Ciclos de Iteración en R

true

Septiembre, 2016

Contents

Instrucciones

Lee este pequeño breviario y resuelve los ejercicios. Deberás entregar las soluciones a más tardar el día Lunes 12 de Septiembre. Para entregar seguiremos el siguiente protocolo

- Crearás un archivo con extensión .R para trabajar los ejercicios.
- Dentro del mismo resolverás cada ejercicio enmarcándolo entre comentarios con el signo #. Incluirás tu nombre y otras credenciales al inicio del documento, también enmarcdado entre comentarios. Mi archivo .R comenzaría así

```
# --- Alumno : Nelson Muriel
# --- Otras cosas sobre mí aquí


# --- Ejercicio 1
for ( i in ...){

}

# --- fin Ejercicio 1

# --- Ejercicio 2
Solución
# --- fin Ejercicio 2
```

- Deberás imprimir este archivo para su entrega.

El ciclo `while`

El ciclo `while` ejecuta una orden (`expr`) mientras se satisfaga una condición (`cond`). Funciona como en otros lenguajes de programación, iterando la expresión mientras la condición sea cierta.

Hay que tener especial cuidado en asegurar que la condición se hará falsa en algún momento para que el ciclo `while` termine exitosamente. De otro modo, el ciclo se vuelve infinito.

La receta es

```
while(cond){
  expr
}
```

Es buena práctica redactarlo así, con la llave que abre y la llave que cierra ocupando cada una, una línea.

Por ejemplo, el código siguiente es la estructura básica para analizar la variable `speed`. Debe imprimir *Bájale* en caso que tu velocidad supere los **30 km/h**. Como buen conductor, haces caso y disminuyes 7 km/h. El ciclo monitorea tu velocidad hasta que es inferior que los 30 km/h.

Si la variable `speed` no existe antes de ejecutar el ciclo `while`, la condición es inmediatamente falsa y no hay ciclo.

Ejercico 1 Corrige la condición e incluye el mensaje

```
speed <- 90

while(speed > 60){
  print("___")
  speed <- speed -7
}

speed
```

El último llamado a `speed` imprime el valor en que la condición se volvió falsa, el de salida del ciclo while.

Ejercicio 2 Utliza lo que sabes de los operadores de control `if`, `else` y `else if` para modificar el ciclo anterior. Ahora

- La velocidad inicializa en 64
- Si la velocidad supera los 48, R debe imprimir "Bájale pero mucho!", y baja la velocidad por 11 km/h
- En otro caso, que imprima "Bájale", y baja la velocidad por 6 km/h. Utiliza este código como guía. Hemos usado la función `paste` para pegar un mensaje con una variable. `?paste`.

```
speed <-
while (speed > ... ){
  print(paste("Tu velocidad es", speed))
  if (....){
   ...}else{
     ...
   }
}
```

Como sabemos, hay situaciones que ameritan acelerar a tope! Podría ser un desastre del que tenemos que escapar. En el siguiente ejercicio, indicarás al ciclo while que debe abortar si la velocidad es mayor que 80. Esto se logra con la instrucción `break`.

Ejercicio 3 Modifica el ciclo que hiciste en el ejercicio anterior de modo que abandone si la velocidad supera los 80 km/h. Inicializa la variable `speed` en 86 y mira lo que sucede. Qué sucede si inicializas en `speed <- 80`? Deberías saberlo sin correr el script.

Ejercicio 4 Escribe un ciclo `while` que monitoree el valor de la variable `i`, y tal que

- Monitoree el valor de i mientras sea menor o igual que 10
- Imprima el triple de `i`, es decir `3*i` y aumente el valor de `i` por 1

- Se abandone con `break` si el triple de `i` es divisible por 8 (recuerda que `x %% y` devuelve el residuo de la división de x entre y, cuánto debe valer? Utiliza `==` para verificar esta condición.) pero que de todos modos imprima el triple de `i` antes de salir del ciclo.

Después de ese ejercicio, estás listo para el ciclo `for`.

El ciclo `for`

La sintáxis general es

```
for ( v in V){
  expr
}
```

El objeto `V` puede ser *cualquier cosa*, un vector, una matriz, una base de datos, una lista... **cualquier** cosa en R. La variable `v` sólo toma valores y sólo se actualiza dentro del ciclo `for` y no se inicializa fuera del mismo. Por ejemplo, los dos códigos siguientes hacen lo mismo

```
for (i in 1:10){
  print(i)
}

### ------

i <- 54
for ( i in 1:10){
  print(i)
}

i
```

Como ves, tras el ciclo `for` la variable `i` no ha cambiado de valor, sigue siendo 54. Esto se llama `scoping` y se refiere al ambiente en el que están definidas las variables. La variable `i` del ambiente global `i <- 54` **no** es la misma que la variable `i in 1:10` del ciclo `for`. Esta segunda **no** se almacena en memoria y sólo se usa durante la ejecución del ciclo.

Hay dos formas de proceder en el ciclo `for`. La primera es usar la expresión general y la segunda es *indexando*, por ejemplo, los siguienes tres ciclos hacen lo mismo.

```
primos <- c(2, 3, 5, 7, 11, 13)

# versión 1
for (p in primos){
  print(p)
}

# versión 2
for(i in 1:length(primos)){
  print(primos[i])
}

# versión 3
for(i in seq_along(primos)){
```

```
  print(primos[i])
}
```

La única diferencia entre las primeras dos versiones es la forma de llamar a `primos`. En la primera versión llamamos directamente a sus elementos con `p in primos` y en la segunda versión llamamos *indexando* por entradas con `i in 1:length(primos)`. Como habrás adivinado `length(x)` devuelve la longitud de un vector.

La tercera versión es idéntica a la segunda salvo que `seq_along(x)` es una forma más *robusta* de hacer `1:length(x)`. Esto porque en caso que `x` no tenga elementos, `seq_along` reacciona mejor.

Ejercicio 5 Considera el vector

```
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
```

Escribe un ciclo `for` en cada una de las tres versiones que imprima cada elemento de `lindekin` por separado.

Hacer un ciclo por una lista es similar, por ejemplo

```
primos.lista <- list(2,3,5,7,11,13)

# versión 1

for(p in primos.lista){
  print(p)
}

# versión 2

for(i in 1:length(primos.lista)){
  print(primos.lista[[i]])
}

# versión 3

for(i in seq_along(primos.lista)){
  print(primos.lista[[i]])
}
```

Observa que tienes que utilizar `[[ ]]` para seleccionar de la lista.

Ejercicio 6 Considera la lista

```
nyc <- list(pop = 8405837,
            suburbios = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),
            capital = FALSE)
```

Imprime sus elementos de las tres formas.

Ejercicio 7 En este ejercicio, la matriz `gato` representa el estado de un juego de gato. Marca con una "X" y una "O" donde hay estas jugadas y con "NA" donde todavía no se ha jugado. Define esta matriz en tu área de trabajo

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

```
## [1,] "O"  NA   "X"
## [2,] NA   "O"  "O"
## [3,] "X"  NA   "X"
```

Harás un `for` dentro de otro `for` para iterar en esta matriz. Completa el siguiente código para que el ciclo exterior vaya sobre los renglones con índice `i` y el ciclo interior vaya sobre las columnas con índice `j`. Puedes usar las funciones `nrow` y `ncol`.
El ciclo debe imprimir una oración como "En el renglón 1 columna 1 el juego contiene O" para cada posición posible.

```
for ( ... ) {
  for ( ... ){
    print(paste('En el renglón', ..., 'columna', ... , 'el juego contiene', gato[i,j]))
  }
}
```

Ejercicio 8 Volvamos al vector `linkedin` que contiene las vistas a tu perfil de LinkedIn durante una semana. Escribe un ciclo `for` con un controlador `if` dentro y tal que

- Avance sobre los elementos del vector `lindekin`
- Si el elemento en turno excede el valor 9, imprima "Venga, fuiste popular!"
- En otro caso, que imprima "Hazte notar más"
- En todos los casos imprime el número de visitas después del mensaje

El comando `break` funciona dentro del ciclo `for` tal y como funcionaba en el ciclo `while`. Hay otro comando útil en este caso y es `next` que se salta la evaluación actual y sigue iterando con las siguientes.

Ejercicio 9 Extiende el ciclo `for` del ejercicio anterior. Añade estos dos comportamientos

- Si el valor excede el 16, imprime "Esto ya es demasiado, me largo" y logra que R abandone el ciclo
- Si el valor es inferior a 5 imprime "Vergonzoso..." y salta a la siguiente iteración sin imprimir el valor (por pena).

Intenta no repetir tus órdenes en el ciclo. En particular, deberás escribir la instrucción `print(i)`, donde `i` es la variable del ciclo, una única vez.

Ejercicio 10 Utiliza estas líneas para definir las variables iniciales en este ejercicio

```
rquote <- "los caminos internos de r son irrefutablemente intrigantes"
chars <- strsplit(rquote, split = "")[[1]]
```

El vector `chars` contiene cada letra de la frase por separado como resulta de la función `strsplit`. Escribe un ciclo `for` que cuente el número de letras r que aparecen antes de la primera letra u. Para ello

- Inicializa una variable `cuenta.r` con el valor 0
- Desplázate sobre `char in chars`
- Si `char == r`, incrementa el valor de `cuenta.r` por una unidad.
- Si `char == u`, sal del ciclo
- Por último, imprime la variable `cuenta.r` y verifica que la cuenta sea correcta.

lapply()

Visita, para empezar `?lapply`. Verás que el uso de esta función sugiere `lapply(x, FUN, ...)`. En general, `lapply` toma un vector o lista `x` y aplica la función `FUN` a cada uno de sus elementos. Se pueden usar funciones predefinidas en R o funciones definidas por el usuario. En caso de que la función utilice argumentos extras, se pueden pasar a `lapply` como argumentos opcionales `(...)`.

El resultado de `lapply()` es una **lista** de la misma longitud que `x`, donde cada elemento es el resultado de aplicar `FUN` . Es muy similar a un ciclo `for`... pero sin escribirlo !

Prueba los siguientes dos códigos para comenzar. Con ellos pasamos un vector tipo `character` de mayúsculas a minúsculas con la función `tolower`.

```
nombres <- c("GAUSS", "BERNOULLI", "KOLMOGOROV")

## versión for
for (i in seq_along(nombres)){
  print(tolower(nombres[i]))
}

## versión lapply
minusculas <- lapply(nombres, tolower)
minusculas
str(minusculas)
```

La diferencia es que `x` resulta en una lista, pero la operación se realiza de forma exactamente igual.

Ejercicio La siguiente función selecciona el primer elemento de un vector.

```
select_prim <- function(x){
  x[1]
}
```

Considera el siguiente código

```
grandes <- c("GAUSS:1777", "BERNOULLI:1700", "KOLMOGOROV:1903")
split <- strsplit(grandes, split=":")
split_min <- lapply(split, tolower)
```

Aplica la función `lapply()` a la lista `split_min` para extraer sólo los nombres de los grandes. Asigna el resultado al objeto `nombres`. Después, define una función para seleccionar el segundo componente de un vector y utilízalo sobre `split_min` para obtener una lista con los años.

Una opción en `lapply()` es utilizar funciones **anónimas**. Esto significa que son funciones que sólo se definen dentro del propio llamado a `lapply`. Por ejemplo, los siguientes son equivalentes

```
## versión con función pre-definida
triple <- function(x){ 3*x}
lapply(list(1,2,3), triple)

## versión con función anónima
lapply(list(1,2,3), function(x) {3*x} )
```

Ejercicio Convierte los llamados del ejercicio anterior a `lapply()` a llamados con función anónima.

La función `lapply()` permite pasar argumentos extras. Siguiendo con el ejemplo de la función `triple` podemos definir otra, que multiplica por un factor específico.

```
multi <- function(x, factor){
  x*factor
}

multi(3, 4)
[1] 12

lapply(list(1,2,3), multi, factor = 4)
```

Observa cómo pasamos el argumento extra, después del argumento `FUN` e incluyendo el nombre espcífico, en este caso `factor`.

Ejercicio Escribe una función llamada `select_ind` que tenga dos argumentos `x` e `index` y que seleccione el componente `x[index]`.

Aplica esta función al vector `split_min` para recuperar los nombres y los años pasando en cada caso el valor apropiado de `index`.

`sapply()`

Es una función similar a `lapply()`, cuyo primer argumento es un vector o lista `x` seguida de una función `FUN` con argumentos potenciales `( ... )`.

La diferencia es que la `s` significa **simplificado**. La función `sapply()` intenta simplificar el resultado del llamado a `lapply()`. Internamente, `sapply()` primero llama a `lapply()` y después intenta desplegar el resultado de forma simple. En general esta simplificación resultará en un vector o una matriz... cuando sea posible...

Trabajarás con la lista `temp` definida como

```
temp <- list( c(3,6,9,7,-3),
              c(6,12,13,4,9),
              c(4,-1,-3,8,7),
              c(1, 7, 2,3, 4),
              c(5,9,8,4,5,-1),
              c(-3,0,3,-1,5,6),
              c(3,6,9,12,4,8) )
```

La lista contiene cinco mediciones de temperatura para siete días distintos en una localidad dada.

Ejercicio Utiliza `lapply()` para calcular le temperatura mínima cada día con la función `min()`. Haz lo mismo pero en vez de utilizar `lapply()` utiliza `sapply()`. Compara los resultados. Repite los pasos anteriores para la temperatura máxima.

Ejercicio Define una función de nombre `prom_extremos` que promedie el mínimo y el máximo de un vector x. Comienza por

```
prom_extremos <- function(x){
  (....) / 2
}
```

Aplica esta función a `temp` con `lapply()` y con `sapply()` y compara los resultados. Ves la diferencia?

Ejercicio Utiliza la función `extremos` definida como

```
extremos <- function(x){
  c(min = min(x), max = max(x))
}
```

Aplícala con `lapply()` y con `sapply()` al vector de temperaturas. Observas que el haber nombrado los componentes como `min` y `max` ayuda a leer la información generada por la segunda funcón?

Ejercicio Define la función `bajo_cero` de tal forma que dado el vector `x` devuelva todos los días en que `x<0`. Aplica esta función con `lapply()` y con `sapply()` al vector `temp`. Alguna diferencia?

El ejercicio anterior muestra un caso en que la simplificación no es posible. Esto se debe a que el resultado de la función `bajo_cero` no tiene un formato fijo. Para ciertos días devuelve un vector de longitud 0, para otros de longitud 2, etc. Esta diferencia entre los resultados impide la simplificación y en est caso `sapply()` se revierte a `lapply()` automáticamente.

Prueba este código

```
sapply(list(rnorm (100), rnorm (100)),
       function(x) c(min = min(x), media = mean(x), max = max(x)))
```

Ejercicio Modifica el código anterior poniendo nombres a las variables de la lista. Haz que la primera se llame `X` y la segunda se llame `Y`. Cambia también la función para que calcule la mediana en vez de la media e incluye el rango intercuantil antes del máximo con `IQR()`. Notas lo simple que hace la lectura `sapply()`?

Recuerda: Aunque `sapply()` puede facilitar la lectura de un resultado, corres el riesgo de creer que la simplificación es posible cuando no lo es y por lo tanto, corres el riesgo de que el resultado no sea lo esperado. Si estás utilizando R para un análisis interactivo en el que puedes ver lo que resulta a cada paso, esto no es gran problema. No obstante, si incluyes `sapply()` dentro de una función escrita por tí, sí puede ser un problema.