sum(x==y)
```

```
## [1] 10000
```

3.12.1 Factor

Los factores son un tipo de vector que puede tomar un número "limitado" de valores, que normalmente se utilizan como variables categóricas. Por ejemplo: macho/hembra. Es útil tenener este tipo de objeto porque varios modelos estadísticos que se pueden correr en `R` los utilizan. A los valores que pueden tomar los elementos del factor se les conoce como *levels*.

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

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

```
as.factor(x)
```

```
##  [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

```
x<-as.factor(x)
x
```

```
##  [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

Los factores son una manera computacionalmente eficiente de almacenar caracteres, pues cada valor único (level) se guarda solo una vez y a los datos se les asigna un valor entero.

```
meses = c("March","April","January","November","January",
       "September","October","September","November","August",
        "January","November","November","February","May","August",
        "July","December","August","August","September","November",
         "February","April")
meses
```

```
##  [1] "March"    "April"    "January"  "November" "January"  "September"
##  [7] "October"  "September" "November" "August"   "January"  "November"
## [13] "November" "February"  "May"      "August"   "July"     "December"
## [19] "August"   "August"   "September" "November" "February" "April"
```

```
meses<-as.factor(meses)
meses
```

```
##  [1] March    April    January  November January  September October
##  [8] September November August   January  November November February
## [15] May      August   July     December August   August   September
## [22] November  February  April
## 11 Levels: April August December February January July March May ... September
```

El que existan los levels permite realizar ciertas operaciones y manipular el contenido del factor.

```
table(meses)
```

```
## meses
##    April   August  December  February   January      July    March      May
##        2        4         1         2         3         1         1         1
##  November   October September
##         5         1         3
```

```
levels(meses)
```

```
##  [1] "April"    "August"   "December"  "February"  "January"   "July"
##  [7] "March"     "May"       "November"  "October"   "September"
```

```
levels(meses)[1]
```

```
## [1] "April"
```

```
levels(meses)[1]<-"Abril"
levels(meses)
```

```
##  [1] "Abril"    "August"   "December"  "February"  "January"   "July"
##  [7] "March"     "May"       "November"  "October"   "September"
```

```
meses
```

```
##  [1] March    Abril    January  November January  September October
##  [8] September November August   January  November November February
## [15] May      August   July     December August   August   September
## [22] November  February  Abril
## 11 Levels: Abril August December February January July March May ... September
```

Ejercicio

1. Lee la ayuda de `as.factor` para determinar cómo crear un factor "ordenado"
2. Crea un vector con los meses en los que todas las alumnas del grupo cumplen años.
3. Aprovecha los levels para generar un objeto que guarde el número de estudiantes que cumplena ño cada mes.

Ejercicio

1. Genera un vector con el nombre de 10 virus
2. Asocia esos nombres con su número de acceso en NCBI
3. Genera otro vector que contega los tamaños en pb y los nombres
4. Determina cuáles son mayores de 300 bp
5. Asocia un subconjunto de vectores que sean mayores (menores a 300 bp) y mayores (mayores a 300 bp)
6. Haz un histograma con los tamaños de todos
7. Dibuja un boxplot con los tamaños de todos.Pon en el eje los nombres de todos.

Chapter 4

Matrices

Una matriz es un arreglo rectangular de datos del mismo tipo. No. No se pueden mezclar.

4.1 Creación de matrices

Para crear una matriz podemos usar la función `matrix()`. Dicha función requiere de, al menos un vector e indicar al menos una dimensión.

```
y <- matrix(c(1,5,8,-4),nrow=2,ncol=2)
y
```

```
##      [,1] [,2]
## [1,]    1    8
## [2,]    5   -4
```

Se el indica el numero de renglones y el número de columnas como opción usando `nrow` y `ncol` respectivamente.

La matriz se llena por renglones hasta completarse

```
z<-matrix(c(TRUE, FALSE,rep(c(TRUE, FALSE),3)),nrow=4)
z
```

```
##       [,1]  [,2]
## [1,]  TRUE  TRUE
## [2,] FALSE FALSE
## [3,]  TRUE  TRUE
## [4,] FALSE FALSE
```

¿Por qué sólo es necesario indicar una dimensión?

Podemos decirle a `R` que cambie el orden con el que llena la matriz, es decir en lugar de que lo haga por columnas, lo haga por renglones

```
m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=TRUE)
m
```

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

4.2 Dimensiones de un matriz

La dimensión de una matriz es el número de renglones y de columnas respectivamente. Se puede obtener usando la función `dim()`

```
dim(y)
```

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

```
dim(z)
```

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

Así una matriz se distingue de un vector ya que tiene, además de renglones, columnas.

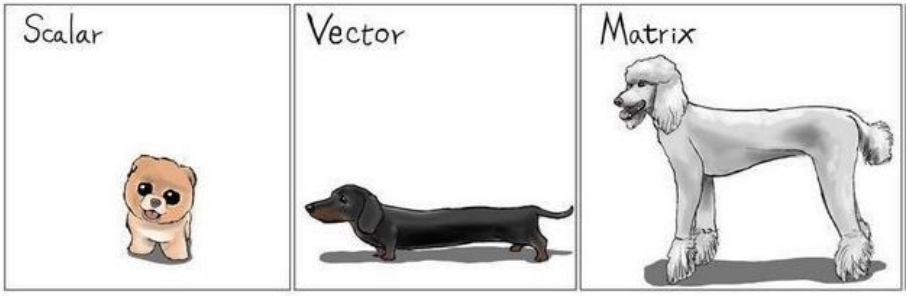


Figure 4.1: Escalar, vector y matriz

Una forma **mucho menos eficiente** de definir una matriz es declarando una matriz sin elementos y después llenándolos de forma explícita

```
 y <- matrix(nrow=2,ncol=2)
y[1,1] <- "Esta"
y[2,1] <- "es"
y[1,2] <- "una"
y[2,2] <- "matriz"
y
```

```
##      [,1]   [,2]
## [1,] "Esta" "una"
## [2,] "es"   "matriz"
```

4.3 Operaciones con matrices

4.3.1 Multiplicación de un escalar con una matriz

```
3*m
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]   12   15   18
```

4.3.2 Suma de dos matrices

```
m + m
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12
```

```
n<-matrix(c(2,3,4,5,6,7),ncol=3)
m+n
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]    7   10   13
```

Para sumar matrices deben tener las mismas dimensiones

```
dim(n)
```

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

```
dim(m)
```

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

```
(dim(n)-dim(m))==0
```

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

4.3.3 Multiplicación de matrices

Se utiliza el operador `%*%`. Sí. Son tres caracteres. E incluyen dos `%`. No hay espacios y es un sólo operador .

```
n<-matrix(c(2,3,4,5,6,7),ncol=2)
n
```

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

```
m %*% n
```

```
##      [,1] [,2]
## [1,]   20   38
## [2,]   47   92
```

¿Recuerdas cuál es el criterio para calcular el producto de matrices? ¿Recuerdas cómo se multiplican dos matrices?

4.4 Seleccionar elementos de matrices

Para seleccionar elementos de matrices se hace de forma análoga a vectores, es decir, se utiliza el operador `[]`. Sólo que esta vez hay que indicar tanto los renglones como la columna en ese orden

```
m[2,3]  # Este es el segundo renglón tercera columna de m
```

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

```
n[3,2]  # Este es el elemento que está en el renglón 3 y columna 2 de la matriz n
```

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

4.4.1 Seleccionar todo(a) un(a) renglón(columna)

Para seleccionar todos los elementos de un renglón dado se utiliza la siguiente sintáxis

```
m[2,]  # Todos los elementos que están en el segundo renglón
```

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

Para una columna

```
m[,3] # Toda la tercera columna
```

```
## [1] 3 6
```

4.4.2 Seleccionar elementos de una matriz

¿Qué hace lo siguiente?

```
m[1:2,1]
```

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

```
m[1:2,2:3]
```

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

```
m[-1,]
```

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

```
m[-1,-c(1,3)]
```

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

4.5 Nombres a renglones y columnas

Al igual que con vectores le podemos poner nombres tanto a renglones como a columnas para ello utilizamos `rownames()` y `colnames()`

```
m   # No tengo nombres :(
```

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

```
colnames(m)<-LETTERS[1:3]
rownames(m)<-letters[5:6]
```

```
m   # Ahora sí. Feos, pero nombres :) :)
```

```
##   A B C
## e 1 2 3
## f 4 5 6
```

```
m["e","C"]
```

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

```
m["e","C"]==m[1,3]
```

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

Chapter 5

Estructuras de selección

1. `if`
2. `if … else`
3. `ifelse`
4. `if ... else if ...else if ...else`

5.1 If (si condicional)

La instrucción `if` nos permite probar una condición y esa condición debe arrojar un valor booleano, es decir, un valor de verdad (`TRUE` o `FALSE`) Si la condición es verdadera se ejecuta lo que está dentro de los corchetes, de lo contrario, ejecuta lo que sigue después del corchete de cierre.

Definición: Lo que se encuentra dentra del corchete se llama cuerpo (*body*) de la declaración `if`

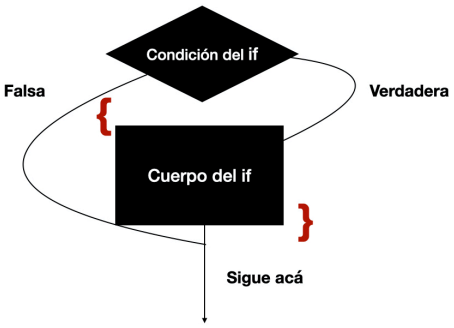


Figure 5.1: Diagrama de flujo del If

La sintáxis de una condición `if` consiste en lo siguiente:

```
if (condicion){

si la condicion es verdadera

Ejecuta TODO lo que está en los corchetes

}
```

5.1.1 Sintaxis

```
if(<condition>) {
        ## Hace algo
}
## Continúa con el resto del código
```

5.1.2 Errores comunes en el `if`

1. No incializar la variable de la condición.
2. La condición no arroja un valor de verdad.
3. No poner todo lo que quieres que haga **dentro** de los corchetes.
4. Este no es un error, es más bien una advertencia, si la condición arroja un sólo valor de verdad sólo toma en cuenta el primero de ellos.

###Ejemplos de uso del `if`

```
mayor_de_edad<-18

edad<-20

if(edad >=mayor_de_edad){
  print("Eres mayor de edad")
}
```

```
## [1] "Eres mayor de edad"
```

```
x<-5+4
print(x)
```

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

```
minimo<-20000
dinero<-15000

if(dinero>=minimo){
  print("¿Cómo está Cancún?")
  print("La vida es buena")
```

```
  sobrante<-dinero-minimo
  print(paste("Me queda $", sobrante))
}
print("Acá sigue")
```

```
## [1] "Acá sigue"
```

5.1.2.1 Ejercicios

1. Elabora un programa que compare tu estatura con tu ídolx y determine si eres más altx.
2. Toma dos archivos fasta de virus distintos. Leelos con Biostrings y compara sus tamaños (en bp) y determina si el primero es más grande que el segundo.
3. A partir del archivo de anotación del genoma de un organismo determina toma dos proteínas al azar y compara sus tamaños. Toma todos los genes de la cadena positiva y todos los de la negativa compara sus tamaños promedio y determina cuál de estos es más grande.

5.2 Combinación de operadores booleanos

Los operadores lógicos o booleanos se pueden combinar para formar enunciados complejos por ejemplo:

1. Tengo vacaciones (del trabajo y/o la escuela)
2. Tengo dinero

Si las dos condiciones son ciertas entonces puedo hacer algo

También podría ser que basta con que una de ellas sea cierta para que haga algo.

5.2.1 And (&)

El operador booleano `&` representa el y lógico. Estos operadores binarios nos sirven para unir al menos dos enunciados que tienen valor de verdadero o falso (Tengo dinero (V/F), Tengo vacaciones (V/F))

Con estas dos operaciones puedo unirlas utilizando el operador `y` lógico (AND (`&`) representado en R con el símbolo del ampersand (`&`)

Tengo dinero AND Tengo vaciones

Para saber el valor booleano (V/F) del enunciado anterior debemos concocer los valores de verdad de los enunciados por separado

Por ejemplo, podemos representar al primer enunciado por `p` y al segundo enunciado por `q`

```
p: Tengo dinero
```

```
q: Tengo vacaciones
```

Para saber cuál es el valor de verdad del enunciado compuesto debemos ver cuáles son todas las combinaciones de valores de verdad de los enunciados que la componen: p verdadero y q verdadero, p falso y q verdadero, p falso y q verdadero, p falso y q falso. Eso se resumen en las tablas de verdad de los operadores

Table 5.1: Tabla de verdad del AND

p	q	p & q
V	V	V
V	F	F
F	V	F
F	F	F

Es decir, el `&` solo es **verdadero** cuando ambas condciones son **verdaderas**.

Esto representa lo que se observa en la realidad: es decir, solo hago algo si tengo y tengo vacaciones. Si una de ellas no se cumple (es decir, es falsa) entonces no se lleva a cabo la acción.

5.2.2 OR (|)

El operador booleano `|` representa el `o` lógico. Estos operadores binarios nos sirven para unir al menos dos enunciados que tienen valor de verdadero o falso (Tengo dinero (V/F), Tengo vacaciones (V/F))

Con estas dos operaciones puedo unirlas utilizando el operador `y` lógico (OR (`|`) representado en R con el símbolo de *la barrita* (`|`)

Tengo dinero OR Tengo vaciones

Para saber el valor booleano (V/F) del enunciado anterior debemos concocer los valores de verdad de los enunciados por separado

Por ejmplo podemos representar al primer enunciado por `p` y al segundo enunciado por `q`

```
p: Tengo dinero
```

`q: Tengo vacaciones`

Para saber cuál es el valor de verdad del enunciado compuesto debemos ver cuáles son todas las combinaciones de valores de verdad de los enunciados que la componen: p verdadero y q verdadero, p falso y q verdadero, p falso y q verdadero, p falso y q falso. Eso se resumen en las tablas de verdad de los operadores

Table 5.2: Tabla de verdad del operador OR

p	q	p | q
V	V	V
V	F	V
F	V	V
F	F	F

Es decir haría algo, por ejemplo, irme a la playa cuando **al menos** una condición se cumpla. Por ejemplo que tenga dinero aunque no tenga vacaciones, que tenga vacaciones aunque no tenga dinero y, obviamente, también cuando las dos se cumplen.

Es decir, el `|` solo es **falso** cuando ambas condiciones son **falsas**.

5.2.3 Ejemplos de combinaciones

Por eejmplo es útil para intervalos

$$18 \leq edad \leq 29$$

Esta condición la podemos expresar mediante la combinación de dos: la edad debe ser mayor igual a 18 y (**AND**, `&`) la edad debe ser menor o igual que 29

```
if (edad >= 18 & edad <=29){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

Pregunta: ¿qué pasaría si se pone un **OR** como unión entre las dos condiciones

```
if (edad >= 18 | edad <=29){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

o así (¿es lo mismo?)

```
if (edad <= 29 | edad >= 18){
  print("Te toca vacunarte")
}
```

```
## [1] "Te toca vacunarte"
```

5.3 Ejercicio

1. ¿Cómo harías una condición que considere que te gusta el mole y el pozole?
2. ¿Cómo harías una condición que considere que te gusta el mole o el pozole?

5.4 If ... else (si ... de otro modo)

Si además quieres que se ejecute algo cuando la condición es **falsa** entonces debes usar la declaración `if … else`

```
if (condición) { # Si la condición es cierta
hace esto
y esto
y esto
} else { # De otro modo, es decir si es falsa hace lo que #está en el corchete
entonces hace esto otro
y esto otro
y esto
}
```

```
minimo<-20000
vacaciones<-"SI"
dinero<-21000
if(dinero>=minimo & vacaciones=="SI"){
  print("Me voy a la playa, looser")
}else{
  print("Me quedo en mi casa")
}
```

```
## [1] "Me voy a la playa, looser"
```

5.5 ifelse

Si la condición es muy simple ,tanto para cuando es verdadero como cuando es falso se puede implementar la función `ifelse` en una línea. Es equivalente a la

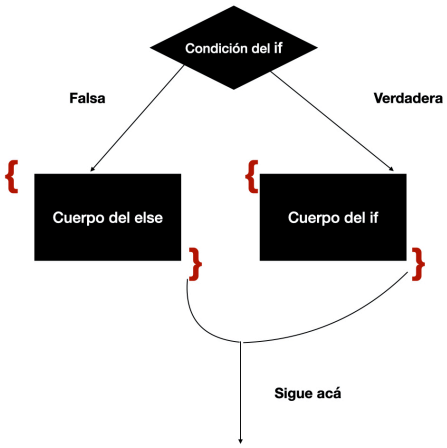


Figure 5.2: Diagrama_if_else

condición compuesta pero ahorramos código.

```
edad<-21
ifelse(edad>=18, "Ya eres grande","Todavia no puedes beber (legalmente)")
```

```
## [1] "Ya eres grande"
```

```
edad<-12
ifelse(edad>=18, "Ya eres grande","Todavia no puedes beber (legalmente)")
```

```
## [1] "Todavia no puedes beber (legalmente)"
```

5.6 If ... else if ... else (si, si no si , si no si, si no)

Si tienes más opciones, es decir no alternativas, puedes usar la sentencia `if … else if …else if …else`

Importante Esta estructura se ejecuta solo en la primera que sea verdadera o si no hay una verdadera ejecuta lo que esta en el `else`

```
if ( condicion 1) {
Hace cosas
} else if ( condcion 2) {
Hace otras cosas
} else if ( condicion 3) {
Hace estas otras cosas
} else {
```

```
No le queda de otra y hace esto
}
```

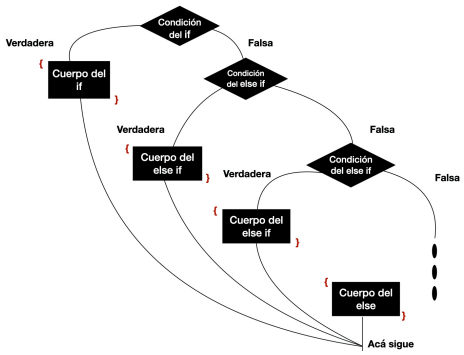


Figure 5.3: Diagrama de flujo del if... else if... else if...else

```
numero<-3
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es positivo"
```

```
numero<- -27
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es negativo"
```

```
numero<- 0
if(numero > 0){
  print("Tu número es positivo")
}else if (numero <0){
  print("Tu número es negativo")
}else{
  print("Tu número es cero")
}
```

```
## [1] "Tu número es cero"
```

Pregunta: ¿por qué no es necesario poner un if en el último else?

5.7 Ejercicios

1. Elabora un programa que con tu fecha de cumpleaños te diga en qué estación del año naciste.
2. Elabora un porgrama que a partir de las calficaciones de tus examenes parciales y 8 quincenales arroje si exentaras o no este curso usando los criterios definidos en el programa del curso. Asume que en las tareas y demás actividades tienes 10.