Estructuras de control

David García Callejas

Hasta ahora hemos visto operaciones generales con datos, sobre todo organizados en dataframes. Este tipo de operaciones nos permiten trabajar con todas las filas a la vez, o podemos seleccionar una observación concreta para modificarla, etc. Pero en muchas ocasiones nos interesa repetir operaciones complejas en cada una de nuestras observaciones. Los lenguajes de programación incluyen lo que se llama "estructuras de control" que nos permiten decir qué órdenes queremos aplicar a diferentes conjuntos de datos.

Estructura condicional

La estructura más sencilla es la estructura condicional `if`. Si se cumple una cierta condición, ejecuta unas líneas de código. Si no se cumple (`else`), ejecuta otras. El código que se ejecuta si se cumple la condición (o si no se cumple) va dentro de llaves, de manera que la sintaxis de esta estructura es:

if(*condición*){

código si se cumple

}

o si queremos ejecutar un código alternativo si no se cumple la condición:

if(*condición*){

código

}else{

código alternativo

}

```
# dataframe sencillo como ejemplo
d1 <- data.frame(caracter = c("a","b","c"), valor = c(1,2,4))

# si el caracter d es uno de los valores en la columna "caracter",
# sacamos un mensaje por pantalla. En caso contrario, sacamos otro mensaje.
if("d" %in% d1$caracter){
  cat("el caracter d está incluido en d1")
}else{
  cat("d no está en d1")
}
```

La utilidad de esta "condicionalidad" es enorme. En la estructura de la función `if` puede ir cualquier condición lógica, que devuelva TRUE o FALSE. Por ejemplo, si leemos un conjunto de datos, podemos hacer diferentes operaciones con él dependiendo de si tiene NAs o no, de si tiene una columna determinada...

```
eq <- read.csv2(file = "../data/Earthquake_data.csv",
                dec = ".",
                header = TRUE,
                stringsAsFactors = FALSE)
```

```r
# nombre de la columna que nos interesa
mi.columna <- "Dep"

# creamos una variable vacía para almacenar
# las operaciones que haremos
resultado <- NA

# esta es una manera estándar de comprobar si una columna está en un dataframe
if(mi.columna %in% names(eq)){

  # si tenemos esta columna, podemos tener otra condición dentro de este código
  # esta condición pregunta literalmente
  # "si la suma de NAs en la columna Dep es igual a cero"
  if(sum(is.na(eq[,mi.columna])) == 0){

    resultado <- mean(eq[,mi.columna])

  # si no se cumple la condición es que hay al menos un NA en la columna Dep
  }else{

    # ¿cuántos NA?
    num.nas <- sum(is.na(eq[,mi.columna]))
    # y la condición contraria, ¿cuántos no son NA?
    datos.validos <- sum(!is.na(eq[,mi.columna]))
    # lo sacamos por pantalla
    cat(paste("existen",
              num.nas,
              "NAs en los datos, la media se calculará con",
              datos.validos,
              "observaciones"))

    # el argumento na.rm elimina los NA antes de calcular la media
    resultado <- mean(eq[,mi.columna],na.rm = TRUE)
  }# aquí acaba el segundo if-else

}else{ # si no se cumple la primera condición es que no existe la columna Dep
  cat(paste("no existe la columna",mi.columna,"en los datos leidos."))
}# aquí acaba el primer if-else

cat(paste("el resultado es",resultado))
```

Bucles `for`

En el ejemplo anterior, estamos operando con una sola columna que hemos seleccionado previamente. Esto nos permite automatizar muchas tareas, y evitar tener que repetir a mano el cálculo de la media, o lo que nos interese, con diferentes columnas o tablas. Pero también existen estructuras de control que nos permiten *repetir* un cálculo todas las veces que nos sea necesario en un solo dataframe. Por ejemplo, hacer la media de *todas* las columnas numéricas, una tras otra. Estas repeticiones se consiguen en R de varias maneras, pero una de las más importantes son los *bucles*. Veamos un ejemplo sencillo

```r
for(i in 1:5){
  cat(paste("el contador tiene valor",i,"\n"))
}
```

¿Qué hace este código? Vayamos paso por paso. La idea principal de los bucles (en este caso, un bucle *for*), es la de repetir una serie de órdenes un número de veces determinadas. El número de veces que repetimos el código viene dado en la primera línea, `for(i in 1:5)`, y el código que queremos repetir viene, como en un `if`, entre llaves. La primera línea, entonces, es donde definimos el bucle. Esta línea, en este primer ejemplo, se lee literalmente como "mientras una variable i va de 1 a 5, repite el código entre llaves". Para definir un bucle, tenemos que pensar en un "contador" asociado al bucle, que nos indica cuántas veces queremos repetir el código interno. En este caso, ese contador lo llamamos "i", y decimos que va de 1 a 5 (1:5). R nos da libertad absoluta para llamar a nuestro contador como queramos, y poner el rango que nos interese.

```
for(contador in 1:10000){
  # repetimos 10000 veces un cierto código
  # con una variable interna que se llama "contador".
}
```

Como habéis visto en el primer ejemplo, dentro del código del bucle podemos usar el contador interno (i) como una variable más. Este contador tomará un valor diferente en cada "vuelta" o "iteración". Además, este contador sólo toma valores diferentes dentro del código del bucle. Fuera del bucle el contador sigue existiendo como variable, y toma el último valor que tenía en el bucle (en el caso de "i", este valor es 5. Podéis mirar en la pestaña "environment" para comprobarlo). En general, debéis evitar confusiones entre variables contador, que actúan dentro de los bucles, y otras variables. Dicho con otras palabras, NO uséis variables contador fuera de su bucle.

Los valores que toma nuestra variable contador no tienen porqué ser enteros. Si tenemos un vector de cualquier tipo, podemos usarlo

```
# un vector numérico
vector.num <- c(2,6.7,333,80)

for(i in vector.num){
  cat(paste(i,"\n"))
}

# o un vector de caracteres
vector.char <- c("a","b","c")

for(i in vector.char){
  cat(paste(i,"\n"))
}
```

Ya podemos empezar a intuir el potencial de los bucles para automatizar operaciones en filas o columnas. Si queremos repetir un código determinado para todas las filas/columnas de nuestro dataframe, sólo necesitamos definir un contador que vaya de la fila 1 al final de los datos. Para el siguiente ejemplo, usamos de nuevo los datos de Star Wars. Queremos saber cuántas naves conduce cada uno de los 87 personajes de nuestros datos.

```
naves_SW <- read.csv2("../data/starwars_personajes_naves.csv",
                      header = TRUE,
                      stringsAsFactors = FALSE)
head(naves_SW)

# las funciones nrow() y length() nos devuelven el número de
# filas y columnas de un dataframe, respectivamente
num.personajes <- nrow(naves_SW)

# este será mi vector de resultados. Un elemento por cada personaje,
# que almacene el número de vehículos que conduce.
num.naves <- numeric(num.personajes)
```

```r
# recordad que un vector también puede tener nombres asociados a sus elementos
names(num.naves) <- naves_SW$name

# el vector se inicia por defecto a cero
head(num.naves)

# para cada personaje, calculamos cuántas naves conduce
# sumando los valores TRUE de su fila.
# y guardamos ese resultado en el vector "num.naves"
for(pers in 1:num.personajes){
  # puedo usar el contador "pers"
  # para trabajar en la fila adecuada del dataframe.
  # a la vez, selecciono las columnas que me interesan
  # recordad: dataframe[fila(s),columna(s)]
  mis.naves <- naves_SW[pers,2:11]

  # usamos la función rowSums, porque la variable "mis.naves"
  # es también un dataframe
  # fijaos que podemos usar nuestro contador "pers" para decir
  # en qué posición del vector de resultados guardamos los datos.
  num.naves[pers] <- rowSums(mis.naves)
}

head(num.naves)
num.naves["Chewbacca"]
```

Familia de funciones apply

El objetivo de este curso no es que salgáis manejando los conceptos de programación en R perfectamente... eso es imposible de todas formas, aun después de muchos años de uso, todos cometemos errores a diario. Antes bien, lo que intentamos con estos apuntes es que conozcáis las diferentes herramientas que podemos usar en R para tratar diferentes problemas. Así que no os preocupéis si no entendéis todo a la primera: el único secreto es, como todo en la vida, la práctica. Si acabáis usando estas herramientas en vuestro trabajo, acabaréis naturalizándolas. Así pues, ahora os presento una manera alternativa, y muy usada en R, de hacer operaciones sobre múltiples filas o columnas a la vez.

Para ello, primero tenemos que familiarizarnos con la "vectorización". Vectorizar una operación es, simplemente, realizarla de manera automática en todos los elementos de un vector. R incorpora operaciones vectorizadas en muchos casos.

```r
mi_vector <- 1:10
mi_vector

# esta operación se aplica de manera automática
# a todos los elementos del vector
# esto es vectorización
resultado <- mi_vector^2

# si R no tuviera esta funcionalidad,
# podríamos crear un bucle para hacer el mismo cálculo
resultado_for <- numeric(length(mi_vector))
# calcula el cuadrado de cada elemento y lo guarda en "resultado_for"
for(i in 1:length(mi_vector)){
  resultado_for[i] <- mi_vector[i]^2
```

```
}

resultado
resultado_for
```

R permite vectorizar con operaciones más complejas que las aritméticas. Para ello existe la familia de funciones `apply`. Estas funciones toman una matriz o un dataframe, y aplican la función que queramos a todas sus filas o columnas. Podemos ver un ejemplo sencillo con la propia función `apply`.

```
mi.matriz <- matrix(data = c(1,2,3,4,5,6,7,8,9),nrow = 3)
mi.matriz

# el argumento MARGIN especifica si queremos aplicar
# una función sobre cada fila (1) o sobre cada columna (2)
suma.filas <- apply(mi.matriz, MARGIN = 1, FUN = sum)
suma.columnas <- apply(mi.matriz, MARGIN = 2, FUN = sum)
suma.filas
suma.columnas

# otro ejemplo
media.filas <- apply(mi.matriz,MARGIN = 1, FUN = mean)
media.filas
```

Existen varias funciones dentro de la familia `apply`, pero todas siguen el mismo concepto. La diferencia radica en el tipo de datos que devuelven (`apply` devuelve un vector, otras devuelven una lista o un dataframe), y el tipo de datos con el que trabajan (`apply` se usa sobre matrices, otras sobre listas o sobre dataframes).

Una de las más comunes es `lapply`, que permite hacer operaciones sobre columnas de dataframes y devuelve una lista (ver, por ejemplo, https://bookdown.org/jboscomendoza/r-principiantes4/lapply.html).

```
# usamos de nuevo los datos de terremotos
eq <- read.csv2(file = "../data/Earthquake_data.csv",
                dec = ".",
                header = TRUE,
                stringsAsFactors = FALSE)

# queremos calcular la profundidad media y la magnitud media usando lapply
eq.clean <- eq[,c("Dep","M")]
head(eq.clean)

mean.values <- lapply(eq.clean, FUN = mean, na.rm = TRUE)
mean.values

# Fijáos que no hemos necesitado eliminar los NAs de los datos.
# La función "mean" tiene un argumento na.rm
# que permite eliminarlos automáticamente.
# Este argumento (o cualquier otro que tenga la función que queramos usar)
# también se puede pasar a lapply, como hemos visto.
# Si no le pasamos ese argumento extra,
# la función "mean" fallará si encuentra NAs
mean.with.nas <- lapply(eq.clean, FUN = mean)
mean.with.nas
```

Esta operación, que se puede hacer usando `lapply` en una sola línea, es bastante más compleja usando un bucle `for`

```r
# definimos el vector donde almacenaremos los resultados
mean.with.for <- numeric(length(eq.clean))
# definimos un bucle que recorra todas las columnas (en este caso sólo dos)
for(i in 1:length(eq.clean)){
  mean.with.for[i] <- mean(eq.clean[,i],na.rm = TRUE)
}

# lapply devuelve una lista, aquí hemos definido un vector
mean.with.for
```