Introducción a R en 60 minutos

(Ay, ay, menudo jardín)

Mariano Rico (mariano.rico@upm.es)

Documento creado el 08/02/2022

Tabla de contenidos

1 Requisitos para empezar

Asumimos que ya conoces CRAN y sabes cómo instalar R, la consola de comandos (no confundir con RStudio, una herramienta que aprovecha un R ya instalado). También asumimos que la versión de R es superior a la 4.0 (en la fecha de escritura de este documento la última versión es la 4.1.2).

Lo más sencillo es usar el "escritorio virtual" de la UPM. Puedes usarlo desde tu navegador o con un *cliente RDP*.

2 Introducción

R es un leguaje "interpretado". Esto quiere decir que no hay un compilador que genere el código máquina del ejecutable. Por este motivo, no se dice "un programa R" sino "un script R".

Cuando se ejecuta un script R, se "compila una línea" del script y se ejecuta, y así para todo el script. Esto quiere decir que podemos tener un error sintáctico (un compilador detectaría el error) en el script y no lo detectamos hasta que se ejecuta la línea del error. Para detectar esos errores sintácticos en tiempo de edición (antes de ejecutarlo) hay entornos de desarrollo como RStudio.

R tiene varias ventajas frente a lenguajes compilados como C o Java: una comunidad enorme que genera librerías (*paquetes* en jerga R) de alta calidad, puede usar librerías escritas en otros lenguales (Fortran, C++, etc.), puede usar todos los procesadores de tu ordenador, usar clusters de ordenadores... y, sobretodo, permite mucha flexibilidad a la hora de escribir código: desde invocar funciones simples y usar bucles `for` tradicionales, a usar técnicas más sofisticadas como programación orientada a objetos, o programación funcional. En definitiva, la flexibilidad permite muchos estilos de programación, pensados para que puedan hacer *scripts* personas de todo tipo de perfiles, desde no programadores hasta avanzados ingenieros software. La misma funcionalidad se puede obtener de diversas maneras, adaptándose a la manera de programar de cada usuario, o a su nivel de conocimientos. Normalmente, los usuarios más avanzados usan paquetes muy sofisticados y en pocas líneas de código logran una gran funcionalidad. Un buen ejemplo es el paquete `fastTextR` que proporciona la misma funcionalidad para dos perfiles de usuarios muy distintos: los que usan funciones, y los que usan orientación a objetos.

2.1 ¿Y qué pasa con Python?

Python está diseñado para programadores. En mi opinión, hacer *scripts* es algo que debería poder hacer cualquier persona con poco esfuerzo. Invocar una función es algo que entiende cualquiera, pero usar programación orientada a objetos requiere unos conocimientos que reducen significativamente el número de personas que lo pueden usar. La llamada 'barrera de adopción' tecnológica, esto es, el esfuerzo necesario para que una persona utilice una tecnologia es, en mi opinión, mayor para Python que para R.

3 Variables y tipos básicos

En R tienes los siguientes tipos básicos:

- enteros
- cadenas
- flotantes
- booleanos

La asignación de la variable se puede hacer usando `=` o `<-`. Por ejemplo:

```r
a <- 1
b <- 0.5
c <- "hola"
d <- TRUE
```

No hay que declarar los tipos de las variables, son dinámicos. En este caso usaremos la variable `a` para que guarde distintos tipos de datos, sobreescribiendo el valor anterior. La función `class(variable)` nos indicará el tipo de la variable.

```r
a <- 1
a
```

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

```r
class(a)
```

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

```r
a <- 0.5
a
```

```
## [1] 0.5
```

```r
class(a)
```

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

```r
a <- "hola"
a
```

```
## [1] "hola"
```

```r
class(a)
```

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

```r
a <- TRUE
a
```

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

```r
class(a)
```

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

3.1 Tipos derivados

Se pueden definir otros tipos. Por ejemplo, en el paquete base se definen los números complejos:

```r
a <- 3 + 2i
class(a)
```

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

Los operadores reconocen los tipos y actúan en consecuencia.

```r
a <- 3 + 2i
b <- 1 - 2i
dotimg <- a*b
dotimg
```

```
## [1] 7-4i
```

```r
class(dotimg)
```

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

Nota: recuerda que $(a, b) \cdot (c, d) = (ac - bd, ad + bc)$

3.2 Constantes

También se conocen las constantes π (como variable `pi`) y e (como resultado de `exp(1)`).

```r
cat(paste("Esta es pi=", pi))
```

```
## Esta es pi= 3.14159265358979
```

```r
cat(paste("Esta es e=", exp(1)))
```

```
## Esta es e= 2.71828182845905
```

La "ecuación más bella", la identidad de Euler ($e^{-i\pi} + 1 = 0$), en R sería:

```r
bella <- exp(-1 * (0 + 1i) * pi) + 1
bella #Observa que es un número imaginario
```

```
## [1] 0-1.224647e-16i
```

```r
Re(bella)#Esta es su parte real
```

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

```r
Im(bella)#Esta es su parte imaginaria
```

```
## [1] -1.224647e-16
```

(que "viene siendo" ≈ 0).

3.3 Números flotantes

Un número flotante es `a <- 1/3` (0.3333...). Los números flotantes en notación científica usan la letra `e`, pero OJO, no debe confundirse con exp(1) (el número e descrito en la sección de las constantes). Por ejemplo, 140 millones ($1.4 \cdot 10^8$) se escribe en notación científica `1.4e+8`.

```
cat(paste("140 millones=", 1.4e+8))
```

```
## 140 millones= 1.4e+08
```

```
cat(paste("140 millones (como entero)=", as.integer(1.4e+8)))
```

```
## 140 millones (como entero)= 140000000
```

En R todos los números (salvo los complejos) son de tipo `numeric`.

```
class(1.4e+8)
```

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

```
class(23)
```

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

```
class(1/3)
```

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

```
class(pi)
```

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

Por este motivo, podemos hacer un vector de números que tenga enteros, fracciones, flotantes o constantes.

```
c(1, 1/3, sqrt(2), 1.3, pi)
```

```
## [1] 1.0000000 0.3333333 1.4142136 1.3000000 3.1415927
```

3.4 Cadenas

```
cadenaUnica <- "hola"
nchar(cadenaUnica)#Número de caracteres de la cadena
```

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

El tipo de las cadenas es `character`

```r
class("hola")
```

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

Las cadenas pueden usar comillas dobles o comillas simples

```r
cadena1 <- "hola"
cadena2 <- 'hola'
cadena1 == cadena2
```

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

3.4.1 Operaciones con cadenas

Unimos cadenas con `paste()`. Por ejemplo:

```r
paste("Hola", "cara", "cola")
```

```
## [1] "Hola cara cola"
```

Se pueden transformar las cadenas (cambiar su contenido) con funciones como `toupper` (convertir a mayúsculas) o `tolower` (convertir a minúsculas).

```r
toupper("Hola")
```

```
## [1] "HOLA"
```

```r
tolower("NASA")
```

```
## [1] "nasa"
```

3.5 Otros tipos

R define en su paquete base otros tipos, tales como matrices, tablas de contingencia, data frames, fechas, y un largo etcétera. Aquí puedes ver un ejemplo que usa matrices, números enteros y bucles para crear una animación del conjunto de Mandelbrot en 16 líneas de código R.

3.6 Operadores y precedencia

Los operadores son intuitivos y no se te harán extraños si conoces C o Java, pero en R hay algunos más, algunos muy sofisticados (como el operador `%any%`, que permite crear nuevos operadores). Tienes una descripción completa aquí.

4 Vectores

R es un lenguaje orientado a vectores. La mayoría de funciones aceptan que sus argumentos sean vectores (cuando se pasa un único valor lo interpreta como un vector con un único elemento). Esto tiene la ventaja de que estructuras que requieren bucles (como `for`, `do/while`) se convierten en R en una única llamada a una función.

Por ejemplo, si creamos dos vectores de enteros:

```
a <- c(1, 3, 5, 7)
b <- c(0, 2, 4, 6)
```

Podemos sumarlos o restarlos:

```
a + b
```

```
## [1]  1  5  9 13
```

```
a - b
```

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

Podemos multiplicarlos o dividirlos:

```
a * b
```

```
## [1]  0  6 20 42
```

```
b / a
```

```
## [1] 0.0000000 0.6666667 0.8000000 0.8571429
```

Los vectores alojan elementos del mismo tipo (ver sección de tipos básicos), por ejemplo: un vector de enteros, o un vector de cadenas. Si quieres alojar elementos de distintos tipos tienes que usar listas. Estas listas pueden contener elementos complejos y otros contenedores. Por ejemplo, puedes tener una lista cuyos elementos sean listas de un en entero, un flotante y un vectore de enteros. Más detalle en la sección de listas.

Muy importante (y que choca a los programadores): los vectores **solo pueden ser de tipos básicos**. Por ejemplo, las fechas no son tipos básicos, por lo que no puedes tener un vector de fechas. El motivo es que los tipos básicos están ligados a los tipos de elementos que puedes leer de un fichero de datos. Una fecha es una estructura compleja que se almacena en fichero de forma variadas, por lo que no se puede leer del fichero de forma tan sencilla como un entero, un flotante o una cadena de texto. En R hay estructuras para contener fechas, e.g. `POSIXlt` o `POSIXct`, pero no podrás hacer vectores de estos elementos. Sí podrás hacer listas con estos elementos, y habrá funciones que puedan aplicarse a listas de estos elementos.

Hay funciones para generar vectores. Por ejemplo, `seq` crea una secuencia de valores:

```
seq(from=1, to=10, by=2)
```

```
## [1] 1 3 5 7 9
```

4.1 Funciones sobre vectores

La función `length()`, aplicada a un vector, devuelve el número de elementos del vector.

```
a <- c(1, 2, 3, 4)
length(a)
```

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

La función `sum()`, aplicada a un vector, devuelve la suma los valores.

```
a <- c(1, 2, 3, 4)
sum(a)
```

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

Podemos calcular la media, la desviación típica, la cuasivarianza (también llamada varianza muestral):

```
a <- c(1, 2, 3, 4)
mean(a)
```

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

```
sd(a)
```

```
## [1] 1.290994
```

```
var(a)
```

```
## [1] 1.666667
```

Observa que `var()` calcula la cuasivarianza (divide por `n-1`), no la varianza (divide por `n`). Esto suele ser motivo de confusión, como en este hilo de stackexchange en el que se indica que la librería numpy de Python usa `var()` para la varianza. En estos detalles se nota que R fue creado por expertos en estadística.

Hay funciones que devuelven vectores, como `cumsum()` (suma acumulada). Veamos un ejemplo:

```
a <- c(1, 2, 3, 4)
#El primer elemento es el primer valor, el segundo es la suma de los dos primeros valores, etc.
cumsum(a)
```

```
## [1]  1  3  6 10
```

4.2 Acceso, modificacion y eliminación de los elementos de un vector

Dado el vector de enteros siguiente:

```r
a <- c(5, 4, 3, 2)
```

¡OJO! A diferencia de otros lenguajes, como C o Java, en el que el primer elemento es el 0, **en R el primer elemento es el 1**.

Accedemos al primer elemento del vector con

```r
a[1]
```

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

Accedemos al último elemento del vector con

```r
a[length(a)]
```

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

Podemos acceder a varios elementos si, en lugar de un índice, indicamos un vector de índices.

```r
a[c(1,2)] #Valores de los elementos de los índices 1 y 2
```

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

En R hay un operador secuencia: `2:5` es equivalente a `c(2, 3, 4, 5)`, por lo que podemos hacer:

```r
a[2:4] #Valores de los elementos de los índices entre el 2 y el 4 (ambos incluidos)
```

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

4.2.1 Acceso usando condiciones

Se pueden obtener los elementos de un vector que cumplen cierta condición. Un ejemplo:

```r
a[a>3] #Valores de los elementos con valor mayor que 3
```

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

En R los operadores lógicos AND y OR se escriben con `&` y `|` respectivamente.

```r
a[a>=3 & a<=5] #Elementos con valor mayor o igual que 3 y menor o igual que 5
```

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

4.2.2 Concatenación de vectores

La función `append()` permite concatenar vectores (deber ser del mismo tipo, claro).

```r
vectorInicial <- c(5, 6, 7)
vectorNuevosElementos <- c(8, 9)
append(vectorInicial, vectorNuevosElementos)
```

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

4.2.3 Si sigues pensando en bucles

Si todavía te cuesta sacar los bucles de tu cabeza, puedes usar estructuras tradicionales. Un ejemplo:

```r
n <- 5
x <- vector(length=n)   #¡OJO! Creación dinámica de un vector
for (i in 1:n){         #Bucle for
    x[i] <- i
}
x
```

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

4.2.4 NULL, NA, NaN y otras lindezas

Los datasets del mundo real no son perfectos. Con frecuencia faltan valores (NA, del inglés "Not Available" (no disponible)) o son nulos (NULL). Cuando operamos con ellos obtenemos `Inf` (infinito), por ejemplo si hacemos 3/0, o `-Inf` (-3/0). Si hacemos 0/0 obtenemos `NaN` (Not a Number, en castallano "indefinido").

Para identificar estar lindezas en uestros datos, R proporciona funciones como `is.null(x)`, `is.na(x)`, `anyNA(x)`, `is.infinite(x)`, `is.nan(x)`.

Por ejemplo, un vector vacío `c()` está a NULL. Por tanto:

```r
is.null(c())
```

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

Podéis buscar estos "errores" en vuestros datos con operaciones como estas:

```r
datosEnBruto <- c(5, NA, 3, Inf)
is.na(datosEnBruto) #Retorna un vector de booleanos. TRUE --> tienen NA
```

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

¡¡OJO!! Los vectores pueden contener NA(s) pero no pueden tener NULL(s). Las listas sí pueden tener NA(s) y NULL(s). Vectores y listas pueden tener NaN(s), Inf(s) y -Inf(s).

Para pasar de vector de booleanos a vector de índices usa `which()`. Continuando el ejemplo anterior:

```r
datosEnBruto <- c(5, NA, 3, Inf)
which(is.na(datosEnBruto)) #Obtenemos los índices de los elementos que tienen NA
```

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

4.2.5 Eliminado elementos

Ahora que sabes los elementos "raros" de un vector de datos, quieres poder eliminarlos del vector. Un ejemplo:

```
datosEnBruto <- c(5, NA, 3, Inf)
toRemove <- which(is.na(datosEnBruto)) #Obtenemos los índices de los elementos que tienen NA
datosEnBruto[-toRemove]  #Fíjate en el signo menos.
```

```
## [1]   5   3 Inf
```

Observa que esto vale para eliminar **todos** los NAs de un vector. Mira este ejemplo:

```
datosEnBruto <- c(5, NA, 3, NA, 7, NA)
toRemove <- which(is.na(datosEnBruto)) #Obtenemos los índices de los elementos que tienen NA
datosEnBruto[-toRemove]  #Fíjate en el signo menos.
```

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

4.2.6 Asignando elementos

Si, en lugar de eliminar los NAs, quieres reemplazarlos por ceros, puedes hacer esto (asignar un valor a los elementos que cumpen cierta condición lógica):

```
datosEnBruto <- c(5, NA, 3, NA, 7, NA)
datosEnBruto[is.na(datosEnBruto)] <- 0 #Asigna el valor 0 a los elementos que cumplen la condición
datosLimpios <- datosEnBruto
datosLimpios
```

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

4.3 Poniendos nombres a los elementos de un vector

Los elementos de un vector pueden tener nombre. En el siguiente ejemplo usamos un vector de enteros, pero puede ser un vector de cadenas, numeric, etc.

```
vectorSinNombres <- c(1, 3, 7)
vectorConNombres <- c(a=1, b=3, c=7)
```

Para saber los nombres uso `names()`

```
names(vectorSinNombres) #Retorna NULL (no hay nombres)
```

```
## NULL
```

```
names(vectorConNombres)
```

```
## [1] "a" "b" "c"
```

Puedo asignar nombres con

```r
names(vectorSinNombres) <- names(vectorConNombres) # Al vector sin nombres le ponemos
                                                   #  los nombres del vector con nombres
names(vectorSinNombres)
```

```
## [1] "a" "b" "c"
```

Los nombres son un caso particular de los atributos que puede tener cualquier objeto R. La fución `attributes()` permite leer y asignar atributos

```r
attributes(vectorSinNombres) #Observa el $names
```

```
## $names
## [1] "a" "b" "c"
```

```r
attributes (vectorSinNombres) <- list(color = "rojo", valor = 7) #Resigno los atributos
                                                                 #  (pierde los antiguos)
attributes (vectorSinNombres)
```

```
## $color
## [1] "rojo"
##
## $valor
## [1] 7
```

5 Data frames

Un data frame (DF) es una "tabla de datos". Las columnas suelen tener nombre para poder referirnos a ellas por su nombre.

En R podemos saber qué datasets tenemos disponibles (aparte de los que creemos nosotros) con el comando `data(package = "nombre paquete")`. En particular, para el paquete `base` (que siempre tienes), podemos ver que:

```r
data(package = "base")
```

```
## Warning in data(package = "base"): datasets have been moved from package 'base'
## to package 'datasets'
```

A la vista del mensaje, lo preguntaremos de esta manera:

```r
data(package = "datasets")
```

Y obtendrás una lista parecida a esta:

objeto	Descripción
AirPassengers	Monthly Airline Passenger Numbers 1949-1960
BJsales	Sales Data with Leading Indicator
BJsales.lead (BJsales)	Sales Data with Leading Indicator

objeto	Descripción
BOD	Biochemical Oxygen Demand
CO2	Carbon Dioxide Uptake in Grass Plants
ChickWeight	Weight versus age of chicks on different diets
... y muchos más	

Empezaremos usando un dataframe muy especial: el que creó Ronald Aylmer Fisher (sí, el Fisher de la F de Fisher-Snedecor) y que le dio fama mundial en 1936. Puedes saber los detalles de este dataset si tecleas `?iris` en la consola de R (o de RStudio).

Podemos ver las primeras filas de la tabla con:

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Como esta salida es muy espartana, en este manual mostraremos una salida más "elegante" que hace más legible el resultado:

```
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Para saber el número de líneas del dataframe usaremos `nrow()`, y para saber el número de columnas `ncol()`. Para este dataset:

```
nrow(iris)
```

```
## [1] 150
```

```
ncol(iris)
```

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

Esta tabla muestra las medidas en centímetros de las variables longitud y ancho del sépalo y largo y ancho del pétalo, respectivamente, para 50 flores de cada una de las tres especies de lirios (iris en inglés, pero en latin se escribe igual). Las tres especies son *iris setosa*, *iris versicolor* y *iris virginica*.

Este dataset hasta tiene entrada en Wikipedia, donde podéis ver las flores objeto del estudio y muchos más detalles.

Puedo saber si hay algún NA en el dataset usando

```
any(is.na(iris)) #any() indica si algún elemento de la matriz es TRUE
```

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

5.1 Acceso a los valores

Puedo acceder a cualquier de la columnas de dos formas: (1) indicado el nombre de la columna, o (2) indicando el índice de la columna.

5.1.1 Acceso por nombre de la columna

```
mean(iris$Sepal.Length) #Valor medio de la columna Sepal.Length
```

```
## [1] 5.843333
```

5.1.2 Acceso por índice de la columna

```
mean(iris[ , 1]) # Observa notación [fila, col] y que fila está vacía (significa "todas").
```

```
## [1] 5.843333
```

5.1.3 Acceso a una celda concreta

Si queremos acceder al segundo elemento de la primera columna haremos:

```
iris[2, 1]
```

```
## [1] 4.9
```

5.1.4 Seleccionado usando lógica

Si queremos acceder al segundo elemento de la primera columna haremos:

```
iris[2, 1]
```

```
## [1] 4.9
```

5.2 Creando nuevas columnas

Puedo crear una nueva columna en el dataset que sea el producto de las columnas 1 y 2. A esta nueva columna la llamaremos Sepal.Area. Haz esto :

```
iris$Sepal.Area <-  iris$Sepal.Length * iris$Sepal.Width
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Area
5.1	3.5	1.4	0.2	setosa	17.85
4.9	3.0	1.4	0.2	setosa	14.70
4.7	3.2	1.3	0.2	setosa	15.04
4.6	3.1	1.5	0.2	setosa	14.26
5.0	3.6	1.4	0.2	setosa	18.00
5.4	3.9	1.7	0.4	setosa	21.06

En la tabla verás una nueva columna, a la derecha, llamada Sepal.Area.

5.3 Las columnas de factores

La columna `Species` no contiene números, sino una **variable categórica**. Podemos ver qué valores toma con `table` aplicado a la columna:

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

Vemos que la variable toma solo tres valores, y que cada una de ellos ocurre 50 veces. Si vemos el tipo de los datos de esta columna veremos que es de tipo `factor`.

```
class(iris$Species)
```

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

Un factor es un número, y se usa para evitar repetir cadenas de caracteres y ahorrar espacio en memoria. Puedo hacer comparación de factores. Por ejemplo, ¿son la misma especie el elemento 1 y el elemento 2?

```
iris$Species[1] == iris$Species[2]
```

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

Puedes ver en la tabla que ambas son `setosa`.

¿Son la misma especie el elemento 1 y el elemento 150?

```r
iris$Species[1] == iris$Species[150]
```

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

¿Cuál es la especie del elemento 150?

```r
iris$Species[150]
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

Los **levels** son los valores que puede tomar la variable categórica. Para obtener las cadenas de un vector de factores puedes usar `as.character(variableFactor)`. Por ejemplo:

```r
as.character(iris$Species)[1] #Convierto los factores a cadenas y muestro la primera
```

```
## [1] "setosa"
```

Observa las comillas que indican que es una cadena. Puedes ver su tipo con `class`:

```r
class(as.character(iris$Species)[1])
```

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

6 Listas

A diferencia de un vector, una lista puede tener elementos de distintos tipos

```r
lst <- list(3.14,                #El primer elemento es un flotante
            "Pepe",              #El segundo elemento es una cadena
            list('A', 'B', 'C'), #El tercer elemento es una lista de cadenas
            c(1,1,2,3)           #El cuarto elemento es un vector de enteros
           )
```

Inicialización de una lista (vacía) con

`lista <- list()`

Para acceder a los elementos de una lista podemos usar `[[i]]` o `[i]`. La primera forma nos devuelve el elemento i-ésimo. La segunda forma nos devuelve **una lista** que contiene el elemento i-ésimo.

```r
lst[[1]] #Retorna el elemento 1
```

```
## [1] 3.14
```

```r
class(lst[[1]]) #El tipo del elemento 1 (un entero)
```

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

```r
lst[1] #Retorna una lista con el elemento 1
```

```
## [[1]]
## [1] 3.14
```

```r
class(lst[1]) #El tipo de la lista
```

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

Igual que los vectores, las listas también pueden tener sus elementos nombrados. Por ejemplo:

```r
lstnom <- list(npi     = 3.14,
               nombre  = "Pepe",
               minilist= list('A', 'B', 'C'),
               vec     = c(1,1,2,3)
              )
```

Puedo acceder por el nombre del elemento de dos formas:

```r
lstnom[['npi']] #Retorna el elemento 1
```

```
## [1] 3.14
```

```r
lstnom$npi      #Retorna el elemento 1
```

```
## [1] 3.14
```

Para saber la longitud (el número de elementos) de una lista usamos, como para los vectores, la función `length()`

```r
length(lstnom) #Hay 4 elementos en la lista
```

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

7 Ingeniería

- En RStudio se puede *debuguear* (aunque no desde un fichero Rmd).
- En RStudio puedes hacer control de versiones creando un proyecto que use tu cuenta de Github.
- Experimentos reproducibles usando Rmd (R markdown).
- El paquete `shiny` permite hacer aplicaciones web de forma muy sencilla.
- El paquete `plumber` permite hacer servicios web se forma sencilla.
- Como entorno común de pruebas usa los escritorios UPM.

8 Más información

- Básico: Los manuales oficiales de CRAN: aquí. En español: aquí.
- Avanzado:
  - El lenguaje R *per se*, aquí.
  - El libro *online* de Hadley Wickham y Garrett Grolemund: aquí. Hay una versión en español: aquí.
- Muy avanzado: Los libros *online* recomendados por RStudio: aquí. Un clásico algo antiguo, pero muy sugerente y provocador: R inferno. La biblia de R (The big book of R)