R basics

Alfredo Cortell-Nicolau

Introducción

Aunque es versátil, R es un lenguaje diseñado por estadísticos con el principal objetivo de aplicar técnicas estadísticas. Como resultado, muchos de sus conceptos y estructura se relacionan con los modelos de trabajo desarrollados dentro de ese campo. Existen grandes cantidades de material de calidad online y abierto para aprender R, pero no todos tienen en cuenta los primeros pasos (cuando suele ser más difícil). Como cualquier idioma (y un lenguaje de programación no deja de ser un idioma nuevo), conocer bien las reglas básicas nos ayudará enormemente en un futuro para poder hablar con fluidez y, sobre todo (en el caso de la computación) con eficiencia. En este primer tutorial, se expondrán brevemente algunos de esos conceptos.

Tipos de datos

Escalares

El tipo más básico de datos en R es lo que denominamos 'escalar' que, en práctica, es un objeto con un único valor. Para crear un escalar, puede teclearse el siguiente código.

```
escalar <- 1
```

El siguiente código hace la misma función.

```
# Es indiferente utilizar comillas o flecha para asignar objetos
# Al teclear una almohadilla delante del código anulamos su ejecución. Se utilizan
# para informar a otros (y a nosotros mismos!) de lo que se intenta ejecutar
escalar = 1
```

Existen cuatro tipos de datos en R: numéricos, de factor o categóricos, de carácter y booleanos.

```
# Datos numéricos
minumero <- 1

# Datos categóricos o de factor
mifactor <- factor("blanco")

# Datos de carácter
micaracter <- "blanco"

# Booleanos
mibooleano <- TRUE ## La otra opción es FALSE
```

Vectores

El vector, o vector atómico, es el siguiente formato de datos en complejidad en R. Es simplemente la concatenación de escalares en un único objeto. Para crear un vector, podemos ejecutar el siguiente código:

```
mivector <- c(5,98,5,4,23)

# Puedes ver el resultado del objeto que has creado
print(mivector)
```

```
## [1]  5 98  5  4 23
```

El número de datos que puedes incluir en un vector es virtualmente ilimitado, pero todos los datos tienen que ser del mismo tipo (numéricos, factoriales, de carácter o booleanos). Qué pasa si ejecutas el siguiente código? Observa el objeto que produce:

```
mivector <- c(4,7,65,76,"blanco",85)
```

Matrices

Aumentando en complejidad, la matriz es el siguiente objeto. En esencia, una matriz es una concatenación de vectores por columnas (o filas). Supón que tienes los siguientes vectores:

```
vec1 <- c(5,4,91,75)
vec2 <- c(8,5,4,12)
```

Puedes convertirlos en una matriz con el siguiente comando

```
mimatriz <- matrix(c(vec1,vec2))
```

Obseva el resultado, viendo el objeto `mimatriz` ¿Es lo que esperabas? Para corregirlo, simplemente teclea lo siguiente:

```
mimatriz <- matrix(c(vec1,vec2), ncol = 2)

# Equivalentemente, también podrías teclear cbind(vec1,vec2) o obtendrías el  mismo
# resultado
```

Acabas de introducir el primer argumento en tu primera función. Más adelante veremos exactamente qué es una función y qué es un argumento.

Data frames

Los data frames son, en esencia, lo mismo que las matrices, con una salvedad. En un data frame, al contrario que en la matriz, puedes introducir diferentes tipos de datos. Mientras una matriz solo admite un tipo de datos y transformará todos los datos a un tipo único que los pueda representar, en un data frame esta transformación no ocurre. Prueba lo siguiente:

```r
# Creamos objetos para introducir en matriz y en data frame
vec1 <- c(8,48,90,12)
vec2 <- c("rojo", "azul", "verde", "negro")

# Creamos la matriz
mimatriz <- matrix(c(vec1,vec2), ncol = 2)
midataframe <- data.frame(vec1,vec2)
```

Fíjate que, aunque puedas introducir distintos tipos de datos, en ambos casos los dos vectores tienen que tener la misma longitud (el mismo número de valores). Si no fuera así, en el caso del data frame R te dará un error y no te dejará construirlo, mientras que, en el caso de la matriz, te dará un warning informándote de que la longitud de tus vectores no coinciden, pero terminará el objeto que le has pedido autocompletándolo (lo cual no sea, probablemente, lo que tú tenías en mente). Pruébalo! Es bueno empezar a acostumbrarte a ver errores en R y qué signfican. Si vas a hacer código en R, te vas a hartar de ver errores!

Lista

Por último, y con permiso de los `array`, que veremos después, el objeto más complejo dentro de los objetos de base de R es la lista. Una lista simplemente es, y siguiendo el razonamiento anterior, una concatenación de cualquiera de los objetos anteriores. Esencialmente, una lista es un objeto que 'contiene' objetos. Puedes pensar en una lista como un armario con un objeto diferente en cada cajón. Cada uno de esos objetos puede ser un escalar, un vector, un data frame, una matriz, un array o incluso otra lista. Un ejemplo aquí.

```r
# Considera los siguientes objetos
unvector <- c(54,74,99,1)
unamatriz <- cbind(vec1,unvector)
undataframe <- data.frame(vec1,vec2)

# Puedo hacer una lista que contenga todos los anteriores
milista <- list(unvector, unamatriz, undataframe)
```

Estructura de objetos

Entendiendo nuestros objetos

Hasta aquí hemos visto los elementos básicos de cómo se crean los objetos en R, pero vamos a necesitar mucho más. Una vez tenemos los objetos creados ¿Qué hacemos con ellos? ¿Cómo operamos?

Como en cualquier situación (no solo estadística ni computacional, sino general!) para solucionar un problema lo primero que tenemos que hacer es entenderlo bien, y para entenderlo bien, hay que observarlo, explorarlo y comprender cómo funciona internamente y externamente. En nuestro caso, cualquier problema que querramos analizar va a depender de los datos que tengamos y, por lo tanto, absolutamente siempre, en estadística y computación, lo primero que vamos a tener que hacer es entender nuestros datos. R tiene muchas herramientas para ello.

En primer lugar, algo que siempre vamos a necesitar saber, es cuánto tenemos de cada cosa. Para esto, la función `length` te puede ayudar. Esta simplemente devuelve el número de valores del objeto sobre el que actúa

```r
length(vec1)
```

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

Si el objeto es un data frame o una matriz, puede que `length` no sea tu mejor opción, y prefieras saber el número de filas, de columnas, o de dimensiones. Para ello, puedes utilizar

```
nrow(mimatriz) ## número de filas
```

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

```
ncol(mimatriz) ## número de columnas
```

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

```
dim(mimatriz) ## número de dimensiones (o de filas y columnas)
```

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

Hemos hablado anteriormente de tipos de datos pero, cómo se conoce esto en R? ¿Cómo puedo saber el tipo de datos de cualquier objeto? Para esto, la función `class` es tu amiga. Hablaremos más adelante de funciones, pero por ahora, simplemente entiende las funciones como un protocolo interno de comunicación (o un algoritmo) que permite realizar acciones específicas sobre determinados objetos. En este caso, considera que nuestro objeto es el vector `vec1`, que hemos creado anteriormente, y lo que queremos es entender el tipo de datos que contiene. Para ello, simplemente introduce el código:

```
class(vec1)
```

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

```
class(vec2)
```

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

Te informa de que tu vector es numérico en el primer caso, y de que está compuesto por caracteres en el segundo. Si aplicas esta función sobre objetos más complejos, te indicará el tipo de objeto con el que estás tratando.

```
class(mimatriz)
```

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

```
class(midataframe)
```

```
## [1] "data.frame"
```

Esto puede ser muy útil más adelante, ya que determinados paquetes y funciones de R te requerirán que los objetos que introduzcas sean de un formato específico. Normalmente, no es complicado cambiar el formato de los objetos, y basta introducir el prefijo `as.` a un tipo de objeto, aunque cuando lo hagas por tu cuenta te encontrarás con sorpresas en casos determinados que tendrás que solucionar. Prueba lo siguiente

```r
midataframe2 <- as.data.frame(mimatriz)
class(midataframe2)
```

```
## [1] "data.frame"
```

La función `class` también puede aplicarse a las listas (`class(milista)`) pero, en este caso, una función aún más interesante es la función `str`. Esta función nos permite indagar sobre la estructura de un objeto, y esto va a ser muy útil y necesario cuando queramos actuar parcialmente sobre él, algo que ocurre mucho más de lo que pensáis. Vamos a explorar nuestra lista

```r
str(milista)
```

```
## List of 3
##  $ : num [1:4] 54 74 99 1
##  $ : num [1:4, 1:2] 8 48 90 12 54 74 99 1
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "vec1" "unvector"
##  $ :'data.frame':    4 obs. of  2 variables:
##   ..$ vec1: num [1:4] 8 48 90 12
##   ..$ vec2: chr [1:4] "rojo" "azul" "verde" "negro"
```

Por último, es posible que quiera saber cómo se llaman mis filas y/o mis columnas para utilizarlo, ya sea en imágenes o en operaciones ulteriores. Para ello, simplemente puedo utilizar la función `colnames` para las columnas y `rownames` para las filas. Por ejemplo, considerando el data frame anterior

```r
colnames(midataframe)
```

```
## [1] "vec1" "vec2"
```

```r
rownames(midataframe)
```

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

Asimismo, puede que quiera asignarles otros nombres que tengan más significado para mi. En ese caso, puedo hacer lo siguiente

```r
colnames(midataframe) <- c("Vector.1", "Vector.2") ## Misma extensión que columnas
rownames(midataframe) <- c("Row.1", "Row.2",
                           "Row.3", "Row.4") ## Misma extensión que filas

# Comprueba que se ha creado lo que tú quieres
```

Consejo! Intenta no dejar espacios cuando crees nombres de filas y columnas. Te ahorrarás problemas y harás tu código posterior más legible.

Indexing

Ahora que se ha comentado la posibilidad de actuar sobre partes del objeto (y no sobre todo el objeto) es momento de abordar cómo se seleccionan diferentes elementos dentro de un objeto. Esto puede conocerse como `subsetting` y es, sencillamente, seleccionar partes específicas de un objeto para trabajar con ellas. Existe una función en Rbase llamada `subset`, específicamente diseñada para esto. Podéis explorarla si queréis, pero no es necesaria, y se puede obtener lo mismo y de forma más eficiente indexando vuestras selecciones. Indexar es simplemente como suena, seleccionar valores específicos dentro de un objeto más amplio de acuerdo con el índice (o posición) que ocupan. Empecemos con un ejemplo sencillo.

```
# Considera nuestro vec1
vec1
```

```
## [1]  8 48 90 12
```

```
# Seleccionemos su tercer valor
vec1[3]
```

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

```
# Seleccionemos todo el objeto, menos el tercer valor
vec1[-3]
```

```
## [1]  8 48 12
```

```
# Puedo seleccionar más de un objeto
vec1[c(1,2)]
```

```
## [1]  8 48
```

```
# Los tres puntos significan todos los elementos entre el
# valor inicial y el valor final
vec1[c(1:3)]
```

```
## [1]  8 48 90
```

Sencillo ¿Verdad? Vamos a complicarlo un poquito más. En el caso de data frames y matrices no podemos introducir solo un número porque tenemos que decirle a R, cuando indexamos, el objeto de qué fila y de qué columna, como sigue:

```
# Considera nuestro data frame anterior
midataframe
```

```
##       Vector.1 Vector.2
## Row.1        8     rojo
## Row.2       48     azul
## Row.3       90    verde
## Row.4       12    negro
```

```
# Para seleccionar filas, elijo el valor a la izquierda de la coma en el corchete.
# Por ejemplo, selecciono la fila 3
midataframe[3,]
```

```
##       Vector.1 Vector.2
## Row.3       90    verde
```

```
# Para seleccionar columnas, elijo el valor a la derecha de la coma en el corchete.
# Por ejemplo, selecciono la columna 2
midataframe[,2]
```

```
## [1] "rojo"  "azul"  "verde" "negro"
```

```
# Puedo combinar ambas. Así, si quiero seleccionar el elemento en la fila 3 de la
# columna 2, simplemente
midataframe[3,2]
```

```
## [1] "verde"
```

Adicionalmente (pero esto solo funciona con data frames y listas), puedo seleccionar una columna por su nombre con el signo del $. Prueba lo siguiente

```
midataframe$Vector.1
```

```
## [1]  8 48 90 12
```

```
# Es lo mismo que
midataframe[,1]
```

```
## [1]  8 48 90 12
```

Por último, puedes seleccionar cada elemento de una lista con doble corchete y, una vez lo tengas seleccionado, aplicar los criterios de selección vistos anteriormente según el tipo de objeto que sea. Algo muy importante de R, y que aprenderás con el tiempo, es que todos los pasos anteriores pueden concatenarse, lo cual suele hacer tu código mucho más efectivo. Prueba lo siguiente:

```
milista[[1]] ## Te devuelve el vector que habías creado como el primer objeto de tu lista
```

```
## [1] 54 74 99  1
```

Como, esencialmente, el objeto `milista[[1]]` es el mismo que `unvector`, cualquier operación de indexado que hubiera podido aplicar sobre `unvector` la puedo aplicar sobre `milista[[1]]`. Así,

```
unvector[1]
```

```
## [1] 54
```

```
## Es lo mismo que

milista[[1]][1]
```

```
## [1] 54
```

Todo esto puede complicarse mucho más, pero este es el funcionamiento básico y vamos a dejarlo aquí por ahora!

Operaciones básicas

Obviamente, todo lo anterior tiene un sentido. La preparación de nuestros es uno de los elementos más descuidados y a la vez más importantes en cualquier aspecto de la investigación, también en los casos de la arqueología computacional o de datos y la modelización matemática. Si no hay una buena higiene en el tratamiento de los datos, no importa lo sofisticado que sea vuestro modelo porque nada es mejor que los datos sobre los que se basa. Además, en muchas ocasiones, algunas funciones requerirán que entreguéis los datos de una determinada específicas, otras que eliminéis datos faltantes (o sobrantes) y, para otras, solo las querréis aplicar sobre subconjuntos específicos. Para adquirir todas estas habilidades, hay que seguir desarrollando los puntos que se acaban de mencionar más arriba, pero por ahora vamos a ver brevemente cómo funciona el núcleo de R una vez tenemos nuestros datos preparados.

Funciones

Las funciones son, esencialmente, el elemento en el que se basa el análisis de datos en R. Cada función es un protocolo que puede estar escrito en R o en C++ pero que, en todo caso, puede ejecutarse en R y tiene asignada una labor específica. Cada función consiste de dos partes, la función en sí `funcion` y un paréntesis `()` dentro del cual se introducen diferentes argumentos. Es decir, una primera parte en que se dice *qué* se va a hacer y una segunda parte en que se dice *cómo* se va a hacer. Pero vamos a verlo con una ejemplo. Supongamos que quiero calcular la media para `vec1`. R dispone de una función específicamente para eso. Simplemente teclea:

```
mean(vec1)
```

```
## [1] 39.5
```

¿Qué ha pasado aquí? Cuando las funciones son complejas, es fácil que no sepas exactamente qué es lo que pasa. No te preocupes. No estás sol@. Si quieres saber cómo funciona exactamente una función, teclea lo siguiente:

```
?mean
```

Al haber tecleado esto, en la ventana inferior derecha de vuestro Rstudio, ha aparecido (en el tab Help) una información que no teníais antes. Veis como aparece una especie de texto intrucciones donde lo primero que aparaece es el nombre de la función y el Rpackage al que pertenece `mean{base}` (veremos qué son los packages después). Debajo aparece el nombre real de la función, una descripción de lo que hace y cómo se utiliza. Usage describe las diferentes formas en que podemos utilizar la función, pero aún nos interesa más el apartado Arguments, en el que podemos ver todos los elementos que se pueden utilizar en la función. Por ejemplo, el argumento `x` es simplemente los datos, o el objeto sobre el que operará la función (este argumento es común a muchas funciones, pero no a todas) ,y los argumentos `trim` y `na.rm` tienen funciones específicas. Leyendo la ayuda, algunospuede decirme qué es lo que hacen estos argumentos?

Vemos otra función donde se incluiremos argumentos adicionales. Por ejemplo

```r
sort(vec1)
```

```
## [1]  8 12 48 90
```

```r
sort(vec1, decreasing = TRUE) ## Cómo funciona un argumento booleano?
```

```
## [1] 90 48 12  8
```

Aparte, también se pueden realizar, simplemente escribiéndolas, toda suerte de operaciones matemáticas, que pueden actuar bien sobre escalares al momento

```r
2+2
```

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

```r
3*4
```

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

```r
3/3
```

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

```r
3^2
```

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

```r
sqrt(4)
```

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

O bien sobre vectores creados previamente

```r
unvector+vec1
```

```
## [1]  62 122 189  13
```

```r
# No confundir con
sum(unvector,vec1)
```

```
## [1] 386
```

Iteraciones

Pero la fuerza de la computación reside en realidad en la capacidad que tienen los ordenadores para realizar tareas repetitivas que a nosotros, si las tuviéramos que hacer a mano, nos llevarían mucho tiempo. En ese sentido, las iteraciones, bucles o loops son algunos de nuestros mejores aliados.

Los bucles, o loops, no son la manera más eficiente de iterar en R. Es mejor utilizar indexado o las funciones de la familia `apply` siempre que sea posible, pero los bucles son lo más expeditivo y rápido de entender así que, por ahora, nos quedaremos aquí y os animo a que sigáis explorando qué es `apply`, cómo se utiliza y por qué es más eficiente que un bucle. Centrándonos en los bucles, existen tres tipos básicos. Los bucles `for`, `while` e `if`, que puede combinarse con `else`.

Los bucles `for`, simplemente repetirán una orden tantas veces como especifiques:

```
for (i in 1:15){
  print(as.character(i))
}
```

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

Los bucles `while` continuarán operando siempre que se cumpla una condición preespecificada:

```
i <- 1
while (i <= 15){
  print(as.character(i))
  i <- i +1
}
```

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

```
## [1] "13"
## [1] "14"
## [1] "15"
```

Los bucles `if` ejecutaran el comando dentro del bucle siempre que se cumpla la condición especificada

```
i <- 15
if (i <= 15){
  print("i es menor o igual que 15")
}
```

```
## [1] "i es menor o igual que 15"
```

Y podemos añadirles una alternativa en el caso de que esa condición no se cumpla con `else`

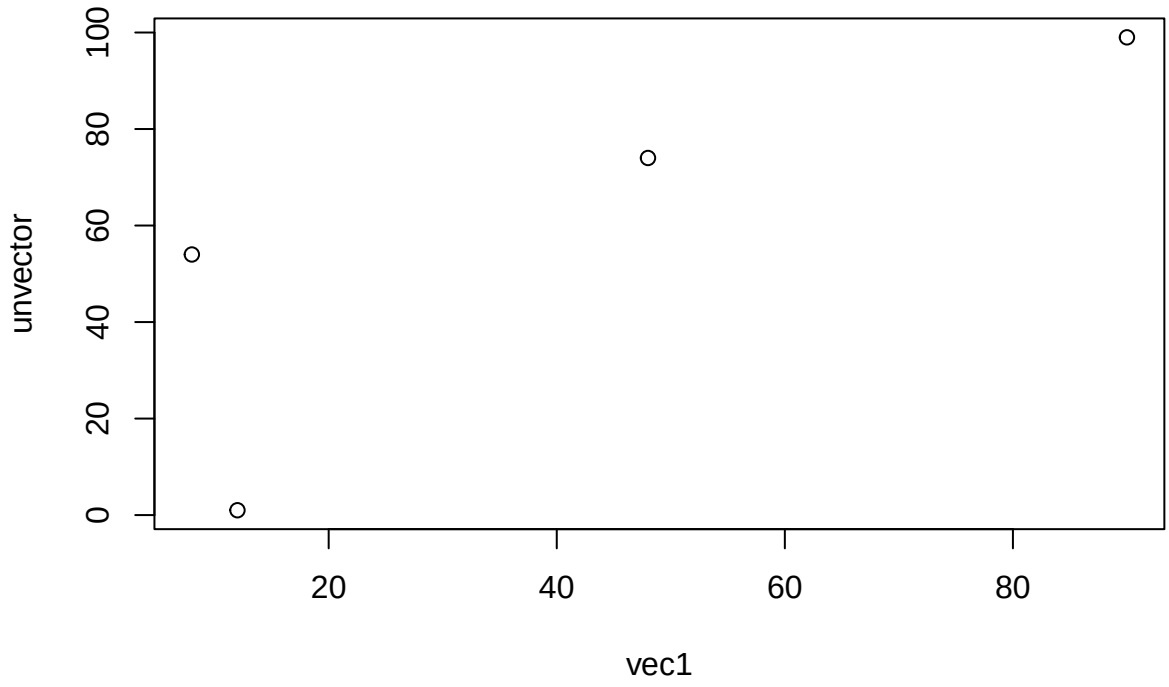
```
i <- 16
if (i <= 15){
  print("i es menor o igual que 15")
} else {
  print("i no es menor o igual que 15")
}
```

```
## [1] "i no es menor o igual que 15"
```

plots

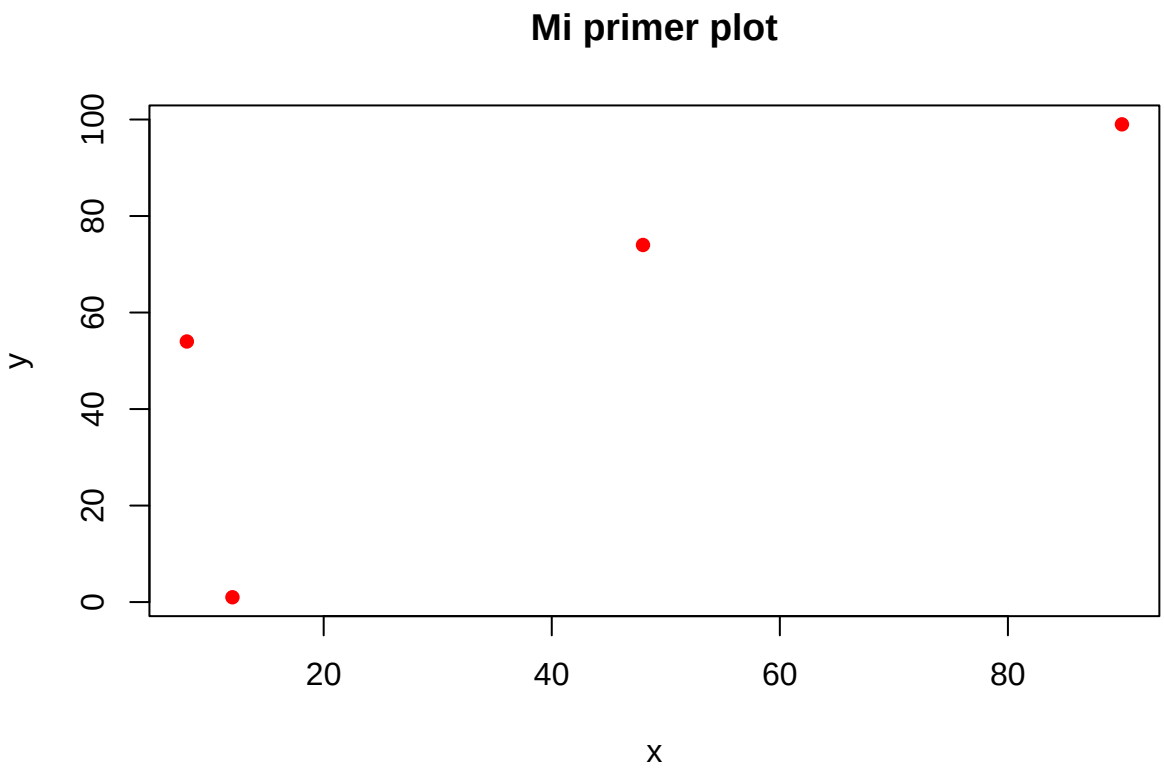
Todo esto nos permitirá analizar datos cuando comencemos a entrar en funciones cada vez más complejas. Sin embargo, uno de los elementos clave para cualquier análisis de datos es poder visualizarlos. Por supuesto, R cuenta con una serie de herramientas muy aptas para el análisis visual y la observación de los distintos análisis. La más básica de ellas es la función plot.

```
plot(vec1,unvector)
```



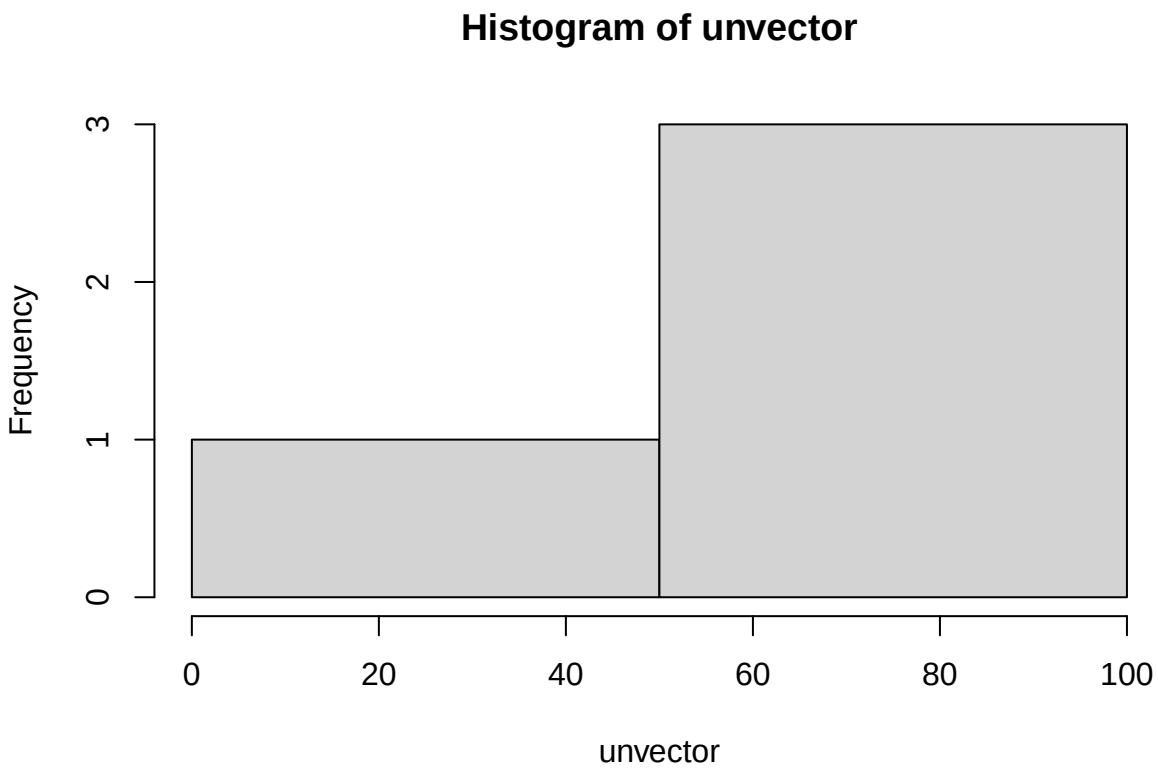
¿Qué ha pasado aquí exactamente? Aunque no lo he escrito arriba, a no ser que se especifique lo contrario, la función `plot` siempre toma el primer elemento como el valor para el eje x del gráfico y el segundo elemento como el valor para el eje y. Además, puedo mejorar mucho mis plots con argumentos simples como los que siguen.

```
plot(vec1, unvector, xlab = "x", ylab = "y", pch = 16, col = "red",
     main = "Mi primer plot")
```

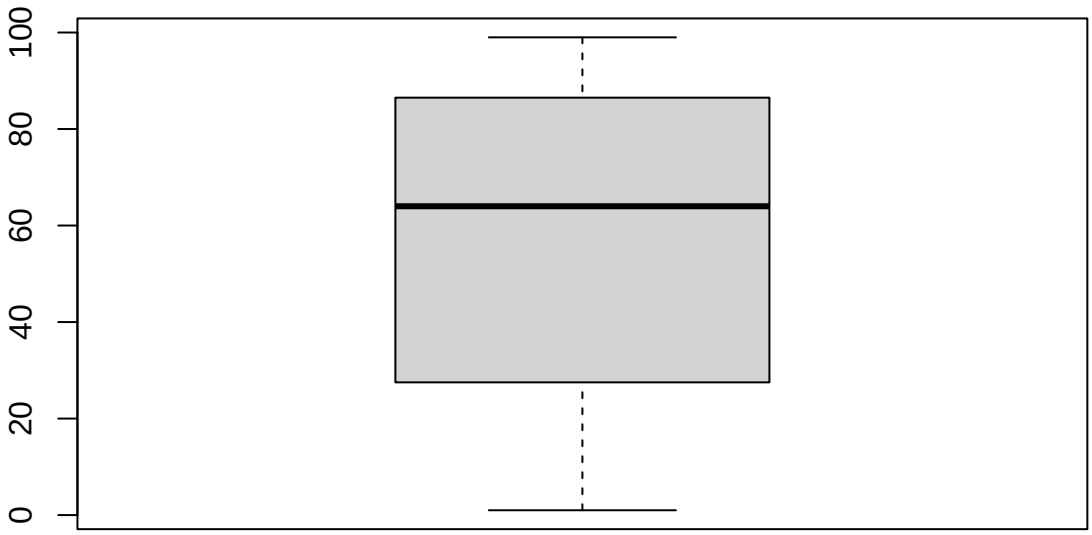


Muchos paquetes tienen sus propios esquemas de visualización, e incluso existen paquetes cuyo único objetivo es conseguir buenos plots, pero siempre es mejor que controléis la base antes de lanzaros a paquetes más complejos que, al introducir más interfaces (y cajas negras potenciales) entre vosotros, vuestro código y el resultado que deseáis, pueden crear problemas de compatibilidades, memoria e incluso de comprensión del propio código. Existen muchos tipo de plots diferentes.

```
hist(unvector) ## Histograma
```




```
boxplot(unvector) ## Gráfico de caja y bigotes
```



Pero esos os los dejo explorar por vuestra cuenta.

Paquetes de R. Instalación y uso.

Se ha hablado anteriormente en varias ocasiones ya de paquetes de R. ¿Qué son estos paquetes de R? Bien, R dispone de una serie de funciones (generalmente conocido como Rbase) que vienen con la instalación primaria del programa, pero R es un lenguaje abierto y cualquiera puede introducir nuevas funcionalidades (y esa es parte de su fuerza). Por lo tanto, es posible que existan elementos específicos que no estén en la base del lenguaje (la instalación primaria) pero que alguien los haya desarrollado y pueden instalarse a modo de `add-ons` o `plug-ins`. Estos nuevos desarrollos se cuentan por miles. De hecho, en este momento existen más de 20.000 paquetes en CRAN (el repositorio oficial para los paquetes de R) más otros tantos miles en git. Por supuesto, no tenéis que saberlos todos, pero hay algunos básicos que más tarde o más pronto acabaréis encontrando. Casi todas las técnicas avanzadas (como la morfometría geométrica) requieren de sus propios paquetes. En nuestro caso, vamos a utilizar esencialmente, dos paquetes, `geomorphy momocs`. Para instalarlos, simplemente teclea lo siguiente:

```
install.packages(c("geomorph","Momocs"))
```

Podéis activar los paquetes para vuestra sesión con el siguiente código:

```
library(geomorph)
```

```
## Loading required package: RRPP
```

```
## Loading required package: rgl
```

```
## This build of rgl does not include OpenGL functions.  Use
##  rglwidget() to display results, e.g. via options(rgl.printRglwidget = TRUE).
```

```
## Loading required package: Matrix
```

```
library(Momocs)
```

```
##
## Attaching package: 'Momocs'
```

```
## The following object is masked from 'package:geomorph':
##
##     mosquito
```

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

```
## Y con esto consultáis los paquetes que tenéis actualmente activados (y sus
## versiones)
sessionInfo()
```

```
## R version 4.2.2 Patched (2022-11-10 r83330)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux 12 (bookworm)
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.11.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.11.0
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] Momocs_1.4.1   geomorph_4.0.6 Matrix_1.6-5   rgl_1.2.8      RRPP_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.12       compiler_4.2.2    pillar_1.9.0      highr_0.10
##  [5] base64enc_0.1-3   tools_4.2.2       digest_0.6.34     jsonlite_1.8.8
##  [9] evaluate_0.23     lifecycle_1.0.4   tibble_3.2.1      gtable_0.3.4
## [13] nlme_3.1-162      lattice_0.20-45   pkgconfig_2.0.3   rlang_1.1.3
```

```
## [17] cli_3.6.2         rstudioapi_0.15.0 yaml_2.3.8        parallel_4.2.2
## [21] xfun_0.41         fastmap_1.1.1     dplyr_1.1.4       knitr_1.45
## [25] htmlwidgets_1.6.4 generics_0.1.3    vctrs_0.6.5       grid_4.2.2
## [29] tidyselect_1.2.0  glue_1.7.0        R6_2.5.1          jpeg_0.1-10
## [33] fansi_1.0.6       rmarkdown_2.25    ggplot2_3.4.4     magrittr_2.0.3
## [37] scales_1.3.0      htmltools_0.5.7   colorspace_2.1-0  ape_5.7-1
## [41] utf8_1.2.4        munsell_0.5.0
```