**Estructuras de Control**

**Introducción**

R como casi cualquier otro lenguaje de programación proporciona estas estructuras que no son más que construcciones sintácticas que determinaran el orden y la lógica de algunas operaciones dentro de nuestro flujo de código.

Es decir, van a dirigir el flujo de las operaciones y van a controlar su ejecución hacia una dirección o hacia otra, dependiendo de las condicionantes que les propongamos.

Entre ellas la más común es la estructura de control de IF, la cual veremos a continuación, estamos de acuerdo en que ya algunos de ustedes han visto y saben cómo se manejan, si han tenido algún curso o diplomado en programación, pero vamos a hacer de cuenta que no las conocen en absoluto y vamos a irnos por lo básico, sin embargo tampoco nos vamos a detener en mucho detalle en ellas, pues la verdad casi no se utilizan en el análisis de datos, o por lo menos yo las utilizo muy poco, sin embargo es importante que las conozcamos y podamos aprovechar sus beneficios en caso de que nos encontremos con algún problema que necesite aplicar este tipo de estructuras, las cuales casi siempre entran al ruedo cuando se trata de consumir datos de fuentes externas que tengan alguna latencia u otro tipo de control.

**IF-ELSE**

Estos funcionaran igual que en cualquier otro lenguaje de programación, la única salvedad es que en R estas funciones pueden ser capitalizadas, es decir las construcciones regresarán un valor que puede ser asignado a un objeto o ser utilizado de otras maneras creativas.

Vamos a mostrar los siguientes ejemplos en R:

aa **<-** 15

**if(**aa **>** 14**)** print**(**"SI MAYOR"**)**

Aquí vemos un código bastante simple, en primer lugar, estamos asignando al objeto “aa” un valor numérico 15.

Posteriormente la estructura de control if() encerramos entre paréntesis la condición si “aa” es mayor que 14, en este caso devuelve un valor lógico de falso o verdadero, como en este caso “aa” que se interpreta dentro de los paréntesis como 15 la función hace una comparación de esta manera:

* (15 es mayor que 14) = VERDADERO

Seguidamente utilizamos la función de print que se ejecutara si el valor es igual a verdadero imprimiendo lo contenido en sus paréntesis en este caso “SI MAYOR”.

De esta manera con las estructuras de control, podemos coaccionar para la toma de decisiones y tomar una ruta determinada dependiendo del resultado de esta.

**Instrucción Compuesta**

Cuando deseamos agregar más de un resultado a determinada ejecución podemos usar una instrucción compuesta la cual encerraremos entre paréntesis {} de esta forma:

**if(**aa **>** 14**){** #Instruccion compuesta

print**(**"Primer Reglon"**)**

print**(**"Si Mayor"**)**

**}**

En el ejemplo anterior usamos parte del código anterior sin embargo a la respuesta que es verdadera le asignamos una respuesta compuesta de dos notificaciones como lo podemos comprobar ejecutándolo en nuestros computadores.

En el código siguiente vamos a capitalizar la respuesta de esta forma:

y **<-** 10

z **<-** **if(**aa **>** 14**)** 50

z

Arriba asignamos al objeto z el valor de la estructura de control antes asignada.

Ahora vamos a proponer algo más complejo, ya que utilizaremos varios operadores y expresiones de una forma compuesta.

**if(**10 **>** aa**)** **{** #1mer Bloque

print**(**"Rango Menor"**)**

**}** **else** **if** **(**10 **<=** aa **&&** aa **<=** 20**)** **{** #Segundo Bloque

print**(**"Primer Renglon"**)**; print**(**"Rango Medio"**)**

**}** **else** **{** #Tercer Bloque

print**(**"Rango Mayor"**)**

**}**

Recordemos que el valor asignado a “aa” es igual a 15, entramos a la lógica y nos indica lo siguiente:

Si 10 es mayor que 15 imprima “Rango Menor”, aquí no cumple una condición verdadera, sino “FALSO”, por lo cual recorre a la siguiente estructura y verifica si 10 es menor o igual que 15 y 15 es menor o igual a 20 en donde retorna “VERDADERO”, lo cual activa la impresión de ese bloque, y en caso de que no se cumpla el primero o el segundo ejecutara la tercera condición. Los algoritmos de clasificación funcionan con este principio, pero con una lógica y matemática avanzada, para obtener algún resultado, a este tipo de código siempre es conveniente dejarle una salida en caso de que cualquiera de las condiciones sea falso.

**Los Ciclos**

En R encontraremos varios tipos de funciones de ciclos o repeticiones, pudiendo ser que ser repitan los mismos un número determinado de veces ya prestablecido, cuando se cumpla o se deje de cumplir alguna condición o infinitas veces, en este último hay que prestar atención y cuidado ya que fácilmente pueden ocasionar un ciclo infinito en su computador y con ello bloquear memoria y procesadores.

Comenzaremos con repeticiones con un número determinado de veces; Con repeticiones explicitas (es decir le asignaremos el valor de las repeticiones).

letras **<-** c**(**"c", "L", "i", "M", "T", "A"**)**

**for** **(**i **in** 1**:**6**)** **{**

print**(**letras**[**i**])**

**}**

**Pregunta rápida:**

¿Qué tipo es el objeto letras?

**Su respuesta aquí: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Continuamos….**

En el código anterior, primero le asignamos al objeto letras los valores "c", "L", "i", "M", "T", "A" con la función de contenedor vista anteriormente.

Posteriormente usamos la función for() para ingresar un ciclo de ejecución, dentro de la función cargamos los argumentos i = el cual será un numero aleatorio que se sumará mas uno en lo que recorre el segundo argumento que es 1:6, es decir desde 1 al 6, recordemos que los dos puntos (:) nos sirven para mostrar un rango especifico de números continuos.

La función print llama al objeto letras (de allí la pregunta que tipo de objeto es), y usamos los [] para recorrer la posición de cada uno de los valores contenidos en ese objeto.

Nuestro resultado cuando corramos el código será el siguiente:

[1] "c"

[1] "L"

[1] "i"

[1] "M"

[1] "T"

[1] "A"

Podemos usar también código en donde el número de veces que se repita cierto ciclo quede implícito; En este caso usaremos la función seq\_along() la cual nos va a generar una secuencia de enteros acorde al número de elementos que contenga el objeto.

El código quedaría así:

letras **<-** c**(**"c", "L", "i", "M", "T", "A"**)**

**for** **(**i **in** seq\_along**(**letras**))** **{**

print**(**letras**[**i**])**

**}**

En el código anterior vemos que no especificamos el número de veces, ya que lo tomamos directamente del objeto. Esta técnica la podemos usar cuando asignamos un objeto que se puede variar en su longitud, por ejemplo: Si usamos un objeto que se llame nits de una base de contribuyentes y lo estemos migrando de forma secuencial mes a mes, el objeto se puede llamar igual, pero el contenido puede variar en número dependiendo de la temporalidad de este.

Al usar la función seq\_along, esta leerá la longitud y asignará el valor de forma automática.

Podemos crear un código más simplificado y hacerlo aún más implícito, aprovechando la propia esencia de la función for() creando una construcción que tratara de ir tomando cada uno de los elementos del objeto consignado.

letras **<-** c**(**"c", "L", "i", "M", "T", "A"**)**

**for** **(**letra **in** letras**)** **{**

print**(**letra**)**

**}**

**Repeticiones con condición**

Podemos habilitar esta operación de ciclo con la función while(), sin embargo con ella debemos de tener mucho cuidado ya que tenemos si o si, que poner un objeto que indique un índice de entrada y salida ya que si no lo hacemos adecuadamente ocasionamos un ciclo infinito y sin salida.

Para su tranquilidad estos temas los estamos viendo como parte de cultura de programación, R posee estructuras y funciones avanzadas que nos ayudan de una manera más fácil y limpia cuando se necesitan este tipo de trabajos, estos los veremos en capítulos posteriores, aunque si es necesario que veamos sus bases con antelación.

En el siguiente código vemos cómo funciona un ciclo con repeticiones por condición:

letras **<-** c**(**"c", "L", "i", "M", "T", "A"**)**

i **<-** 1

**while(**i **<=** 6**)** **{**

print**(**letras**[**i**])**

i **<-** i **+** 1

**}**

En esta parte de código nuevamente agregamos el objeto letras como en los ejemplos anteriores.

Le asignamos al objeto “i” un valor inicial, en este caso 1, dentro de la función while() estamos especificando que el ciclo continue mientras i sea menor o igual a 6 con el operador lógico “menor o igual que” **“<=”** , esto quiere decir que mientras no se de este caso el algoritmo se repetirá hasta que la condición se cumpla.

En el ciclo imprimirá cada vez que pase el valor que se encuentra en la posición del vector en especifico en este caso el vector se llama letras y se accede a cada una de sus posiciones por el mismo objeto i, siendo por ejemplo lo siguiente.

Letras [1] = “c”, letras[2] = “L” y así sucesivamente hasta que el ciclo se interrumpa.

En el caso que ya no existiera mas contenido en el vector, R le asignara NA a los siguientes valores. Para comprenderlo mejor podemos modificar el valor de “i” cambiándolo de 6 a 8 o 10 o el numero de repeticiones que usted guste.

**Repeticiones Infinitas**

Existe la función repeat() que realiza ciclos infinitos, el lenguaje posee facilidades para poder interrumpir el ciclo desde su interior cuando se cumpla una condición, lo cual veremos en el siguiente código:

letras **<-** c**(**"c", "L", "i", "M", "T", "A"**)**

**repeat** **{**

print**(**letras**[**i**])**

i **<-** i **+** 1

**if(**i **>** 6**)**

**break**

**}**

En este código a diferencia de los otros, declaramos el objeto contador dentro de la función y posteriormente con la función if(), marcamos un alto o parada para la misma disparando la función break.

Los ciclos pueden ser interrumpidos con 3 instrucciones diferentes, ya sea break, next o return.

En el siguiente ejemplo vamos a usar un generador de numero aleatorios para tratar de explicar con claridad estos eventos.

set.seed**(**140**)** #Cualquier numero

aprox **<-** 0.003 #Valor de la salida del ciclo

Y\_ini **<-** 2.7 #Valor inicial de Y supuesto

**for(**iter **in** 1**:**1000**)** **{** #Aseguro no mas de 1000 interaciones

#Procedimiento para calcular la siguiente Y, que en este caso simularemos mediante un generador aleatorio

Y **<-** Y\_ini **+** 0.008**\***rnorm**(**1**)**

**if(**abs**(**Y **-** Y\_ini**)** **<=** aprox**)**

**break** #salir del ciclo

Y\_ini **<-** Y

**}**

paste**(**"Y\_ini", Y\_ini, "Y:", Y, "Num.Iter", iter**)** #Unir el Resultado

Pasaremos a explicar el código:

La función set.seed pone una semilla que tomará como un punto aleatorio para que el ejemplo se comporte igual en todos los equipos, es decir que fuerza que un resultado sea el mismo independientemente del equipo en el que se esté corriendo, aunque sea un trabajo aleatorio.

El objeto “aprox” únicamente nos sirve para establecer el valor de la salida del ciclo, en este caso 0.003

Y\_ini es la ponderación que le damos al objeto como salida 2.7

En la secuencia del for() establecemos un objeto iter que será recorrido no superior de mil veces.

Al objeto “Y” le aplicamos una fórmula matemática en donde tenemos el valor inicial de Y\_ini que es 2.7 más 0.008 \* rnorm(1), rnorm es una función que nos ayuda a mantener una distribución normal, existe el dnorm, pnorm, qnorm y rnorm, estos los veremos cuando entremos de lleno al área de estadística, pero la función rnorm nos va a ayudar a mantener el ejemplo dentro de una distribución normal con su media y si desviación estándar adecuadamente.

La función abs() también la vemos y nos ayuda a calcular el valor absoluto de lo que estamos obteniendo de los objetos Y restándole el Y\_ini y cuando este sea menor o igual al objeto “aprox” para el ciclo, pasandole el valor a Y de lo que en ese momento tenga de Y\_ini.

Posteriormente usamos la función de paste() que nos ayuda a concatenar los resultados de los objetos que queremos mostrar.

Este solo es un ejemplo de como se puede codificar un ciclo, donde podemos ver que el objetivo fue alcanzado en 8 iteraciones.

En el siguiente ejemplo utilizaremos next para interrumpir flujo normal de la ejecución de una manera diferente, en vez de salir del ciclo, solamente impedirá la ejecución de las instrucciones siguientes, retornando al principio del ciclo nuevamente.

Aquí el código:

**for(**i **in** 1**:**7**)** **{**

**if(**3 **<=** i **&&** i **<=** 5**)**

**next**

print**(**i**)**

**}**

**FUNCIONES**

Una de las particularidades de R, es que va a tratar las funciones como cualquier otro objeto. En mi caso suelo utilizar mucho las funciones cuando estoy haciendo aplicaciones web con Shiny Server, ya que me ayuda a mantener el flujo y control de los datos cuando creo las geometrías reactivas al usuario, sin embargo, nos pueden ayudar en muchos otros procesos como domado de datos, raspado web o incluso al momento de realizar algunas pruebas estadísticas como las famosas pruebas de Montecarlo.

Al ser estas mismas tratadas como cualquier otro objeto, tenemos la posibilidad de manipularlas y se pueden incluso pasar como argumentos para que regresen un valor final, se pueden definir en el interior de otra función y solo nuestra creatividad nos dirá que tanto podemos explotar este tema.

Para cargarla al objeto sencillamente utilizamos la directiva function().

La función solo cuenta con dos partes, la primera es donde definiremos los argumentos formales que va a utilizar, la segunda es en si el cuerpo en donde haremos las operaciones o lo que necesitemos que la función retorne.

En este caso para estar mas claro diremos que el cuerpo de la función es constituido por una o mas expresiones que son validas en nuestro lenguaje R.

Seguidamente tenemos dos momentos importantes en la vida de la función, la definición la cual básicamente ocurrirá una única vez y la ejecución que puede ocurrir N cantidad de veces.

Cuando creemos un objeto para poderlo diferenciar, podemos utilizar la letra “f” guion bajo “\_” y después el nombre para poder identificar la función en nuestro código cuando estemos desarrollando un proyecto más adelante.

Aquí su esquema general:

f <- function("<Argumentos>") { #Argumentos Formales

Expresiones #Cuerpo de la funcion expresiones

<Valor> #Valor que regresa la funcion

} #Fin de la funcion

Podremos decir que los Argumentos formales es donde nuestra función se comunicará con el ambiente exterior.

**Ejemplos:**

**Definición de función versión 1:**

MiFunc.v1 **<-** **function(**x,yyy,z**=**5, t**)** **{**

w **<-** x **+** yyy **+** z

w

**}**

MiFunc.v1**(**5,10,6**)**

Explicación: En este caso nombramos el objeto MiFunc.v1 al cual le agregamos los argumentos formales “x, yyy, z =5, t” ; Abrimos corchetes y en el cuerpo de la función asignamos otro objeto de nombre “w” en el cual cargaremos las operaciones matemáticas x+yyy+z, posteriormente copiamos el objeto “w” nuevamente abajo para que se ejecute su contenido (es lo mismo ponerlo en un print(w), en este caso cuando solo ponemos el objeto, R reconoce el print() que está embebido; Finalizamos nuestra función cerrando el corchetes.

El ejecutar utilizamos el objeto y le insertamos los argumentos: MiFunc.v1**(**5,10,6**)**

Primero vamos a aclarar que los nombres de los argumentos pueden ser cualquier cosa, en general yo le pongo al argumento un “df” si lo que va a operar es un dataframe, “fec”, si será alguna operación que lleve fecha y latencia, “int” si se trata de un numero entero, “fac” si es factor, aunque no es una regla per se, sino únicamente valores que me pueden servir a mi para identificar el tipo de argumento que estoy enviando a mi función.

Otra observación que podemos ver es que a un argumento (en este caso al z), le asigne un valor definido, la función en caso de que este argumento este vacío o no exista, automáticamente le asignara el valor que traiga por defecto.

Finalmente puse un argumento que no uso, en este caso el “t”, si en el cuerpo de la función, no existe ninguna llamada al argumento esté sencillamente no se usa.

El resultado de la función es igual a 21.

**Definición de Función versión 2.**

MiFunc.V2 **<-** **function(**x, yyy, z**=** 5, t**)** **{**

w **<-** x **+** yyy **+** z

return**(**w**)**

3.1416 #Aqui no se va a ejecutar nada

**}**

MiFunc.V2**(**5,10**)**

Tal como en el ejemplo anterior asignamos los mismos argumentos, sin embargo tenemos un par de observaciones extra.

La primera es que el número que pusimos después del return() de la función no será mostrado o ejecutado, ya que el mismo hace un corte y muestra el resultado de la operación de “w”.

La segunda es que, al momento de ejecutar la función, le estamos enviando únicamente dos argumentos, sin embargo, como ya tenemos asignado un valor por defecto para “z=5”, toma este valor y los opera con los dos argumentos anteriores.

Tal como los vectores, vemos que los argumentos tienen un orden explicito, primero e primero, segundo el segundo y así sucesivamente.

El resultado de esta función es igual a 20.

**Definición de Función versión 3.**

MiFunc.V3 **<-** **function(**x, yyy, z**=**5, t**)** **{**

x**+**yyy**+**z

**}**

MiFunc.V3**(**5,10**)**

Como vemos en este ejemplo, tenemos el mismo argumento sin embargo no estamos creando un nuevo objeto, sino únicamente hacemos la operación tal cual y luego la llamamos seguidamente.

Funciona igual que las demás, el código es mas corto, pero sin embargo en operaciones complejas este no puede ser un buen estándar para trabajar ya que a posterior ni nosotros mismos vamos a saber que hicimos, por lo que no es recomendable, aunque es necesario ponerla también como un ejemplo de simplicidad y llamados embebidos.

**Ejemplo para estudio:**

Aquí vamos a poner el código una función que ejecute los números de fibbonacci, el cual argumenta lo siguiente:

Tenidos o dados los dos primeros números F0 y F1, este calculará cada uno de los siguientes como la suma de los dos anteriores.

Quedando así:

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**)** #Buscar el no.

return**(**s**)**

F0 **<-** F1

F1 **<-** s

i **<-** i**+**1 #Incrementamos el indice

**}**

**}**

Podemos ejecutarla por ejemplo así:

fibbonacci**(**8**)**

Ahora haremos práctica, si tiene alguna duda, lea el documento nuevamente, la serie de practicas está realizada para desarrollar el pensamiento lógico, las soluciones son sencillas y muchas veces más fáciles de lo que nosotros podemos imaginarnos. Sin embargo, se vale hacer grupos, consultas y sobre todo tratar de analizar detenidamente cada una de las preguntas que son planteadas, utilice su imaginación, realice pruebas y valídelas, recuerde en ultima instancia yo voy a dar las soluciones a cada una para que las compare con sus respuestas, hay mas de un camino para poderlas resolver y lo importante es llegar al resultado.

**Practica**

**Instrucciones:** Carguemos nuestra librería “tidiverse”, lea detenidamente el documento y busquemos la mejor ruta para nuestra respuesta.

**Practica No. 1**

Cree una función que calcule el factorial de un número, para este ejercicio vamos a usar la función reduce(). Investigue en R como utilizar está función y que hace para luego hacer la función.

En matemáticas, la factorial de un número es igual al producto de todos los números enteros positivos desde 1 hasta dicho número.

**Practica No. 2**

Cree una función llamada cum\_add que haga una suma acumulada, por ejemplo, si x <- 1: 3 cum\_add (x) resultará en 1 3 6.

Sin usar la función cumsum().

**Practica No. 3**

Cree una función que tome un data frame y devuelva la media, el mínimo y el máximo de todas las columnas numéricas. Su función debe tomar un dataframe como entrada, por ejemplo, su\_funcion(iris)

Trate de evitar los loops, es posible hacerlo en una sola línea, aunque no imprescindible.

**Practica No. 4**

Cree una función centerColumnAroundMean que tome un dataframe numérico y manipule el mismo de tal manera que todos los valores de la columna estén centrados con el valor promedio de la columna.

Podemos usar la función sweep() que nos ayudará con esto.

Por ejemplo, si mi dataframe es como este

**df <- data.frame (x = 1: 5, y = 6: 10, c = 11: 15)**

Luego de centerColumnAroundMean (df) resultará en:

**# x y c**

**#1 -2 -2 -2**

**#2 -1 -1 -1**

**#3 0 0 0**

**#4 1 1 1**

**#5 2 2 2**

**Practica No. 5**

Tengo una lista de películas, que tienen dos franquicias como los elementos, Starwars y LOTR. Puede crear la lista de películas mediante:

my\_movielist **<-** list**(**star\_wars**=** list**(**"A NEW HOPE","The Last Jedi","The Force Awakens"**)**,LOTR**=**list**(**"THE FELLOWSHIP OF THE RING","THE Two Towers","The RETURN of the KING","Hobbit" **=** list**(**"An unexpected Journey","The Battle of the FIVE ARMY","The Desolation of Smaug"**)))**

**Ahora el problema:** Algunos de los textos están en mayúsculas y otros en minúsculas, esté es un pequeño ejercicio creado como entrada para cuando nos toque ver el domado de datos, los anteriores están sin ningún orden en particular.

**Su solución:** Me gustaría que la lista tuviera un formato como:

**“The Last Jedi”**

Escriba una función que haga esto.

Tenga en cuenta que la lista es una lista anidada (nested list).

**Practica No. 6**

Cargue el conjunto de datos diamonds del paquete ggplot 2. Para ello primero carguemos la librería tidyverse en la cual ya está incluida la de ggplot.

data**(**"diamonds"**)**

**El Problema:** Quiero comprar un diamante de cada color, pero no quiero pagar el precio más alto, creo que el segundo precio más alto es lo suficientemente bueno para mí. Escribamos una función para encontrar el segundo precio más alto para cada color del conjunto de datos.

**Practica No. 7**

Utilice el conjunto de datos de diamons ya cargados del ejercicio anterior. Quiero saber el precio promedio de cada combinación de corte y color. Su salida debería ser similar a la siguiente. No use la función table().

## D E F G H I J

## Fair 4291.061 3682.312 3827.003 4239.255 5135.683 4685.446 4975.655

## Good 3405.382 3423.644 3495.750 4123.482 4276.255 5078.533 4574.173

## Very Good 3470.467 3214.652 3778.820 3872.754 4535.390 5255.880 5103.513

## Premium 3631.293 3538.914 4324.890 4500.742 5216.707 5946.181 6294.592

## Ideal 2629.095 2597.550 3374.939 3720.706 3889.335 4451.970 4918.186

**Practica No. 8**

Cargue el conjunto de datos del iris, quiero obtener la tercera fila para cada grupo de especies.

data**(**"iris"**)**

Escriba una función corta para que realice eso.

**Practica No. 9**

Cree un nuevo entorno con el comando new.env () y cree 3 variables vectoriales bajo ese entorno como a = 1: 10; b = 100: 500; c = 1000: 1500 sin saber o llamar manualmente la media para todas las variables vectoriales, imprima el promedio de todas las variables del nuevo entorno.

Investiguemos acerca de las siguientes funciones: apply()