Funciones en R

Diego López Tamayo *

Contents

#Requisitos

Conocimiento básico del lenguaje de R base, crear y modificar objetos, utilizar condicionales (and,or,if) y ciclos (for y while). Recomiendo ver el tutorial de Básicos de R.

Introducción

Una función es un conjunto de comandos organiazados para realizar una tarea específica. R tiene una gran cantidad de funciones integradas y nosotros podemos crear nuestras propias funciones.

En R, una función es un objeto, por lo que se guarda en el ambiente general (Environment) y espera los argumentos necesarios para que la función realice las acciones.

La función a su vez realiza su tarea y devuelve el control a R base, así como cualquier resultado que pueda almacenarse en otros objetos.

Definir una función

Se crea una función en R utilizando la palabra clave **function()**. La sintaxis básica de una función R es la siguiente:

*El Colegio de México, diego.lopez@colmex.mx

```
#    function_name <- function(arg_1, arg_2, ...) {
#       Function body
#    }
```

Las diferentes partes de una función son:

- Nombre de la función (Name): este es el nombre real de la función. Se almacena en el entorno R como un objeto con este nombre.
- Argumentos (Arguments): un argumento es un marcador de posición (importa el orden en que metamos los argumentos). Cuando se invoca una función, pasa un valor al argumento. Los argumentos son opcionales; es decir, una función puede no contener argumentos. También los argumentos pueden tener valores predeterminados.
- Cuerpo de la función (Body): el cuerpo de la función contiene una colección de declaraciones que define lo que hace la función.
- Valor de retorno (Return): el valor de retorno es la última expresión en el cuerpo de la función que se evaluará y regresa como "resultado" de la función.

Veamos algunos ejemplos de funciones predeterminadas en R:

Ejemplos simples de funciones incorporadas son seq (), mean (), max (), sum () y paste (. . .) etc. Son directamente llamadas por el códgigo escrito por el usuario:

```
# Mostrar el número 3
print(3)
```

```
## [1] 3
```

```
# Mostrar la palabra "Hola"
print("Hola")
```

```
## [1] "Hola"
```

```
# Mostrar la media de una lista de números, del 25 al 82
print(mean(25:82))
```

```
## [1] 53.5
```

Podemos darnos cuenta que estamos utilizando la función **print()** con distintos tipos de argumentos, tenemos argumentos numéricos, cuerdas (strings) y otras funciones como mean(). Al parecer, print() acepta varios tipos de argumentos pero realiza la misma función con todos: los muestra o "imprime" en la pantalla.

Podemos ver una lista de funciones base de R en este enlace.

Funciones definidas por el usuario

Las funciones creadas por el usuario responden a necesidades específicas y pueden incorporar funciones base de R para crear funciones más complejas. Una vez creadas, funcionan igual que las funciones base de R ya que se agregan al "ambiente global" (Global Environment). Ejemplo:

```
# Creamos una función para imprimir el cuadrado de una secuencia de números.

new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}
```

Observemos que el código anterior crea una función que toma como argumento "a" e imprime los cuadrados de los números entre 1 y a. Por ejemplo a=6

```
# Llamamos la función new.function que proporciona 6 como argumento.
new.function(4)
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

Esta es una función sencilla que imprime por separado cada uno de los elementos del ciclo "for", no crea ningún objeto nuevo ni afecta a otos objetos.

Funciones sin argumento

También podríamos crear una función sin argumentos de la siguiente forma (notar que sobreescribimos la función anterior):

```
# Creamos una función sin argumento en function()
new.function <- function() {
   for(i in 1:5) {
      print(i^2)
   }
}

# Llamamos a la función sin proporcionar un argumento.
new.function()
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

¿Por qué queremos una función que no tome argumentos? Podría ser para ejectuar una actividad que utilizamos frecuentemente con un sólo comando. Por ejemplo la función base de R **rm(list=ls())** limpia el espacio de trabajo borrando funciones, dataframes y figuras creadas.

```
rm(list=ls())
```

Funciones con argumentos, posición y nombre

Los argumentos para una llamada de función se pueden introducir en la misma secuencia que se define en la función o se pueden inroducirse en una secuencia diferente pero asignados a los nombres de los argumentos.

```
# Creamos una función con argumentos.
new.function <- function(a,b,c) {
   result <- a * b + c
   print(result)
}

# Llamar a la función por posición de argumentos. (Importa el orden)
new.function(5,3,11)
```

```
## [1] 26
```

```
# Llame a la función por los nombres de los argumentos.
new.function(a = 11, b = 5, c = 3)
```

```
## [1] 58
```

```
# Llame a la función por los nombres de los argumentos (No importa el orden)
new.function(b = 5, c = 3,a = 11)
```

```
## [1] 58
```

Notar que la primera vez que llamamos la función por posición R asume que el número 5 corresponde a la letra "a", el número 3 a la letra "b" y el número 11 a la letra "c".

Si especificamos el cada argumento con su respectivo nombre, el orden en que los introducimos no importa.

Funciones con argumentos y valores default

Podemos definir el valor de los argumentos en la definición de la función y llamar a la función sin proporcionar ningún argumento para obtener el resultado predeterminado o introducir nuevos argumentos y utilizar la función de forma normal. Nortar que los valores default se introducen dentro de **function(default)**

```
# Creamos una función con argumentos predeterminados.
new.function <- function(a = 3, b = 6) {
   result <- a * b
   print(result)
}

# Llamamos la función sin dar ningún argumento, obtenemos valor predeterminado
new.function()
```

```
## [1] 18
```

```
# Llamamos la función con valores, la función funciona de forma normal.
new.function(9,5)
```

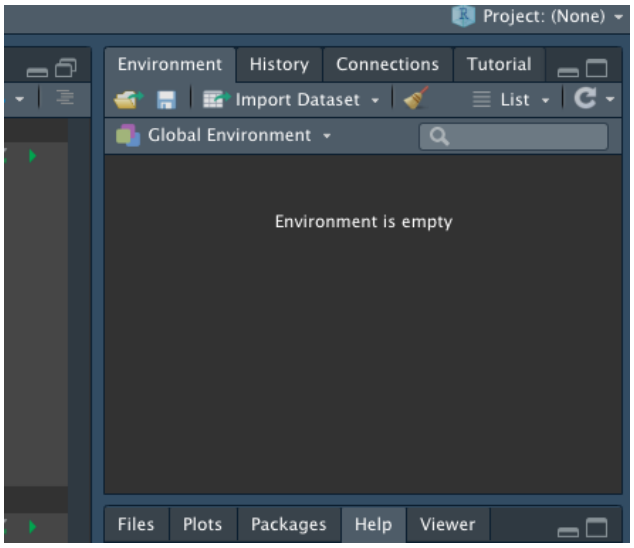
```
## [1] 45
```

```
rm(list=ls())
```

Entorno y alcance

Para escribir funciones de manera adecuada y evitar errores o comportamiento inusual, necesitamos conocer el concepto de entorno y alcance (environment and scope) en R.

El entorno puede considerarse como una colección de objetos (funciones, variables, etc.). Se crea un entorno cuando iniciamos por primera vez R. Cualquier variable que definamos ahora está en este entorno. So utilizamos R Studio podemos ver nuestro entorno en la venta superior derecha



El entorno de nivel superior disponible para nosotros en el símbolo del sistema R (command prompt) es el entorno global llamado **R_GlobalEnv.** El entorno global también puede denominarse .GlobalEnv en los códigos R.

Podemos usar la función **ls()** para mostrar qué variables y funciones se definen en el entorno actual. Además, podemos usar la función **environment()** para obtener el entorno actual.

```
a <- 2
b <- 5
f <- function(x) x<-0
ls()
```

```
## [1] "a" "b" "f"
```

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
.GlobalEnv
```

```
## <environment: R_GlobalEnv>
```

En el ejemplo anterior, podemos ver que a, b y f están en el entorno R_GlobalEnv.

Observe que x (en el argumento de la función) no está en este entorno global.

Cuando definimos una función, se crea un nuevo entorno.

En el ejemplo anterior, la función f crea un nuevo entorno dentro del entorno global. En realidad, un entorno tiene un marco propio (frame), que tiene todos los objetos definidos, y un puntero (pointer) al entorno envolvente (padre) (R_GlobalEnv). Por lo tanto, x está en el marco del nuevo entorno creado por la función f.

Funciones compuestas

En el siguiente ejemplo creamos una función anidada (nested) o compuesta.

$$f(g(x)) = f \circ g : X \to Z = (f \circ g)(x)$$

```
f <- function(f_x){
  g <- function(g_x){
```

```
    print("Inside g")
    print(environment()) # Ambiente de la función g
    print(ls()) # Objetos en g
  }
  g(5)
  print("Inside f")
  print(environment()) # Ambiente de la función f
  print(ls()) # Objetos en f
}
```

Ahora cuando la ejecutamos el código, obtenemos.

Aquí, definimos la función g dentro de f y está claro que ambos tienen diferentes entornos con diferentes objetos dentro de sus respectivos frames.

```
rm(list=ls())
```

Alcance

Consideremos el siguiente ejemplo.

```
outer_func <- function(){
  b <- 20 # Variable local de outer_function y global para inner_func
  inner_func <- function(){
    c <- 30 # Variable local de inner_func
  }
}
a <- 10 # Variable local en R_GlobalEnv y global para ambas fuciones
```

Variables globales y locales

Las variables globales son aquellas variables que existen durante la ejecución de un programa. Se puede cambiar y acceder desde cualquier parte del programa.

Sin embargo, las variables globales también dependen de la perspectiva de una función.

Por ejemplo, en el ejemplo anterior, desde la perspectiva de inner_func (), tanto “a” como “b” son variables globales.

Sin embargo, desde la perspectiva de outer_func (), b es una variable local y solo “a” es una variable global. La variable “c” es completamente invisible para outer_func().

Por otro lado, las variables locales son aquellas variables que existen solo dentro de cierta parte de un programa, como una función, y se “liberan” cuando finaliza la función.

En el programa anterior, la variable c es una variable local.

Si asignamos un valor a una variable con la función inner_func (), el cambio solo será local y no se podrá acceder desde fuera de la función.

Esto también es igual incluso si coinciden los nombres de las variables globales y las variables locales, entonces podríamos usar la misma letra en cada “nivel” de entorno sin conflicto, aunque no se recomienda porque sería confuso.

Ejemplo:

```
outer_func <- function(){
  a <- 20
  inner_func <- function(){
    a <- 30
```

```
    print(a)
  }
  inner_func()
  print(a)
}
```

Llamamos la función principal outer_func

```
a <- 10

outer_func()
```

```
## [1] 30
## [1] 20
```

```
print(a)
```

```
## [1] 10
```

Vemos que la variable a se crea localmente dentro del frame de ambas funciones y es diferente a la del frame global. La variable "a" a nivel global no cambió su valor de 10.

```
rm(list=ls())
```

Accediendo a variables globales

Las variables globales se pueden leer desde dentro de una función, pero cuando intentamos sobreescribirla, se crea una nueva variable local y no se afecta la global.

Para realizar asignaciones a variables globales, se utiliza el **operador de superasignación, «-**

Cuando se utiliza este operador dentro de una función, busca la variable en el entorno "superior" (padre), si no se encuentra, continúa buscando el siguiente nivel hasta que alcanza el entorno global. Si la variable aún no se encuentra, se crea y se asigna a nivel global.

Ejemplo:

```
outer_func <- function(){
  inner_func <- function(){
    a <<- 30
    print(a)
  }
  inner_func()
  print(a)
}
```

```
outer_func()
```

```
## [1] 30
## [1] 30
```

```
print(a)
```

```
## [1] 30
```

Cuando se encuentra la declaración a «-30 dentro de inner_func(), busca la variable a en el entorno superior outer_func(). Cuando la búsqueda falla, busca en R_GlobalEnv. Como tampoco está definida en el entorno global, se crea y asigna allí, lo que ahora se hace referencia e imprime desde dentro de inner_func () así como a external_func ().

Ejemplos de funciones

Comencemos definiendo una función fahrenheit_to_celsius que convierta las temperaturas de Fahrenheit a Celsius:

$$fahrenheit^\circ = \frac{celsius^\circ \cdot 9}{5} + 32$$

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

- Definimos fahrenheit_to_celsius asignándolo a la salida de "function"
- La lista de argumentos está contenida entre paréntesis function()
- El cuerpo de la función, las instrucciones que se ejecutan cuando se llama, se incluye entre llaves ({}). Las declaraciones en el cuerpo están sangradas (indented) por dos espacios, lo que hace que el código sea más fácil de leer pero no afecta su funcionamiento.
- Al llamar la función fahrenheit_to_celsius() espera un argumento numérico que indique la temperatura en fahrenheit y lo utiliza para el cálculo $(temp_F - 32) * 5/9$ y guarda el resultado en la variable local $temp_C$, posteriormente "devuelve" (return) la variable $temp_C$ a la función.

  En R, no es necesario incluir la declaración de devolución. R devuelve automáticamente la variable que se encuentre en la última línea del cuerpo de la función. Durante esta fase de aprendizaje, definiremos explícitamente la declaración de devolución.

Intentemos ejecutar nuestra función. Llamar a nuestra propia función es igual a llamar a cualquier otra función en R:

```
fahrenheit_to_celsius(32)
```

```
## [1] 0
```

```
# 0 celcius es el punto de congelación del agua.
```

```
fahrenheit_to_celsius(212)
```

```
## [1] 100
```

```
# 0 celcius es el punto de ebullición del agua.
```

Composición de funciones

Ahora que hemos visto cómo convertir Fahrenheit en Celsius, es fácil convertir Celsius en Kelvin:

$$celsius^\circ = kelvin^\circ - 273.15$$

```
celsius_to_kelvin <- function(temp_C) {
  temp_K <- temp_C + 273.15
  return(temp_K)
}

# Punto de congelación del agua en Kelvin
celsius_to_kelvin(0)
```

```
## [1] 273.15
```

¿Qué hay de convertir Fahrenheit a Kelvin? Podríamos escribir la fórmula, pero no necesitamos hacerlo. En cambio, podemos componer las dos funciones que ya hemos creado:

```
fahrenheit_to_kelvin <- function(temp_F) { #La función espera los grados en farenheit
  temp_C <- fahrenheit_to_celsius(temp_F) # Primero convierte de Farenheit a Celcius
  temp_K <- celsius_to_kelvin(temp_C) # Segundo convierte de Celcius a Kelvin
  return(temp_K)
}

# Punto de congelación del agua en Kelvin
fahrenheit_to_kelvin(32)
```

```
## [1] 273.15
```

Una segunda forma de calcular esto es **anidando (nesting) las funciones.**

```
# Punto de congelación del agua en Kelvin
celsius_to_kelvin(fahrenheit_to_celsius(32.0))
```

```
## [1] 273.15
```

Esta es nuestra primera muestra de cómo se crean los programas más grandes: definimos operaciones básicas y luego las combinamos en fragmentos cada vez más grandes para obtener el efecto que deseamos. Las funciones que utilizamos en R generalmente serán más grandes que las que se muestran aquí, pero las bases son las mismas.

> Pro tip: Para explorar una función en R mantenga oprimida la tecla: ctrl (Windows) cmd (Mac) doble click y dé doble click sobre la función, esto abrirá una nueva ventana con el código de la función.

Vectores como argumentos

Escribimos una función llamada **resaltar** que toma dos vectores como argumentos, llamados **contenido** y **contenedor**, y devuelve un nuevo vector que tiene el vector contenido rodeado al inico y final del vector contenedor:

```
resaltar <- function(contenido,contenedor){
  frase <- c(contenedor,contenido,contenedor)
  return(frase)
}
```

Probamos nuestra función

```
buena_practica <- c("Escribe", "código", "para", "personas",",", "no", "computadoras")
asteriscos <- "***"  # R interpreta una variable con un solo valor como un vector unitario

resaltar(buena_practica,asteriscos)
```

```
## [1] "***"          "Escribe"      "código"       "para"         "personas"
## [6] ","            "no"           "computadoras" "***"
```

Escribimos una función llamada **bordes** que devuelve un vector compuesto solo por el primer y el último elemento de su entrada:

```
bordes <- function(vector){
  borde <- c(vector[1],vector[length(vector)])
  return(borde)
}
```

```r
prueba_bordes <- c("Primera", "Segunda", "Tercera", "Cuarta")
bordes(prueba_bordes)
```

```
## [1] "Primera" "Cuarta"
```

Ejemplo del alcance

Para una comprensión más profunda de las funciones, es importante entender cómo crean sus propios entornos y llamar a otras funciones. Las llamadas a funciones se gestionan a través de la pila de ejecución o pila de llamadas (call stack). Para más detalles ver Functions to Acces the Function Call Stack.

Las funciones pueden aceptar argumentos asignados explícitamente a un nombre de variable en la función llamada functionName(variable = value), así como argumentos por orden:

```r
suma <- function(input_1, input_2 = 10) { # Notar que sólo input_2 tiene un default.
  output <- input_1 + input_2
  return(output)
}

suma(input_1 = 1, 3) # input_2=3 remplaza el default: Suma 4
```

```
## [1] 4
```

```r
suma(3) # input_1=3, input_2=default=10:  Suma 13
```

```
## [1] 13
```

```r
# suma(input_2 = 3) produce error, no hay default para input_1
```

Pruebas, manejo de errores y documentación

Una vez que comenzamos a crear funciones queremos poder reutilizarlas, para eso debemos comenzar a probar que esas funciones funcionan correctamente.