BSM

Blockchain School
for Management

Tema 4

Módulo II: R. Funciones, Bucles, etc.

Nicolás Forteza

2022-11-09

Funciones en R

Hemos visto que en R, todo funciona con funciones, a las cuales les pasamos un argumento y nos devuelven un objeto.

En R, podemos crear nuestras propias funciones para usar R de manera eficiente y flexible.

Se suelen programar o codificar funciones si vamos a realizar un conjunto de pasos lógicos varias veces.

La sintaxis para crear funciones en R:

```
f <- function(arg1, arg2, ...) {
## codifica aquí lo que quieras hacer
}
```

- Las funciones en R pueden ser pasadas como argumentos a otras funciones.
- Puedes crear funciones dentro de otras funciones (*nested functions*)

Ejemplo de función

Vamos a crear una función que calcula el cuadrado del número que le pasas como argumento:

```
squared <- function(x){
  return(x * x)
}
squared(2)
```

```
[1] 4
```

Tenemos un objeto de clase `function`:

```
class(squared)
```

```
[1] "function"
```

Ejemplo de función

Si queremos invocarla, tenemos que pasarle los argumentos entre los paréntesis.

```
squared(c(2, 3))
```

```
[1] 4 9
```

Podemos especificar los argumentos o no al invocar la función.

```
squared(x=c(2, 3))
```

```
[1] 4 9
```

Ejemplo de función

Sin embargo, no se suele recomendar alterar el orden de los argumentos.

```
datos <- c(1:100)
sd(datos)
```

```
[1] 29.01149
```

```
sd(x=datos, na.rm = T)
```

```
[1] 29.01149
```

Para ver los arugmentos de la función, podemos ver la documentación pulsando `F1` mientras el cursor está encima de la misma, o bien hacer:

```
args(squared)
```

```
function (x)
NULL
```

Puedes también fijar argumentos a `NULL` si tu función lo precisa, así como también fijar valores por defecto si el usuario decide no pasar dichos argumentos. Incluso puede sincluir *booleanos*!

```
expfunc <- function(x, b=2) {
  # por defecto, eleva al cuadrado
  # excepto si lo indicas
  return(x**b)
}
expfunc(2)
```

```
[1] 4
```

```
expfunc(2, 3)
```

```
[1] 8
```

Hemos dicho que podemos crear funciones dentro de funciones:

```
make_power <- function(n) {
  pow <- function(x) {
    x^n
  }
  pow
}

cuadrado <- make_power(2)
cuadrado(3)
```

```
[1] 9
```

```
cubo <- make_power(3)
cubo(3)
```

```
[1] 27
```

Podemos usar también funciones cuya lógica interna usa variables del entorno:

```
y <- 10
g <- function(x) {
  x * y
}
f <- function(x) {
  y <- 2
  y^2 + g(x)
}
f(3)
```

```
[1] 34
```

Ejercicio

- Crear una función que devuelve un vector de unos, y cuya longitud está parametrizada por un argumento.
- Crear una función que suma dos vectores.
- Crear una función que imprime vuestro nombre en pantalla.

Podemos controlar el flujo de ejecucuón de sentencias mediante los if-else statements.

Sirven para crear programas y funciones con algo más de complejidad.

La sintaxis para escribir este tipo de sentencias:

```
if(<condition>) {
## haz algo aquí
} else {
## haz lo otro aquí
}
```

La clave de estas sentencias es que dependen de la condición en su totalidad.

Debemos tener muy presente los operadores *booleanos* y estar atentos a la lógica que estamos codificando.

Ejemplo If-else statements

Siguiendo con los ejemplos anteriores, vamos a crear un if-else:

```
x <- 4
if(x >= 5) {
  print("x es mayor o igual que 5")
} else {
  print("x es menor o igual que 5")
}
```

```
[1] "x es menor o igual que 5"
```

Obviamente que puedo guardar y crear variables dentro de un if-else.

Ejemplo If-else statements

Podemos incluirlas dentro de las funciones:

```
squared_if <- function(x, p=2){
  if (p < 10) {
    x ** p
  }
  else  {
    print("p es demasiado alto")
    x
  }
}
squared_if(3)
```

```
[1] 9
```

```
squared_if(3, 10)
```

```
[1] "p es demasiado alto"
```

```
[1] 3
```

Las condiciones pueden ser múltiples!

```
x <- 2
if(x>0 && x<5) {
  y<-x**2
} else {
  y<-x**3
}
print(y)
```

```
[1] 4
```

Y además podemos encadenar varios if-else:

```
x <- 3
if(x >= 4) {
  print("Mayor igual que 4")
} else if (x <= 2){
  print("Menor igual que 2")
} else {
  print("Está entre 2 y 4")
}
```

```
[1] "Está entre 2 y 4"
```

```
print(y)
```

```
[1] 4
```

Para escribir if-else de calidad, acordaros de las funciones:

```
is.numeric(x)
```

```
[1] TRUE
```

Todas las funciones que empiecen por `is.` nos devolverán un *booleano*, que en conjunto con otros, podremos crear if-else complejos.

Ejercicio

- Crea una función que suma 0.1 a un número si el número es mayor que 1, y resta 0.1 si es menor que 1.
- Aplica la función sobre un vector que tenga 10 valores entre 0 y 10. Sé creativo para crear el vector.
- Crea una función que eleve al cuadrado un número si éste es menor que 5, y aplícalo al vector.

Los bucles son usados para iterar sobre los elementos de un objeto (lista, vector, ...), y realizar *algo* con ese elemento.

La sintaxis de un bucle es:

```
# por cada elemento dentro de la secuencia
for(elemento in secuencia) {
  # realizar algo con ese elemento
}
```

El proceso acaba cuando ha iterado sobre todos los elementos.

Se pueden combinar con if-else statements. Con esto y las funciones, ya podemos realizar_ casi_ cualquier programa de R.

```
n <- c(-2:2)
for (number in n){
  # si es número par
  if ((number %% 2) == 0) {
    print(number)
  }
}
```

Podemos *parar* la ejecución del `for` con el comando `break`.

```
n <- c(-2:2)
for (number in n){
  # si es número par, el bucle se
  # para e imprime el número
  if ((number %% 2) == 0) {
    print(number)
    break
  }
}
```

Un ejemplo de bucle para ir modificando los elementos de un vector:

```
n <- c(-2:2)
for (i in c(1:length(n))){
  n[i] <- n + i
}
print(n)
```

Por convención, se usa la letra ´i´ para indicar el *índice* del elemento en el que estamos.

Ejercicio

- Crea un vector con números enteros, de longitud igual a 10. Quédate con los números pares y guárdalos en otro vector. Tienes que resolver esto de al menos 2 maneras diferentes.

Solución propuesta 1

```
v1 <- c(1:10)
isEven <- function(number){
  if ((number %% 2) == 0) {
    TRUE
  } else FALSE
}
v2 <- c()
for (i in c(1:length(v1))){
  if (isEven(v1[i])){
    v2[length(v2) + 1] = v1[i]
  }
}

print(v2)
```

```
[1]  2  4  6  8 10
```

Solución propuesta 2

```r
v1 <- c(1:10)
v2 <- v1[(v1 %% 2) == 0]
print(v2)
```

```
[1]  2  4  6  8 10
```

Podemos anidar bucles también. Por convención, usaremos la letra `j` en una segunda iteración, `k` en la tercera y así sucesivamente.

Esto sirve para iterar sobre una matriz, por ejemplo.

```
x<-matrix(1:6,2,3)
dims <- dim(x)
# por cada fila
for (i in c(1:dims[1])){
  # por cada columna
  for (j in c(1:dims[2]))
    print(x[i, j])
}
```

En la anterior slide, podríamos haber usado:

```
seq_len(nrow(x))
```

en vez de

```
i in c(1:dims[1])
```

Los bucles `while` evalúan una condición de manera repetida. Si la condición se cumple, se ejecuta la expresión que está dento del cuerpo del bucle. Si la condición no se cumple en algún momento, el bucle se para.

El bucle se podría estar ejecutando de manera indefinida!

Ejemplo:

```
count<-0
while(count<10) {
  count<-count+1
}
print(count)
```

```
[1] 10
```

Ejercicio

- Crea un bucle `while` que genere números aleatorios en su cuerpo y que pare si el número es mayor que 1.

Solución

```
while (TRUE) {
  x <- rnorm(1)
  print(x)
  if (x > 1) {
    break
  }
}
```

El comando `next` sirve para pasar a la siguiente iteración de un bucle.

Ejemplo:

```
for(i in 1:5) {
  if(i<=3) {
    next
  }
  print(i)
}
```

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

Ejercicio

- Realiza el mismo ejercicio que antes (Crea un bucle `while` que genere números aleatorios en su cuerpo y que pare si el número es mayor que 1), pero usa el comando `next` para que no imprima números negativos.

Solución

```
while (TRUE) {
  x <- rnorm(1)
  if (x < 0) {
    next
  }
  print(x)
  if (x > 1) {
    break
  }
}
```

Las funciones de tipo *apply* son funciones de R *base* que sirven para aplicar una función a cada uno de los elementos de una lista, vector, etc.

Ejemplo como el de antes:

```
v1 <- c(1:10)
v1 %% 2
```

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

En realidad, el comando `v1 %% 2` es una operación que **aplica** la comprobante (%%) a cada elemento de forma vectorizada.

Las funciones Apply hacen esto a cada elemento de otros objetos que no son vectores.

Tenemos diferentes tipos de apply:

- `apply` básico, útil para matrices y como veremos en el siguiente tema, data frames.
- `lapply`: devuelve una lista con la misma longitud que `X`, con cada elemento modificado con la `FUN` que le pasas como argumento.
- `sapply`: idónea para vectores o matrices.

```
v1 <- c(1:10)
sapply(v1, isEven)
```

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

Ejemplos con Apply

```
lapply(1:5, seq)
```

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

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3

[[4]]
[1] 1 2 3 4

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

```
lapply(list(a=1:10, b=rnorm(10)), mean)
```

```
$a
[1] 5.5

$b
[1] -0.1832037
```

Ejemplos con Apply

```
sapply(list(a=1:10, b=rnorm(10)), mean)
```

```
         a          b
 5.5000000 -0.1047979
```

Otros. Repeat.

Este es un bucle que se llevará a cabo el número de veces que especifiquemos, usando un `break` para detenerse. Si no incluimos un break, el bucle se repetirá indefinidamente y sólo lo podremos detener pulsando la tecla ESC, así que hay que tener cuidado al usar esta estructura de control.

```
val <-  0
vec <- NULL

repeat{
  val <- val + 1
  if(val == 5) {
    break
  }
}
```

Este tipo de bucle es el que menos se usa.

Ejercicio

Usa la función `apply` para lo siguiente: - Crea una matriz cuadrada y suma los elementos de cada fila. - Crea una matriz cuadrada y suma los elementos de cada columna.

Solución

```
m <- matrix(rnorm(9), 3, 3)
apply(m, 1, sum)
```

```
[1] -0.10132720 -0.07050803  0.61415311
```

```
apply(m, 2, sum)
```

```
[1]  3.0644080 -2.0074836 -0.6146065
```

Preguntas

PREGUNTAS???