AprendeR: Parte I

The UIB-AprendeR team

2020-03-05

Contents

Presentación

Edición preliminar en línea de la 1a parte del libro *AprendeR*, producido por un grupo de profesores del Departamento de Ciencias Matemáticas e Informática de la UIB.

El libro está escrito en *R Markdown*, usando *RStudio* como editor de texto y el paquete **bookdown** para convertir los ficheros *markdown* en un libro.

Este trabajo se publica bajo licencia Atribución-No Comercial-SinDerivadas 4.0

Chapter 1

Estructuras de control básicas

Una de las grandes ventajas de R es su capacidad de **vectorizar construcciones**, es decir, la posibilidad de aplicar de golpe una función a todo un vector, o a todas las entradas de un vector, o a todas las filas o columnas de un *data frame* o una matriz. Por ejemplo, para calcular la suma de todos los elementos de un vector, en muchos lenguajes de programación tendríamos que llevar a cabo un proceso recurrente como el que sigue:

1. Iniciar una variable "Suma" igual al primer elemento del vector
2. Para cada posición i entre 2 y la longitud del vector, redefinir "Suma" como la suma de su valor actual y el elemento i-ésimo del vector
3. Al llegar al final del vector, dar el valor final de "Suma"

En cambio, como sabéis, con R basta entrar `sum(vector)`.

De manera similar, para calcular todos los cuadrados de números naturales del 1 al 100, con R basta entrar `(1:100)^2`, mientras que en muchos lenguajes de programación tendríamos que llevar a cabo un proceso similar al anterior:

1. Definir un vector "Cuadrados" vacío
2. Para cada n entre 1 y 100, añadir al final del vector "Cuadrados" el valor de n^2
3. Al terminar, dar el contenido final del vector "Cuadrados"

Incluso para construcciones más complicadas que elevar al cuadrado, R dispone de funciones, genéricamente llamadas **funciones de tipo apply** que permiten aplicar una función a todos los elementos de un objeto. Algunas de estas funciones ya han aparecido en lecciones anteriores:

- `apply`, para aplicar una función a todas las filas o a todas las columnas de una matriz

- `aggregate`, para aplicar una función a todos los grupos formados al clasificar una variable según un factor
- `lapply`, para aplicar una función a todos los elementos de un objeto y dar el resultado en forma de *list*
- `sapply`, para aplicar una función a todos los elementos de un objeto y dar el resultado con una estructura lo más sencilla posible (por ejemplo, si el resultado es una lista de números, forma con ellos un vector)

Las normas usuales de "buena práctica de la programación" en R fomentan el uso de la vectorización de cálculos usando funciones de tipo *apply* u otras funciones específicas tipo `sum` cuando existen. Pero hay ocasiones en que es necesario, o nos puede convenir por algún motivo, escribir un pequeño **bucle** (un proceso que repite varias veces una misma secuencia de operaciones) del estilo de los que hemos explicado. En esta lección explicamos cómo usar en R los dos tipos de bucles más sencillos: los bucles de tipo **for** y de tipo **while**. Completamos la lección introduciendo otro tipo de estructura de control muy común: las estructuras condicionales.

1.1 Bucles de tipo "for"

Un **bucle de tipo "for"** es una estructura del tipo "Para todo ... haz ..." en la que se repite una determinada operación o secuencia de operaciones para todos los elementos de un vector (también puede ser de una list, pero para simplificar aquí siempre tomaremos un vector). Con R, este tipo de bucles se especifican con una instrucción de la forma

```
for(índice in vector){acción}
```

Esta sintaxis indica a R que

> para todo `índice` que pertenezca al `vector`, efectúe la `acción` especificada entre las llaves.

En esta instrucción el `índice` juega un papel de variable interna, como las variables entre los paréntesis en la definición de funciones.

Así, por ejemplo, para construir el vector de los cuadrados de números naturales del 1 al 10 por medio de un bucle de tipo "for" entraríamos

```
for(n in 1:10){Cuadrados[n]=n^2}
```

```
## Error in eval(expr, envir, enclos): objeto 'Cuadrados' no encontrado
```

Bueno, ya veis que no. Esta construcción solo implementa el paso 2 del proceso descrito en la introducción de la lección. Antes, hay que efectuar el paso 1: definir el vector de Cuadrados y darle un valor inicial, aunque sea igualarlo a un vector vacío, para que luego R lo vaya construyendo:

```
Cuadrados=c()  #Iniciamos Cuadrados como un vector vacío
for(n in 1:10){Cuadrados[n]=n^2}
Cuadrados
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

En vez de empezar con el vector `Cuadrados` vacío, podríamos iniciarlo con su primera entrada, 1. Y en vez de ir definiendo las entradas sucesivas de este vector en las diferentes iteraciones del bucle, también podríamos ir añadiéndolas al final:

```
Cuadrados=c(1)
for(n in 2:10){Cuadrados=c(Cuadrados,n^2)}
Cuadrados
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Desde el punto de vista de R, estos dos bucles de tipo "for" tienen un defecto: como cada iteración del bucle cambia la dimensión del vector `Cuadrados`, en cada iteración R define un nuevo objeto "Cuadrados" en el que copia el contenido del `Cuadrados` anterior y le añade el nuevo elemento. En bucles más complicados, esto podría ralentizar el cálculo. La manera correcta de definir este bucle en R es iniciar el vector `Cuadrados` como un vector de la longitud que va a tener al final, 10 en nuestro caso, y con el contenido que queramos, por ejemplo constante, y entonces ir modificando sus entradas una a una:

```
Cuadrados=rep(1,10)
for(n in 2:10){Cuadrados[n]=n^2}
Cuadrados
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Ejemplo 1.1. Estamos seguros de que conocéis la sucesión de Fibonacci F_n,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots,$$

donde cada término es la suma de los dos anteriores. Esto corresponde a la recurrencia

$$F_n = F_{n-2} + F_{n-1},$$

que, junto con las condiciones iniciales $F_1 = F_2 = 1$, determina completamente la sucesión:

$$\begin{aligned} F_3 &= F_1 + F_2 = 2 \\ F_4 &= F_2 + F_3 = 3 \\ F_5 &= F_3 + F_4 = 5 \\ &\vdots \end{aligned}$$

Vamos a calcular los 20 primeros números de Fibonacci, $(F_n)_{n=1,\dots,20}$. Para ello, usaremos un bucle de tipo "for" que traduzca la construcción "para cada n de 3 a 20, F_n es la suma de F_{n-1} y F_{n-2}". Como $F_1 = F_2 = 1$, iniciaremos el vector

que al final contendrá nuestros números de Fibonacci como un vector formado por veinte unos, y con el bucle redefniremos sus entradas a partir de la tercera:

```
Fib=rep(1,20)
for(n in 3:20){Fib[n]=Fib[n-1]+Fib[n-2]}
Fib
```

```
##  [1]    1    1    2    3    5    8   13   21   34   55   89  144  233  377
## [15]  610  987 1597 2584 4181 6765
```

Como en la definición de funciones, si la construcción que queremos realizar en cada paso de un bucle, sea de tipo "for" o, en la siguiente sección, de tipo "while", requiere de más de una instrucción, las podemos separar dentro de las llaves mediante signos de punto y coma o simplemente escribiéndolas en líneas aparte. Además, podéis anidar instrucciones `for` si necesitáis que el conjunto de índices sea multidimensional.

Ejemplo 1.2. Supongamos que queremos definir una función SimM que, aplicada a una matriz cuadrada A, la *simetrice*, es decir, substituya cada entrada (i,j) de A fuera de su diagonal principal por la media de las entradas (i,j) y (j,i). Para ello lo que haremos será, dada la matriz A, si n indica sus números de filas y de columnas:

1. Definir una copia AA de A
2. Para cada i entre 1 y n-1 y para cada j entre i+1 y n, redefinir AA[i,j] y AA[j,i] como (A[i,j]+A[j,i])/2
3. Dar como resultado la matriz simétrica AA resultante

```
SimM=function(A){
  n=dim(A)[1]
  AA=A
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      AA[i,j]=(A[i,j]+A[j,i])/2
      AA[j,i]=(A[i,j]+A[j,i])/2
    }
  }
AA
}
```

Veamos con un ejemplo si funciona:

```
A=matrix(c(1:9),nrow=3,byrow=TRUE)
A
```

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

```
SimM(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    3    5    7
## [3,]    5    7    9
```

Naturalmente, y como casi siempre con R, no hacía falta usar bucles de tipo "for" para definir esta función:

```
SimM2=function(A){(A+t(A))/2}
SimM2(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    3    5    7
## [3,]    5    7    9
```

1.2 Bucles de tipo "while"

Los **bucles de tipo "while"** repiten una determinada secuencia de operaciones mientras (*while*) una cierta condición lógica se satisfaga. Su sintaxis es

```
while(condición){acción}
```

e indica a R que

> antes de efectuar por primera vez la `acción` especificada entre las llaves y tras cada ejecución de la misma, compruebe si la `condición` especificada entre paréntesis se satisface; en caso afirmativo, vuelva a efectuar la acción, y en caso negativo pare la ejecución del bucle.

Por ejemplo, el bucle

```
Cuadrados=rep(1,10)
for(n in 1:10){Cuadrados[n]=n^2}
Cuadrados
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

se podría reescribir como un `while` de la manera siguiente

```
n=1 #Iniciamos un contador
Cuadrados=c() #Iniciamos el vector
while(n<=10){ #Mientras n sea menor o igual a 10...
  Cuadrados[n]=n^2 #añadimos n^2 al vector de Cuadrados...
  n=n+1 #y aumentamos en 1 el contador n
}
Cuadrados
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Este bucle:

- Como $1 \leqslant 10$, definirá Cuadrados[1]=1 y redefinirá n=2;
- Comprobará que $2 \leqslant 10$, y entonces definirá Cuadrados[2]=4 y redefinirá n=3;
- Comprobará que $3 \leqslant 10$, y entonces definirá Cuadrados[3]=9 y redefinirá n=4;
- ...
- Comprobará que $10 \leqslant 10$, y entonces definirá Cuadrados[10]=100 y redefinirá n=11;
- Comprobará que $11 > 10$ y parará.

Veamos otro ejemplo. Supongamos que queremos calcular los números de Fibonacci menores o iguales a 100. Podemos usar un bucle de tipo "while" de la manera siguiente:

1. Iniciaremos un vector con los dos primeros números de Fibonacci
2. Mientras la suma de los dos últimos números de Fibonacci calculados sea menor o igual a 100, añadiremos esta suma al final del vector, como siguiente número de Fibonacci

```
Fib=c(1,1)
while(Fib[length(Fib)]+Fib[length(Fib)-1]<=100){
  Fib[length(Fib)+1]=Fib[length(Fib)]+Fib[length(Fib)-1]
  }
Fib
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55 89
```

El siguiente número de Fibonacci ya sería mayor que 100:

```
Fib[length(Fib)]+Fib[length(Fib)-1]
```

```
## [1] 144
```

1.3 Estructuras condicionales

Las **estructuras de control condicionales** (o **condicionales** a secas) permiten que un programa decida de manera automática entre varias opciones en función de si se cumplen o no determinadas condiciones. Estas estructuras tienen en R la misma estructura que en casi todos los otros lenguajes de programación. Por un lado, tenemos la estructura

```
if(condición){acción}
```

que indica a R que

> si se satisface la `condición` especificada entre paréntesis, se lleve a cabo la `acción` especificada entre llaves y luego se continúe con el programa, mientras que si no se satisface dicha `condición`, se continúe con el programa sin efectuar la `acción`.

Por otro lado, tenemos la estructura

```
if(condición){acción1} else {acción2}
```

que indica a R que

> si se satisface la `condición` se lleve a cabo la `acción1` y si no se satisface dicha `condición` se lleve a cabo la `acción2` (y luego, en ambos casos se continúe con el programa).

Por ejemplo, si quisiéramos definir una función que valiera x^2 si $x \leqslant 0$ y x^3 si $x \geqslant 0$, tendríamos que usar un condicional:

```
f=function(x){
  if(x<=0){x^2} else {x^3}
}
f(-3)
```

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

```
f(3)
```

```
## [1] 27
```

Vamos a definir una función más complicada, que usará varios condicionales y un bucle. Se trata de una función que usa la **criba de Eratóstenes** (un número natural n mayor que 1 es primo si, y solo si, no es divisible por ningún número natural entre 2 y $n-1$) para decidir si un número natural es primo o no, indicándolo con los valores lógicos usuales `TRUE` y `FALSE`. En esta función:

1. En primer lugar usaremos un condicional para que si la entrada no es un número natural nos dé un mensaje de error y si es un número natural proceda con la criba de Eratóstenes.
2. A continuación, con un segundo condicional separaremos las entradas 0 y 1, que no son primos, del resto.
3. Luego, con un tercer condicional separaremos la entrada 2, que es un número primo, de las mayores o iguales que 3.
4. Finalmente, para un número $n \geqslant 3$, con un bucle de tipo "for" probaremos todos los divisores naturales entre 2 y $n-1$: si algún resto de la división entera da resto 0 (un cuarto condicional), el número entrado no es primo, y si todos los restos son diferentes de 0, sí que es primo.

```
Es.Primo=function(n){
  if (n!=round(n) | n<0){ #Si n no es entero o si es negativo...
```

```
    stop("El argumento no es un número natural")  #da error y para
  } else {
    if (n==0 | n==1){
          Primo=FALSE
    } else {
      Primo=TRUE
      if (n>=3){
           for(i in 2:(n-1)) {
          if ((n %% i) == 0) {
            Primo=FALSE
          }
}
}
}
Primo
}
}
```

Veamos un ejemplo que dé error:

```
Es.Primo(-3)
```

```
## Error in Es.Primo(-3): El argumento no es un número natural
```

Y ahora vamos a aplicar esta función a todos los números naturales entre 0 y 20. No, no usaremos un `for`, usaremos `sapply`.

```
sapply(0:20,FUN=Es.Primo)
```

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

1.4 Guía rápida de funciones

- `apply` aplica una función a todas las filas o a todas las columnas de una matriz
- `aggregate` aplica una función a todos los grupos formados al clasificar una variable según un factor
- `lapply` aplica una función a todos los elementos de un objeto y da el resultado en forma de *list*
- `sapply` aplica una función a todos los elementos de un objeto y simplifca la estructura el resultado
- `for(índice in vector){acción}` implementa un bucle de tipo "for"

- `while(condición){acción}` implementa un bucle de tipo "while"
- `if(condición){acción}` implementa un condicional "if... then"
- `if(condición){acción1} else {acción2}` implementa un condicional "if... then... else"
- `stop` para una ejecución y da un mensaje de error

1.5 Ejercicios

Test

(1) Dad el valor del término x_{100} de la sucesión x_n definida por $x_1 = 1$, $x_2 = 2$ y $x_n = 2x_{n-1} - x_{n-2} + \lfloor n/2 \rfloor$ para todo $n \geqslant 3$.

(2) Tenemos dos sucesiones x_n y y_n que evolucionan de manera conjunta siguiendo las ecuaciones $x_{n+1} = 2x_n - 3y_n$ e $y_{n+1} = x_n + (-1)^n y_n$. Si partimos de $x_1 = 10$ y $y_1 = 1$, ¿qué vale y_n para el primer n tal que $x_n > 10^6$?

Ejercicio

La **conjetura de Collatz** es una de las más misteriosas de la matemática actual. Dice lo siguiente.

> Empezando con un número natural mayor o igual que 1 cualquiera, iterad el proceso siguiente: si el número es par, lo dividimos por 2, y si es impar, lo multiplicamos por 3 y le sumamos 1, y vuelta a empezar. Esto proceso terminará por producir, más pronto o más tarde, siempre un 1.

Por ejemplo, empecemos con 7.

1. Como 7 es impar, lo multiplicamos por 3 y le sumamos 1: 22.
2. Como 22 es par, lo dividimos por 2: 11.
3. Como 11 es impar, lo multiplicamos por 3 y le sumamos 1: 34.
4. Como 34 es par, lo dividimos por 2: 17.
5. Como 17 es impar, lo multiplicamos por 3 y le sumamos 1: 52
6. Como 52 es par, lo dividimos por 2: 26.
7. Como 26 es par, lo dividimos por 2: 13.
8. Como 13 es impar, lo multiplicamos por 3 y le sumamos 1: 40.
9. Como 40 es par, lo dividimos por 2: 20.

10. Como 20 es par, lo dividimos por 2: 10.
11. Como 10 es par, lo dividimos por 2: 5.
12. Como 5 es impar, lo multiplicamos por 3 y le sumamos 1: 16.
13. Como 16 es par, lo dividimos por 2: 8.
14. Como 8 es par, lo dividimos por 2: 4.
15. Como 4 es par, lo dividimos por 2: 2.
16. Como 2 es par, lo dividimos por 2: **1**.

Fijaos que al llegar al 1, entraríamos en un bucle. Como es impar, lo multiplicaríamos por 3 y le sumaríamos 1, daría 4; como es par, lo dividiríamos por 2, daría 2; como es par, lo dividiríamos por 2, daría 1; y volveríamos al 4.

(1) Definid una función que, aplicada a un número natural no nulo n, produzca la secuencia que se obtiene con este procedimiento empezando con n hasta llegar a un 1.

(2) Definid una función que, aplicada a dos números naturales no nulos n y N, produzca la secuencia que se obtiene con este procedimiento empezando con n hasta llegar a un 1, o que pare si han efectuado N pasos sin alcanzarlo y entonces escriba en la consola "Tras N pasos no hemos llegado al 1" (con N el número que se ha entrado). Para esto último podéis usar la función `paste`; consultad su Ayuda. Además, la función ha de dar algún mensaje de error si la n a la que se aplique no es un número natural mayor que 0.

(3) ¿Cuál es el primer número natural para el que, tras 250 pasos, no se ha llegado a 1?

Respuestas al test

(1) 84576

(2) 156941