Estructuras de control y manejo de datos

Las estructuras de control, denominadas también sentencias de control son muy similares en los lenguajes de programación modernos. Básicamente lo que varía entre las estructuras de control de los diferentes lenguajes es su sintaxis; cada lenguaje tiene una sintaxis propia para expresar la estructura.

Las estructuras de control nos permite tomar decisiones y realizar un proceso repetidas veces. Se trata de estructuras muy importantes, ya que son las encargadas de controlar el flujo de un programa.

Esta instrucción sirve para realizar un conjunto de operaciones si se cumple cierta condición. A continuación se muestra la estructura básica de uso.

```
if (condicion){
  operación 1
  operación 2
  ...
}
```

Por ejemplo, la construcción “IF”, permite ejecutar o saltar un conjunto, bloque o secuencia de instrucciones dentro del código de un programa.

A continuación, notemos que la expresión señalada por el if se ejecuta y omite dependiendo del valor de la condición, TRUE o FALSE, respectivamente

```
# Ejemplo

A <- 17
if (A > 11)
  print("Mayor")

# Otra forma

if (A > 11) print("Mayor")
```

La construcción IF admite una sola expresión, **pero ésta puede ser la expresión compuesta que se construye mediante los paréntesis de llave { }** y las expresiones en su interior, separadas ya sea por el cambio de renglón o por ';'.

```
# Otra forma

if (A > 11){ # Intrucción compuesta
  print("Primera línea")
  print("Mayor")
}

# otra forma

if (A > 11){ # Intrucción compuesta
  print("Primera línea"); print("Mayor")
}
```

```
# Usando el valor que regresa el if

A <- 6
A <- if (A > 4) 17
A
```

INSTRUCCIÓN IF-ELSE

Esta instrucción sirve para realizar un conjunto de operaciones cuando NO se cumple cierta condición evaluada por un if. A continuación, se muestra la estructura básica de uso.

```
if (condición){
  operación 1
  operación 2
  ...
  operación final
}
else {
  operación 1
  operación 2
  ...
  operación final
}
```

Ejemplo: queremos saber si un numero es par o impar

```
# Ejemplo

x <- 5
#x <- 4

if (x %% 2 == 0){ # 1er. bloque
  print(" El número ingresado es par")
} else{ # 2o. bloque
  print("El número ingresado es impar")
}
```

Se pueden plantear múltiples condiciones simultáneamente: si se cumple la (Condición 1) se ejecuta (Bloque de sentencias 1). En caso contrario se comprueba la (Condición 2); si es cierta se ejecuta (Bloque de sentencias 2), y así sucesivamente hasta n condiciones. Si ninguna de ellas es cumple se ejecuta (Bloque de sentencias else).

Veamos más claramente esto con un ejemplo:

```
# La siguientes construcciones, redirigen la ejecución
# del código a distintos bloques o conjuntos de instrucciones deper
# de que se cumplan o no las condiciones establecidas:

if (10 > A) { # 1er. bloque
  print("RANGO MENOR")
} else if ( 10 <= A && A <= 20) { # 2o. bloque (expresión compuesta
  print("primer renglon"); print("RANGO MEDIO")
} else { # 3er. bloque
  print("RANGO MAYOR")
}
```

Instrucción ifelse

```
# Instrucción ifelse

# Se recomienda usar la instrucción ifelse cuando hay una sola instrucción para
# el caso if y para el caso else. A continuación se muestra la estructura básica
# de uso.

ifelse(condición, operación SI cumple, operación NO cumple)
```

Clarifiquemos esto con un ejemplo:

```
# Ejemplo

# Suponga que usted recibe un vector de números enteros, escriba un procedimiento
# que diga si cada elemento del vector es par o impar

x <- c(5,3,2,8,-4,1)
ifelse(x %% 2 == 0, 'Es par', 'Es impar')
```

CICLOS

R cuenta con varios tipos de ciclos o repeticiones: repeticiones por un número determinado de veces, repeticiones mientras se cumple una condición y repeticiones infinitas.

A continuación, discutiremos cada uno de estos casos:

Para la instrucción FOR, el número de veces que se repite la expresión o expresiones englobadas en la instrucción puede estar implícita en ella misma

```
# INSTRUCCIÓN FOR: repeticiones por un número determinado de veces

# A continuación, se muestra la estructura básica de uso.

for (i in secuencia){
  operación 1
  operación 2
  ...
  operació final
}
```

Por ejemplo:

```
# EJEMPLO 1

letras <- c("U","d","e","A")
for (i in 1:4){
  print(letras[i])
}
```

Notemos que el resultado anterior lo podemos obtener con cualquiera de las siguientes dos formas:

```
# FORMA 1

for (i in seq_along(letras)){
  print(letras[i])
}

# FORMA 2
for (letra in letras){
  print(letra)
}
```

En el primer caso se llamó a la función `seq_along()`, que genera una secuencia de enteros de acuerdo con el número de elementos que tenga el objeto que se le de como argumento. El segundo caso, tipifica la esencia de la construcción `for`, ya que se trata de ir tomando uno a uno los elementos del objeto consignado después de la partícula `in` del `for`.

Realicemos ahora un segundo ejemplo para tener más claridad en cuando a la instrucción for

```
# EJEMPLO 2: Escriba un procedimiento para crear 10 muestras de tamaño 100 de
# una distribución uniforme entre uno y tres. Para cada una de las muestra, se
# debe contar el número de elementos de la muestra que fueron mayores o iguales
# a 2.5.


nm <- 10 # número de muestras
n <- 100 # Tamaño de la muestra
contador <- numeric(nm) #vector para almacenar el conteo

for (i in 1:nm){
  x <- runif(n=n,min=1,max=3)
  contador[i] <- sum(x >= 2.5)
}

contador # Para obtener el conteo
```

INSTRUCCIÓN WHILE

```
# INSTRUCCIÓN WHILE: repeticiones mientras se cumple una condición

# A continuación, se muestra la estructura básica de uso.

while (condición){
  operación 1
  operación 2
  ...
  operació final
}

# EJEMPLO 1

letras <- c("U","d","e","A")
i <- 1
while (i <= 4){
  print(letras[i])
  i <- i+1
}
```

REPETICIONES INFINITAS

(Instrucción repeat)

repeticiones infinitas. La instrucción while es muy útil para repetir un procedimiento siempre que se cumple una condición pero como la instrucción no tiene condición de salida o interrupción, el resultado que la instrucción produciría en sí misma sería una repetición interminable;

AUNQUE en realidad no se quiere significar que las repeticiones sean infinitas o interminables, pues desde el interior del bloque de expresiones que se repiten, se obliga la interrupción del ciclo, por ejemplo, mediante la instrucción break

A continuación, se muestra la estructura básica de uso.

```
# A continuación, se muestra la estructura básica de uso.

repeat {
  operación 1
  operación 2
  ...
  operación final
  if (condición) break # La intrucción 'break' sirve para salir de un proceso iterativo
}
```

Ilustremos esto más fácilmente con un ejemplo1 a continuación

"Este ejemplo que hicimos anteriormente lo podemos realizar aquí nuevamente pero usando la instrucción REPEAT()"

```
# EJEMPLO 1

i <- 1
repeat{
  print(letras[i])
  i <- i+1
  if (i > 4)
    break
}
```

Ejemplo 2: imprimir los números del 3-7

```r
# EJEMPLO 2

x <- 3

repeat{
  print(x)
  x <- x+1
  if (x == 8){
    break
  }
}
```

INTERRUPCIONES DEL FLUJO NORMAL DE LOS CICLOS

(ANTES UNA BREVE INTRODUCCIÓN DE FUNCIONES)

Para crear o definir una función, se emplea la directiva “function”,

```r
# Definición de una función

f <- function(<Argumentos>){
  expresiones
  <valor>
}
```

IMAGEN PARA EXPLICAR:

El cuerpo de la función está constituido por una o más expresiones válidas del lenguaje. Al ejecutarse la función, el valor de la última expresión en la secuencia de ejecución, es el valor que la función entrega como resultado;

Argumentos y valor de resultado de una función

Ejemplos

```
# Forma 1
func1 <- function(x,yyy,z,t=5){
  w <- x+yyy+z
  w
}

func1(1,2,3)

# Forma 2
func2 <- function(x,yyy,z=5,t){
  w <- x+yyy+z
  w
}

func2(1,2)
func2(1,2,3)

# Forma 3
func3 <- function(x,yyy,z,t=5){
  w <- x+yyy+z
  return(w)
  3.1416 # Esta línea nunca se ejecuta
}

func3(1) # ERROR
func3(z=3,yyy=1,x=2)

# Forma 4
func4 <- function(x,yyy,z=5,t){
  w <- (x+yyy+z)*t
  return(w)
}
```

```r
func4(1,2,0)

# OBSERVACIÓN: los argumentos predefinidos siempre deben ir al final

func4 <- function(x,yyy,t,z=5){
  w <- (x+yyy+z)*t
  return(w)
}

func4(1,2,0)

# Revisión de los argumentos de una función

# args()
args(func1)

# formals()
(ar <- formals(func1))
#Si se quiere revisar el argumento t, se hace así:

ar$t

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Por ejemplo, si se quiere revisar los argumentos para la función lm(), que se
# usa para ajustes lineales, se puede hacer con:

args(lm)



################################################################################
```

Ahora, vamos a continuar con:

INTERRUPCIONES DEL FLUJO NORMAL DE LOS CICLOS

En el último ejemplo de (repeat) se ha utilizado la instrucción break para obligar la salida de un ciclo infinito que se realiza mediante la instrucción repeat sin embargo esta no es la única forma

EJEMPLO

```r
# RETURN

#Esta instrucción está asociada con las funciones y su propósito es
# interrumpir u obligar la salida de la función en la cuál se invoca, entregando,
# opcionalmente, como resultado de la función un valor si se da como argumento
# del return.


# Ejemplo: crear y ejecutar la función generadora de los números de fibonacci.

fibbonacci <- function(n){
  if (n %in% c(0,1))
    return(1)
  F0 <- 1; F1 <- 1; i <- 2
  repeat{
    s <- F0+F1 # Suma de los fib anteriores
    if (i == n) # Ya es el que se busca
      return(s)

    # Recorremos los últimos dos próximos números
    F0 <- F1
    F1 <- s
    i <- i+1
  }
}
```

NEXT (leer)

EJEMPLO

```r
# EJEMPLO:

for (i in 1:7){
  if (3 <= i && <=5)
    next
  print(i)
}
```

Hasta aquí, se han visto diferentes estructuras de control que, al igual que otros lenguajes de programación, permiten definir el flujo de ejecución de las instrucciones de algún programa. PERO la riqueza de este lenguaje está en el manejo de cada una de las distintas estructuras de información, implementadas a través de las clases de datos estructuradas, como vectores, factores, data frames, etc.,

FUNCIONES DE CLASIFICACIÓN, TRANSFORMACIÓN Y AGREGACIÓN DE DATOS

```
# Las funciones sapply() y lapply()

# Función sapply(): permite aplicar una operación o una función a cada uno de
# los elementos la lista o data frame, dado como argumento.
```

EJEMPLO

```
# EJEMPLO:

(datos <- data.frame(uno = runif(5,10.5,40.3), dos = runif(5), tres = runif(5, 155, 890)))

sapply(datos, mean)
class(sapply(datos, mean))
```

```
# Función lapply(): El resultado obtenido con la función lapply es más o menos
# similar, pero el resultado se entrega siempre en una lista:

lapply(datos, mean)
class(lapply(datos, mean))
```

El argumento opcional simplify, especificado aquí como TRUE: **obliga a que el resultado, si se puede, sea entregado como un vector, con un elemento correspondiente a cada una de las columna**s en este caso, de otra manera, el resultado es entregado como una lista. El resultado obtenido con la función lapply es más o menos similar, pero el resultado se entrega siempre en una lista:

```
class(lapply(datos, mean))
```

```
sapply(datos, mean, simplify = T )
```

OPERACIONES MARGINALES EN MATRICES Y LA FUNCIÓN apply()

las matrices resultan útiles para hacer diversidad de cálculos. Una de sus principales características es que tanto sus renglones como sus columnas

pueden ser tratados como elementos individuales. Resulta que hay operaciones que se efectúan para todas sus columnas o para todos sus renglones; a estas se les denominará operaciones marginales. R tiene algunas de estas operaciones implementadas directamente, entre ellas están las funciones: rowSums(), colSums(), rowMeans() y colMeans(). Para ejemplificar, se calcularán las sumas de las columnas de una matriz y los promedios de sus renglones.

EJEMPLO: Hacer una matriz arbitraria de 3 renglones y 5 columnas,

con rbind, que la construye por renglones:

```
# Ejemplo 1:

# Creemos una matriz arbitraria de 3 renglones y 5 columnas
# con rbind, que la contruye por renglones

(M <- rbind(5:9, runif(5,10,20),c(2,-2,1,6,-8)))

colSums(M)

rowMeans(M)
```

Estas operaciones, que se tienen implementadas directamente, evitan mucha programación que se tendría que hacer con ciclos y operaciones individuales con los elementos de la matriz.

USANDO LA FUNCIÓN apply()

Para la función apply(), no es necesario que la función a aplicar esté predefinida, ni que el resultado de esa función sea un único número. Para clarificar esto, se propone el siguiente ejemplo:

Veamos que usando la función apply(), podemos obtener las operaciones marginales dichas anteriormente

```
# USANDO LA FUNCIÓN apply()

# Margen de los renglones está dado por '1'
# Margen de las columnas está dado por '2'

apply(M,1,mean) # Para los renglones

apply(M,2,sum) # Para las columnas
```

EJEMPLO 2

```r
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Ejemplo 2

f <- function(v){
  return(v * (1:length(v)))
}

# probemos la función:
v <- c(2, 4, 6, 8)
f(v)

# Ahora la aplicaremos a todas las columnas de la matriz
apply(M, 2, f)
```

CLASIFICACIONES Y USO DE LA FUNCIÓN split()

4.3.4. Clasificaciones y uso de la función `split()`

Generalmente, aunque los datos con los que se quiere trabajar están dados en tablas, no están organizados de la manera que se pretende trabajar sobre ellos. La función `split()`, permite clasificar los datos, típicamente dados como un vector, o como un *data frame*. Para explicar el uso de esta función se recurrirá a un ejemplo. Supóngase que los datos, que se muestran en la Fig. 4.1, se tienen

"DESPUÉS DE INSERTAR LOS DATOS, MOSTRAR LO QUE DESEO HACER"

Supongamos ahora que se desea tener la misma información pero clasificada por años. Con este propósito se puede emplear la función split(), que recibe básicamente dos argumentos principales: el objeto a ser clasificado típicamente, un data frame o un vector-, y una serie de datos que servirán para la clasificación -típicamente, un factor o un vector numérico o de cadenas de caracteres

EJEMPLO:

```r
# La función split(), permite clasificar los datos, típicamente dados como
# un vector, o como un data frame.


# Ejemplo

datos <- Precipitaciones

#View(datos)
#head(datos)

(u <- split(datos,datos$year))

(u <- split(datos[,3:5],datos$Season))
(u <- split(datos[,3:5],datos$year)) # Note: datos[, 3:5] == datos[3:5]
(u <- split(datos[3:5],datos$year))

# Notemos que el orden sí importa
(u <- split(datos$year,datos[,3:5]))

(u <- split(datos[,3:5],datos$year)) # solo para actualizar u
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

El resultado de la función, en el caso del ejemplo, es una lista de tablas, cada una correspondiente a cada uno de los años registrados en la columna datos$año

X

PARÉNTESIS

Supóngase ahora que se desea tener la misma información pero clasificada por años. Con este propósito se puede emplear la función `split()`, que recibe básicamente dos argumentos principales: el objeto a ser clasificado -típicamente, un *data frame* o un vector-, y una serie de datos que servirán para la clasificación -típicamente, un factor o un vector numérico o de cadenas de caracteres-. Este último argumento debe tener la misma longitud que una columna del *data frame*, dado como primer argumento, o que el vector dado como argumento. según el caso, y no es necesario que este argumento sea parte del objeto dado como primer argumento. Si, como primer argumento de la función

X

Solo para mostrar la potencia del lenguaje con estas operaciones, supóngase que se quiere aplicar una operación a cada una de las tablas de la lista resultante, el promedio por columnas o la suma por renglones, por ejemplo. Dado que se tata

de una lista, podemos utilizar cualquiera de las funciones lapply() o sapply(), como se muestra a continuación.

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Supongamos que se quiere aplicar una operación a cada una de las tablas de la
# lista resultante,el promedio por columnas o la suma por renglones, por ejemplo:


lapply(u,colMeans)
lapply(u,rowSums)

# Hacer caso omiso de los valores NA

lapply(u,colMeans,na.rm=TRUE)
class(lapply(u,colMeans,na.rm=T))# T es TRUE

sapply(u,colMeans,na.rm=TRUE)
class(lapply(u,colMeans,na.rm=T))# T es TRUE

sapply(u, rowSums, na.rm = T) # T es TRUE
class(sapply(u, colMeans, na.rm = T))
```

Una nota final aquí es que en este último ejemplo se ha utilizado la función `sapply()`, en vez de la función `lapply()`. Como se puede ver, en este caso el resultado ha sido simplificado lo más posible, entregando una matriz en vez de una lista con cada uno de los resultados. Obsérvese también que para el mes de marzo y el año 1978, la tabla ha arrojado un resultado `NaN` (*not a number*), aunque se ha instruido hacer caso omiso de los valores `NA` en los cálculos

EXTRAAAAAAAAA

CLASIFICACIÓN Y OPERACIÓN: las funciones by(), aggregate() Y tapply()

En los últimos ejemplos se muestra una secuencia de acciones separadas, es decir, primero una clasificación de los datos, para después aplicar una operación sobre cada una de las partes encontradas mediante la clasificación.

De este modo, se aplicó primeramente la función split(), para clasificar la información, y después, sobre el resultado de esta clasificación, se aplicó cualquiera de las funciones lapply() o

sapply(), para distribuir una operación, colMeans(), sobre cada una de las partes de la clasificación resultante.

- BY()

Mediante la función by(), el lenguaje provee una manera de hacer estas dos acciones simultáneamente.

LEER:

La función by(), básicamente tiene tres argumentos principales: el objeto al cual se aplica -típicamente un data frame-, el factor o vector usado para clasificar y la función que se aplica a cada uno de los elementos resultantes de la clasificación, y puede además tener argumentos adicionales, tales como na.rm,

EJEMPLO:

```
# CLASIFICACIÓN Y OPERACIÓN: las funciones by(), aggregate() Y tapply()

# Función by()

# La función by(), básicamente tiene tres argumentos principales: el objeto al
# cual se aplica -típicamente un data frame-, el factor o vector usado para
# clasificar y la función que se aplica a cada uno de los elementos resultantes
# de la clasificación, y puede además tener argumentos adicionales, tales
# como na.rm,


# Ejemplo 1
(u <- by(datos[, 3:5], datos$year, colMeans, na.rm = T))
class(u)

# ¿Cómo acceder a la información del resultado anterior?

# Primera forma:
u$`1981`

# Segunda forma
u[["1981"]]
u[['1981']]
```

```
# Tercera forma
u["1981"]
```

Función aggregate()

Nótese que, a diferencia de la función `by()`, para la función `aggregate()` se ha pasado como función de operación sobre el primer argumento la función `mean()` en vez de la función `rowMeans()`. Esto se debe a que `by()` actúa como si operara con la salida de una clasificación hecha con `split()`, que es una lista con cada resultado de la clasificación, mientras que `aggregate()` opera directamente sobre las columnas del *data frame*. También, como se puede ver, el resultado de `aggregate()`, convenientemente es una tabla de datos vertida en un *data frame*, que es fácilmente manipulable como se ha mostrado antes.