

Funciones

Arturo Pérez

1. Características generales de una función.

Las funciones representan uno de los aspectos más poderosos en programación, se trata de la transición entre usuario y programador.

Las funciones se crea utilizando el comando `function()` y se compone de las siguientes partes:

1. Nombre de la función
2. Argumentos
3. Cuerpo de la función

```
x <- function(<argumentos>){  
  ## Cuerpo de la función  
}
```

```
soyunafuncion <- function(x,y){  
  x+y  
}  
soyunafuncion(10000, 40000)
```

```
## [1] 50000
```

```
soyunafuncion <- function(x,y=10){  
  result <- x+y  
  print(result)  
}  
soyunafuncion(10)
```

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

Las funciones son objetos de primera clase y pertenecen a la clase de objetos “function”, nosotros podemos diseñar las funciones para que puedan hacer lo que nosotros queramos.

Pueden ser pasadas como argumentos a otras funciones

Pueden anidarse, es decir, meter una función dentro de otra función.

El valor regresado de una función es la última expresión en el cuerpo del código de la misma.

2. Argumentos

Argumentos formales

Son los argumentos incluidos cuando definimos una función.

Las funciones tienen varios argumentos, dentro de los cuales podemos encontrar argumentos nombrados con valores por default. No todas las funciones hacen uso de todos sus argumentos formales. Por ejemplo, una función con 10 argumentos puede tener valores especificados o nosotros debemos especificarlos (forzosamente, si no la función no hace nada). Para ver los argumentos de una función podemos hacer uso de los comandos `str()` o `formals()`

```
formals(rnorm)
```

```
## $n
##
##
## $mean
## [1] 0
##
## $sd
## [1] 1
```

```
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

argument matching

Los argumentos de las funciones pueden unirse por su posición o por el nombre, por ejemplo.

```
## por posición
set.seed(1)
rnorm(5, 3, 2)
```

```
## [1] 1.747092 3.367287 1.328743 6.190562 3.659016
```

```
##por nombre
set.seed(1)
rnorm(n=5, mean = 3, sd = 2)
```

```
## [1] 1.747092 3.367287 1.328743 6.190562 3.659016
```

```
##No recomendado invertir el orden de los args.
set.seed(1)
rnorm(mean = 3, 5, 2)
```

```
## [1] 1.747092 3.367287 1.328743 6.190562 3.659016
```

Cuando tenemos muchos argumentos, podemos mezclar el posicionamiento por nombre y por posición.

```
##equivalentes
str(lm)
lm(data = mydata, x~y, model = FALSE, 1:100)
lm(x~y, mydata, 1:100, model = FALSE) ##personalmente me gusta más este.
```

No es necesario especificar todo el tiempo, a menos que sea necesario o nos sea más cómodo. Por lo general, queremos nombrar los argumentos cuando tenemos una gran lista de estos y queremos modificar su valor por default a otro, además es útil escribirlos si sabes el nombre de los argumentos, pero no su posición en la lista.

Matching parcial

Cuando especificamos una parte del argumento, R une y le asigna el valor correcto

```
set.seed(1)
rnorm(10, m = 4, 1)
```

```
## [1] 3.373546 4.183643 3.164371 5.595281 4.329508 3.179532 4.487429 4.738325
## [9] 4.575781 3.694612
```

Entonces tenemos 3 tipos de match para los argumentos

- 1) Match exacto (por nombre)
- 2) Match parcial (por nombre, pero sin mencionar
 Todo el argumento)
- 3) Match por posición

definiendo una función

En los argumentos definimos cuando tiene y cuando no tiene un valor por default un argumento

a: No tiene valor por default (nosotros lo definimos) b, c, d: Si tienen un valor por default.

NULL: Valor para un argumento que indica que no hay
 Un valor específico

```
f <- function(a,b=1,c=2,d=NULL){}
```

Evaluación perezosa

Implica que los argumentos de una función serán evaluados sólo si se necesitan.

```
f <- function(a, b){
  a^2
}
f(2)

##Sin embargo
f <- function(a, b){
  x <- a^2
  y <- b^3
  c(x,y)
}
f(2)
```

El argumento “...”

... indica un numero variable de argumentos que son pasados a otras funciones, es decir, extendemos los argumentos de nuestra función con los argumentos de otra función.

```
fun <- function(x, y, ...){
  plot(x,y, ...)
}
```

Se replican los argumentos de la función plot, no hay necesidad de escribirlos todos.

Otra función del argumento ... se utiliza cuando un cierto número de argumentos que queremos pasar a la función no se pueden conocer por adelantado. Son objetos no definidos que son concatenados.

Por ejemplo:

```
args(data.frame)
```

```
## function (... , row.names = NULL, check.rows = FALSE, check.names = TRUE,
##      fix.empty.names = TRUE, stringsAsFactors = FALSE)
## NULL
```

```
args(rbind)
```

```
## function (... , deparse.level = 1)
## NULL
```

```
args(paste)
```

```
## function (... , sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

En esta ocasión **TODOS** los argumentos que vengan después de ... deben ser nombrados explícitamente y no pueden tener match parcial.

```
paste("a", "b", "c", sep = ":")
```

```
## [1] "a:b:c"
```

```
paste("a", "b", "c", se = ":")
```

```
## [1] "a b c :"
```

3 Reglas de ámbito

Symbol binding

¿Cómo sabe R qué valor asignar a qué símbolo cuando escribo una función?

```
lm <- function(x){x+x}
```

Cuando R intenta unir valores a un símbolo, busca a través de una serie de entornos para encontrar el valor apropiado.

Entorno: Es una colección de objetos y valores o símbolos y valores.

Esta búsqueda tiene un orden. Por ejemplo, cuando buscamos un objeto en R, este lo busca en el entorno global por nombre y símbolo, haciendo el match que buscamos, si el objeto no está ahí, entonces lo busca en los diferentes paquetes de R en este caso el paquete *stats*, si este no está ahí, lo busca en el paquete *graphics* y así sucesivamente hasta llegar al paquete *base*. Sin embargo, no hay que asumir que el programa tiene todo lo que necesitamos.

Cargar un paquete con `library()` hace que este paquete quede en la posición 2 de búsqueda moviendo un lugar al resto de paquetes.

```
library(rafalib)
```

Sin miedo al éxito nosotros podemos nombrar un objeto llamado `sd` y al mismo tiempo una función que se llame `sd()` y uno no interferirá con el funcionamiento del otro.

```
mean <- function(mean, otro){  
  y <- otro+mean  
  mean(y)  
}  
  
mean(4,5)
```

Reglas de ámbito

Lo que hacen es determinar cómo un valor se asocia con una variable libre en una función. En la siguiente función tenemos que hay 2 argumentos formales y uno libre.

```
f <- function(x,y){  
  x^2 + y/z  
}
```

FUNCION: 2 Tipos de variables.
-Argumentos
-Símbolos u objetos que no forman
parte de la función (variable libre).

¿Entonces cómo asigno un valor a esos símbolos? Hay dos formas: utilizando las reglas de *ámbito léxico* y las reglas de *ámbito dinámico*

Reglas de ámbito léxico

Esta regla nos dice que “*los valores de variables libres serán buscados en el entorno en el cual la función fue definida*”

Entorno: Colección de pares (símbolos asociados a un valor)

Cada paquete en R tiene un nombre y un entorno asociado.

Las funciones en R están asociadas a un entorno entonces, para buscar una variable libre:

1. Si el valor de un símbolo no está en el entorno donde la función fue creada, la búsqueda continua con el entorno "padre".
2. Esto se vuelve escalonado, hasta llegar al nivel top de entornos padre.

TOP: global environment.

Si esta búsqueda fracasa, se marca un error.

```
makepower <- function(n){  
  pow <- function(x){  
    x^n  
  }  
  pow  
}  
  
cube <- makepower(3)  
cube(2)
```

```
## [1] 8
```

```
square <- makepower(2)  
square(4)
```

```
## [1] 16
```

Lexico y dinámico

```
y <- 10  
f <- function(x){ ##ambito léxico  
  y <- 2  
  y^2+g  
}  
g <- function(x){ ##ambito dinámico  
  x*y  
}
```

Cuando una función es definida en el entorno global y es subsecuentemente llamado de este entorno, el entorno de definición y el entorno de llamada es el mismo y da apariencia de ámbito dinámico

```
g <- function(x){  
  a <- 3  
  x+a+y  
}  
y <- 3  
g(2)
```

```
## [1] 8
```

4. Funciones de repetición

lapply

Argumentos: list, función, ... (argumentos de la función)

```
lapply(as.list(na.omit(airquality)), quantile, probs=c(.5,.90))
```

```
## $Ozone
## 50% 90%
## 31 89
##
## $Solar.R
## 50% 90%
## 207 285
##
## $Wind
## 50% 90%
## 9.7 14.9
##
## $Temp
## 50% 90%
## 79 90
##
## $Month
## 50% 90%
## 7 9
##
## $Day
## 50% 90%
## 16 28
```

```
quantile(na.omit(airquality$Ozone), probs = c(.4,.2))
```

```
## 40% 20%
## 23 14
```

sapply

Argumentos: Los mismos que lapply.

```
sapply(as.list(na.omit(airquality)), quantile, probs=c(.5,.90))
```

```
##      Ozone Solar.R Wind Temp Month Day
## 50%    31      207  9.7  79    7  16
## 90%    89      285 14.9  90    9  28
```

apply

Argumentos: x(matriz, data frame o arreglo con 2 o más dimensiones), margin, fun, ...

```
data <- na.omit(airquality)
apply(data,2, sd)
```

```
##      Ozone      Solar.R      Wind      Temp      Month      Day
## 33.275969 91.152302  3.557713  9.529969  1.473434  8.707194
```

mapply

aplica una función en paralelo sobre un set de argumentos de esta función. Argumentos: FUN, ... (argumentos de la función), MoreArgs, SIMPLIFY.

```
mapply(rep, 1:4, 4:1)
```

```
## [[1]]  
## [1] 1 1 1 1  
##  
## [[2]]  
## [1] 2 2 2  
##  
## [[3]]  
## [1] 3 3  
##  
## [[4]]  
## [1] 4
```

```
rep(1:4, each=4:1)
```

```
## Warning in rep(1:4, each = 4:1): first element used of 'each' argument
```

```
## [1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
```

```
ruido <- function(n, mean, sd){  
  rnorm(n, mean, sd)  
}  
ruido(1:5, 1:5, 2)
```

```
## [1] 4.0235623 2.7796865 1.7575188 -0.4293998 7.2498618
```

```
mapply(ruido, 1:5, 1:5, 2)
```

```
## [[1]]  
## [1] 0.9101328  
##  
## [[2]]  
## [1] 1.967619 3.887672  
##  
## [[3]]  
## [1] 4.642442 4.187803 4.837955  
##  
## [[4]]  
## [1] 5.56427260 4.14912997 0.02129661 5.23965150  
##  
## [[5]]  
## [1] 4.887743 4.688409 2.058495 4.043700 5.835883
```


tapply

Aplica una función a un vector dividido según una variable categórica del mismo largo del vector que nos indique qué valor corresponde a cada grupo. Argumentos: x(vector), index (var. categorica), FUN, ..., simplify

```
tapply(data$Temp, factor(data$Month), mean)
```

```
##           5           6           7           8           9
## 66.45833 78.22222 83.88462 83.69565 76.89655
```

Split

Se utiliza en conjunción con otras funciones como lapply o sapply Argumentos: x(vector, lista o data.frame),f(factor o lista de factores), drop.

```
splitdata <- split(data, factor(data$Month,
                                labels = c("mayo", "junio", "julio", "agosto", "septiembre")))
sapply(splitdata, colMeans)
```

```
##           mayo           junio           julio           agosto septiembre
## Ozone      24.12500    29.44444    59.115385    60.00000     31.44828
## Solar.R    182.04167   184.22222   216.423077   173.08696    168.20690
## Wind       11.50417    12.17778     8.523077     8.86087    10.07586
## Temp       66.45833    78.22222    83.884615    83.69565    76.89655
## Month       5.00000     6.00000     7.000000     8.00000     9.00000
## Day        16.08333    14.33333    16.230769    17.17391    15.10345
```

```
lapply(splitdata, colMeans)
```

```
## $mayo
##      Ozone   Solar.R      Wind      Temp      Month      Day
## 24.12500 182.04167 11.50417 66.45833  5.00000 16.08333
##
## $junio
##      Ozone   Solar.R      Wind      Temp      Month      Day
## 29.44444 184.22222 12.17778 78.22222  6.00000 14.33333
##
## $julio
##      Ozone   Solar.R      Wind      Temp      Month      Day
## 59.115385 216.423077  8.523077 83.884615  7.000000 16.230769
##
## $agosto
##      Ozone   Solar.R      Wind      Temp      Month      Day
## 60.00000 173.08696  8.86087 83.69565  8.00000 17.17391
##
## $septiembre
##      Ozone   Solar.R      Wind      Temp      Month      Day
## 31.44828 168.20690 10.07586 76.89655  9.00000 15.10345
```

Funciones anónimas.

```
lapply(splitdata[c("mayo", "junio", "julio")], function(x) colMeans(x[,c("Ozone", "Wind", "Temp")]))
```

```
## $mayo
##      Ozone      Wind      Temp
## 24.12500 11.50417 66.45833
##
## $junio
##      Ozone      Wind      Temp
## 29.44444 12.17778 78.22222
##
## $julio
##      Ozone      Wind      Temp
## 59.115385  8.523077 83.884615
```