

Fastbook 04

Programación en R

Funciones



Edix Educación

04. Funciones

En este cuarto fastbook vamos a centrarnos en las funciones. Hasta el momento, solo hemos usado aquellas que vienen incluidas en R base o en paquetes externos que hemos tenido que instalar e importar en nuestras sesiones.

Sin embargo, ha llegado el momento de aprender a construir nuestras propias funciones R, que, sin duda, son un elemento indispensable cuando nos enfrentamos al desarrollo de un proyecto de código. De forma posterior, pondremos el foco en aquellas situaciones en las que necesitamos aplicar una función sobre los diferentes elementos de una variable estructural. Para ello, abordaremos la vectorización y la familia de funciones `apply`.

Autor: Juan Jiménez García

Funciones

Vectorización

Apply family

Conclusiones

Funciones

X Edix Educación

¿Qué son las funciones?

Las funciones son fragmentos de código a los que les hemos asignado un nombre que les representa y que están compuestos por una serie de operaciones que realizan una o más tareas.

Si resulta necesario, pueden recibir **argumentos de entrada** y devolver los correspondientes resultados.

Ya en el primer fastbook comentamos los tres posibles orígenes que puede tener una función:

- Venir incluidas con la instalación de R. Lo que denominamos R base.
- Estar contenidas en paquetes externos que tenemos que descargar e instalar.
- Ser creadas, definidas y programadas por nosotros mismos.

Hasta el momento ya hemos trabajado sobre los dos primeros casos, por lo que ahora nos toca centrarnos en el tercero: aprender a construir nuestras propias funciones.

Antes de nada, vamos a comentar las **ventajas** que aporta el hecho de dividir nuestro código en funciones.

Tamaño

Evita la repetición innecesaria de código. Si vamos a realizar una acción varias veces, podemos crear una función y llamarla cuando resulte necesario en lugar de volver a escribirla.

Organización

Permite construir el código de una forma más estructurada y ordenada mejorando su comprensión.

Colaboración

Facilita el trabajo en equipo. Al partir el código en bloques o módulos, resulta más sencillo dividir tareas y programar en paralelo.

Mantenimiento

Cuando tenemos que solucionar errores o implementar mejoras, solo tenemos que realizar el cambio en un lugar del código. Sin embargo, si la misma tarea se encuentra repetida en varias partes del script, estas modificaciones suelen volverse complejas.

Testing

Permite testar el buen funcionamiento de cada función por separado para comprobar que el código se encuentra en buen estado y realiza las tareas pertinentes.

¡Pasemos a la acción!

Para construir una función en R debemos usar la sentencia ***function()*** y definir sus **4 partes principales**:

1

Nombre

Las funciones también son objetos que se almacenan en nuestro entorno y, por tanto, les daremos un nombre. En el ejemplo de la imagen se llamará *suma*. Debemos evitar la utilización de nombres que ya estén siendo usados por otras funciones de R, como por ejemplo, *mean*.

```
#La funcion suma recibe dos numeros, realiza su suma  
#y devuelve el resultado.  
#El valor por defecto del parametro b es 10.  
suma <- function(a,b = 10){  
  resultado <- a + b  
  return(resultado)  
}
```

2

Parámetros de entrada

Son variables que reciben la función para poder trabajar con ellas. Vienen separadas por una coma dentro de la sentencia `function()`. Podemos definirlas únicamente con su nombre (como en el caso de la `a`) y también asignarles un valor por defecto (como en el caso de la `b`). Esto significa que, si llamamos a la función sin entregar la variable `b`, se asumirá que es igual a 10, siendo siempre obligatorio introducir un valor para la variable `a`.

Por supuesto, pueden existir funciones que no reciben ningún parámetro de entrada.

```
#La funcion suma recibe dos numeros, realiza su suma  
#y devuelve el resultado.  
#El valor por defecto del parametro b es 10.  
suma <- function(a,b = 10){  
  resultado <- a + b  
  return(resultado)  
}
```

3

Cuerpo

Es el lugar en el que se desarrollan las tareas para las cuales ha sido diseñada la función. En el caso del ejemplo, guardamos la suma de los dos números en la variable `resultado`.

```
#La funcion suma recibe dos numeros, realiza su suma  
#y devuelve el resultado.  
#El valor por defecto del parametro b es 10.  
suma <- function(a,b = 10){  
  resultado <- a + b  
  return(resultado)  
}
```

4

Resultado

Aunque resulta obligatorio, normalmente las funciones devuelven una variable como resultado de las líneas de código que se han ido ejecutando. Podemos hacerlo de dos formas:

- Escribiendo en la última línea de la función el nombre de la variable que queremos entregar.
- Incluyendo la variable a devolver dentro de la sentencia `return()`, tal y como se ve en el ejemplo. Es la opción más recomendada, ya que nos permite dar el resultado y acabar con la ejecución aunque no estemos en la última línea.

```
#La funcion suma recibe dos numeros, realiza su suma  
#y devuelve el resultado.  
#El valor por defecto del parametro b es 10.  
suma <- function(a,b = 10){  
  resultado <- a + b  
  return(resultado)  
}
```

Para que podamos usar la función que estamos definiendo, debemos ejecutar su código en la consola. En el ejemplo anterior estaríamos asignando la definición de la función al objeto llamado `suma`.

Antes de continuar con otros ejemplos, debemos hablar sobre un tema que presenta gran importancia y conviene comprender. Me refiero a la diferencia que existe entre las variables de tipo local y global.

- **Variables globales:** se construyen en la ejecución principal de un script y, por tanto, forman parte de nuestro entorno global y pueden ser accedidas desde cualquier parte de nuestro código.
- **Variables locales:** se construyen dentro de las funciones. Su existencia se ciñe únicamente al contexto interno de la función, por lo que no forman parte de nuestro entorno global y no pueden ser accedidas desde el resto del código.

La siguiente imagen nos va a ayudar a entender estos dos conceptos de una forma más clara:

The screenshot shows the RStudio interface with the following details:

- Script Editor:** Displays the R script code. The code defines a function `suma` that adds two numbers. It also creates a variable `numero` and assigns it the value 20. Then, it calls the `suma` function with `numero` as the argument and stores the result in `resultado_obtenido`.
- Global Environment:** Shows the current state of the global environment.
 - Values:** `numero` has a value of 20, and `resultado_obtenido` has a value of 30.
 - Functions:** The `suma` function is listed.
- Console:** Shows the command history and the execution of the script. The output matches the values shown in the Global Environment.

- La creación de la función *suma* se realiza en la ejecución global, por eso aparece a la derecha como un elemento que forma parte de nuestro entorno.
 - La variable *resultado* se construye dentro de la función *suma*, por lo tanto, es de tipo local. Como podéis ver, no aparece a la derecha como un elemento de nuestro entorno global.
 - La variable *numero* se construye en la ejecución global, por eso aparece a la derecha como un elemento que forma parte de nuestro entorno.
 - La variable *resultado_obtenido* se construye en la ejecución global, por eso aparece a la derecha como un elemento que forma parte de nuestro entorno.
-

A continuación, vamos a construir una nueva función.

Se va a llamar *operaciones_numericas* y va a recibir los parámetros *a* y *b* (igual que en la función *suma*). Tras ello, calculará la suma, resta, multiplicación y división, habiendo comprobado que la información recibida es numérica. En caso contrario, devolverá un mensaje de error.

Las funciones de R solo permiten la devolución de una variable, por lo que, **si queremos devolver más de un elemento, debemos introducirlos en una variable estructural**. En este caso usaremos una lista.

```
#Creamos una función que calcula la suma, resta, multiplicación y división de los dos
#números que recibe. Devuelve una lista con los resultados.

operaciones_numericas <- function(a, b = 10){

  if( is.numeric(a) & is.numeric(b) ){

    suma <- a + b
    resta <- a - b
    multiplicacion <- a * b
    division <- a / b

    return(list(suma=suma,resta=resta,multiplicacion=multiplicacion,division=division))

  } else {
    return("No se ha podido realizar la suma. Las variables no son de tipo numerico")
  }
}
```

Hasta el momento, hemos creado funciones relativamente sencillas pensadas para recibir variables básicas de tipo numérico. Sin embargo, la flexibilidad que nos da R en este sentido es mucho mayor.

Las funciones pueden recibir cualquier tipo de objeto (listas, vectores, dataframes...), y por supuesto, si resulta necesario, también pueden llamar a otras funciones.

Como siempre, la mejor forma de entenderlo es llevándolo a la práctica:

Recibiendo un vector

La siguiente función calcula el rango estadístico de un vector (diferencia entre el valor máximo y mínimo), siempre que sea de tipo numérico. En caso contrario, imprime un mensaje de error y devuelve NULL.

```
rango_estadistico <- function(vector){  
  if( is.numeric(vector) ){  
    maximo <- max(vector)  
    minimo <- min(vector)  
    rango_estadistico <- maximo - minimo  
    return(rango_estadistico)  
  } else {  
    print("No se ha podido realizar el calculo. La variable no es de tipo numerico")  
    return(NULL)  
  }  
}
```

Activar Windo
Ve a Configuració

Recibiendo un dataframe

La siguiente función recibe un dataframe y calcula el rango estadístico de cada una de sus columnas. Para ello, las recorre una por una, llama a la función *rango_estadistico()* y guarda el resultado en un vector. Debemos tener en cuenta que aquellas columnas que no sean numéricas no aparecerán en el vector salida (resultado NULL).

```
rango_estadistico_columnas_dataframe <- function(df){  
  columnas <- names(df)  
  resultado <- c()  
  for( columna in columnas ){  
    if( is.numeric(df[,columna] ){  
      resultado[columna] <- rango_estadistico( df[,columna] )  
    }  
  }  
  return(resultado)  
}
```

Vectorización

X Edix Educación

Después de haber aprendido a construir nuestras propias funciones, toca responder a la siguiente pregunta: ¿qué pasa si queremos aplicar una función sobre cada uno de los elementos que se encuentran almacenados en una variable estructural?

La respuesta nos lleva a la **vectorización** y a la familia de funciones **apply**.

Llevándolo a nuestros ejemplos, se trataría de aplicar la función *suma()* sobre cada uno de los números almacenados en un vector, o de aplicar la función *rango_estadístico()* sobre cada uno de los vectores almacenados en una lista.

Puede que la primera solución que se os haya venido a la cabeza sea la de trabajar con bucles *for*, y, aunque sí nos permiten hacer frente al problema, no son la opción más recomendada en términos de eficiencia, sencillez y calidad de código.

```
#Recorremos un vector numérico y aplicamos la función suma sobre cada uno de sus elementos. #Por lo general, hacerlo con un bucle for es una opción no recomendada.

vector_numérico <- c(1,2,3,4,5,6,7,8,9,10)

vector_resultado <- c()

for (i in vector_numérico){

  vector_resultado[i] <- suma(i)

}
```

La vectorización nos permite aplicar una función de forma directa sobre todos los elementos de un vector, matriz o dataframe. Su uso solo está disponible con ciertas funciones y operaciones que se encuentran vectorizadas en R.

Dado que el operador suma está vectorizado, podemos usar el ejemplo anterior.

Vectorización sobre un vector

```
#Aplicamos la función suma de forma vectorizada sobre un vector

vector_numérico <- c(1,2,3,4,5,6,7,8,9,10)

vector_resultado <- suma(vector_numérico)
```

Vectorización sobre una matriz

```
#Aplicamos la funcion suma de forma vectorizada sobre una matriz  
matriz_numerica <- matrix( c(1,2,3,4,5,6,7,8,9,10) , nrow = 5 , ncol = 2)  
matriz_resultado <- suma(matriz_numerica)
```

Vectorización sobre un dataframe

```
#Aplicamos la funcion suma de forma vectorizada sobre un dataframe  
vector_numerico <- c(1,2,3,4,5,6,7,8,9,10)  
df_numerico <- data.frame(columna1=vector_numerico, columna2=vector_numerico)  
df_resultado <- suma(df_numerico)
```

Como ya hemos comentado, esta metodología solo está disponible con ciertas operaciones que están vectorizadas. Si no disponemos de esta posibilidad, debemos usar la familia de funciones apply.

Apply family

X Edix Educación

Vamos a empezar con una definición.

Apply family es un conjunto de funciones RBase que nos permite **aplicar una función sobre cada uno de los elementos almacenados en una variable estructural.**

Existen 8 diferentes, pero vamos a centrarnos en las más importantes y generales, *lapply()* y *apply()*.

lapply()

La función *lapply()* está definida para aplicar una función sobre cada uno de los elementos de una lista, aunque en la práctica también podemos usarla con vectores y dataframes.

Función *lapply()*

Parámetro	Clase	Definición
X	List	Lista que contiene los elementos sobre los que vamos a aplicar la función.
FUN	function	Nombre de la función que se desea aplicar sobre cada elemento (debe estar definida o importada en el entorno).
...	...	Si la función necesita algún parámetro más, se pueden incluir de forma adicional.

Veamos un par de ejemplos:

LAPPLY() SOBRE LISTA

LAPPLY() SOBRE VECTOR

```
#Aplicamos la función suma a cada uno de los elementos de una lista. Para eso hacemos uso #de la funcion lapply.

lista_numerica <- list(1,2,3,4,5,6,7,8,9,10)

lista_resultado <- lapply( X = lista_numerica , FUN = suma )

#Podemos dar valor al parámetro b de la función suma

lista_resultado_2 <- lapply( X = lista_numerica , FUN = suma , b = 25)
```

LAPPLY() SOBRE LISTA

LAPPLY() SOBRE VECTOR

```
#Aplicamos la función suma a cada uno de los elementos de un vector. Para eso hacemos uso #de la funcion lapply.  
vector_numerico <- c(1,2,3,4,5,6,7,8,9,10)  
lista_resultado <- lapply( X = vector_numerico , FUN = suma )  
#Podemos dar valor al parámetro b de la función suma  
lista_resultado_2 <- lapply( X = vector_numerico , FUN = suma , b = 25)
```

Al igual que en el resto de casos, podríamos barajar la opción de construir un bucle que vaya recorriendo los elementos uno por uno. Sin embargo, como ya hemos comentado, no es la opción más recomendable ni elegante cuando estamos programando en R.

Imaginemos ahora que queremos aplicar la función *rango_estadístico()*, la cual recibe un vector, sobre un conjunto de vectores almacenados en una lista o en un dataframe (sus columnas son vectores). Para ello, también debemos usar *lapply()*.

LAPPLY() SOBRE LISTA

LAPPLY() SOBRE DATAFRAME

```
#Aplicamos la funcion rango_estadistico sobre dos vectores que estan almacenados en una  
#lista  
  
vector_numerico_1 <- c(1,2,3,4,5)  
  
vector_numerico_2 <- c(10,12,14,16,18)  
  
lista_de_vectores <- list( vector_numerico_1 , vector_numerico_2)  
  
lista_resultado <- lapply(X = lista_de_vectores , FUN = rango_estadistico)
```

LAPPLY() SOBRE LISTA

LAPPLY() SOBRE DATAFRAME

```
#Aplicamos la funcion rango_estadistico sobre las dos columnas que forman el dataframe  
#(son dos vectores)  
  
vector_numerico_1 <- c(1,2,3,4,5)  
  
vector_numerico_2 <- c(10,12,14,16,18)  
  
df_numerico <- data.frame(columna1 = vector_numerico_1, columna2 = vector_numerico_2)  
  
lista_resultado <- lapply(X = df_numerico , FUN = rango_estadistico)
```

apply()

La función *apply()* está definida para aplicar una determinada función sobre cada una de las filas o cada una de las columnas de una matriz, aunque en la práctica también se puede utilizar con un dataframe.

Función <i>apply()</i>		
Parámetro	Clase	Definición
X	matriz	Matriz sobre la que vamos a aplicar la función.
MARGIN	numeric	Dimensión sobre la que vamos a aplicar la función. Si vale 1, aplicamos la función sobre cada fila. Si vale 2, aplicamos la función sobre cada columna.
FUN	function	Nombre de la función que se desea aplicar (debe estar definida o importada en el entorno).
...	...	Si la función necesita algún parámetro más, se pueden incluir de forma adicional.

A continuación, vamos a aplicar la función *rango_estadistico()* sobre las filas y columnas de una matriz (para un dataframe es la misma metodología).

FILAS

COLUMNAS

Aplicamos la función sobre las filas, obteniendo el resultado para cada una de ellas. En concreto, será un vector de 5 elementos.

```
matriz_numerica <- matrix( c(1,2,3,4,5,10,12,14,16,18) , nrow = 5 , ncol = 2)

vector_resultado <- apply( X = matriz_numerica, MARGIN = 1 , FUN = rango_estadistico)
```

FILAS

COLUMNAS

Aplicamos la función sobre las columnas, obteniendo el resultado para cada una de ellas. En concreto, será un vector de 2 elementos.

```
matriz_numerica <- matrix( c(1,2,3,4,5,10,12,14,16,18) , nrow = 5 , ncol = 2)

vector_resultado <- apply( X = matriz_numerica, MARGIN = 2 , FUN = rango_estadistico)
```

Conclusiones

X Edix Educación

En este cuarto fastbook has aprendido a crear tus propias funciones de código, y créeme cuando te digo que este es uno de los aspectos más importantes de cualquier lenguaje de programación. Gracias a ellas, vas a ser capaz de programar de una forma más elegante y eficiente.

Además, hemos abordado la vectorización y la familia de funciones apply. Como ya hemos visto, la primera de ellas nos va a resultar útil cuando nuestra función trabaja con operaciones que permiten dicha vectorización. En caso de una mayor complejidad, debemos usar funciones de la familia apply.

¡Enhорabuena! Fastbook superado

edix

Creamos Digital Workers