Funciones

Oscar Perpiñán Lamigueiro

Marzo de 2013

Contenidos

Conceptos Básicos

Lexical scope

Debug y profiling

Miscelánea

Fuentes de información

- R introduction
- R Language Definition
- Software for Data Analysis

Componentes de una función

- Una función se define con `function`

```
name <- function(arg_1, arg_2, ...)  expression
```

- Está compuesta por:
  - Nombre de la función (`name`)
  - Argumentos (`arg_1`, `arg_2`, `...`)
  - Cuerpo (`expression`): emplea los argumentos para generar un resultado

Mi primera función

- Definición

```
myFun <- function(x, y){
    x + y
    }
```

- Argumentos

```
formals(myFun)
```

```
$x


$y
```

- Cuerpo

```
body(myFun)
```

```
{
    x + y
}
```

Mi primera función

```
myFun(1, 2)
```

```
[1] 3
```

```
myFun(1:10, 21:30)
```

```
 [1] 22 24 26 28 30 32 34 36 38 40
```

```
myFun(1:10, 3)
```

```
 [1]  4  5  6  7  8  9 10 11 12 13
```

Argumentos: nombre y orden

- Una función identifica sus argumentos por su nombre y por su orden (sin nombre)

```
power <- function(x, exp){
   x^exp
   }
```

```
power(x=1:10, exp=2)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

```
power(1:10, exp=2)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

```
power(exp=2, x=1:10)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Argumentos: valores por defecto

- Se puede asignar un valor por defecto a los argumentos

```
power <- function(x, exp=2){
   x ^ exp
   }
```

```
power(1:10)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

```
power(1:10, 2)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Funciones sin argumentos

```
hello <- function(){
   print('Hello world!')
   }
```

```
hello()
```

```
[1] "Hello world!"
```

Argumentos sin nombre: . . .

```
pwrSum <- function(x, power, ...){
    sum(x ^ power, ...)
    }
```

```
x <- 1:10
pwrSum(x, 2)
```

```
[1] 385
```

```
x <- c(1:5, NA, 6:9, NA, 10)
pwrSum(x, 2)
```

```
[1] NA
```

```
pwrSum(x, 2, na.rm=TRUE)
```

```
[1] 385
```

Argumentos ausentes: `missing`

```
suma10 <- function(x, y){
   if (missing(y)) y <- 10
   x + y
   }
```

```
suma10(1:10)
```

```
 [1] 11 12 13 14 15 16 17 18 19 20
```

Control de errores: `stopifnot`

```
foo <- function(x, y){
   stopifnot(is.numeric(x) & is.numeric(y))
   x + y
   }
```

```
foo(1:10, 21:30)
```

```
 [1] 22 24 26 28 30 32 34 36 38 40
```

```
foo(1:10, 'a')
```

```
Error: is.numeric(x) & is.numeric(y) is not TRUE
```

Control de errores: stop

```
foo <- function(x, y){
   if (!(is.numeric(x) & is.numeric(y))){
      stop('arguments must be numeric.')
      } else { x + y }
   }
```

```
foo(2, 3)
```

```
[1] 5
```

```
foo(2, 'a')
```

```
Error en foo(2, "a") : arguments must be numeric.
```

Contenidos

Conceptos Básicos

Lexical scope

Debug y profiling

Miscelánea

Clases de variables

- Las variables que se emplean en el cuerpo de una función pueden dividirse en:
  - Parámetros formales (argumentos): `x`, `y`
  - Variables locales (definiciones internas): `z`, `w`, `m`
  - Variables libres: `a`, `b`

```
myFun <- function(x, y){
   z <- x^2
   w <- y^3
   m <- a*z + b*w
   m
   }
```

```
a <- 10
b <- 20
myFun(2, 3)
```

```
[1] 580
```

Lexical scope

- Las variables libres deben estar disponibles en el entorno (`environment`) en el que la función ha sido creada.

```
environment(myFun)
```

```
<environment: R_GlobalEnv>
```

```
ls()
```

```
 [1] "a"          "anidada"    "b"          "constructor" "f"
 [6] "fib"        "foo"        "hello"      "lista"       "ll"
[11] "myFoo"      "myFun"      "noise"      "power"       "pwrSum"
[16] "suma10"     "sumNoise"   "sumProd"    "sumSq"       "tmp"
[21] "x"          "zz"
```

Lexical scope: funciones anidadas


```
anidada <- function(x, y){
    xn <- 2
    yn <- 3
    interna <- function(x, y){
        sum(x^xn, y^yn)
        }
    print(environment(interna))
    interna(x, y)
    }
```

```
anidada(1:3, 2:4)
```

```
<environment: 0xb6f6f4c>
[1] 113
```

```
sum((1:3)^2, (2:4)^3)
```

```
[1] 113
```

Lexical scope: funciones anidadas

```
xn
```

```
Error: objeto 'xn' no encontrado
```

```
yn
```

```
Error: objeto 'yn' no encontrado
```

```
interna
```

```
Error: objeto 'interna' no encontrado
```

Funciones que devuelven funciones

```
constructor <- function(m, n){
   function(x){
      m*x + n
      }
   }
```

```
myFoo <- constructor(10, 3)
myFoo
```

```
function(x){
        m*x + n
        }
<environment: 0xb6eaf4c>
```

Funciones que devuelven funciones

```
class(myFoo)
```

```
[1] "function"
```

```
environment(myFoo)
```

```
<environment: 0xb6eaf4c>
```

```
ls()
```

```
 [1] "a"          "anidada"    "b"          "constructor" "f"
 [6] "fib"        "foo"        "hello"      "lista"       "ll"
[11] "myFoo"      "myFun"      "noise"      "power"       "pwrSum"
[16] "suma10"     "sumNoise"   "sumProd"    "sumSq"       "tmp"
[21] "x"          "zz"
```

```
ls(env=environment(myFoo))
```

```
[1] "m" "n"
```

```
get('m', env=environment(myFoo))
```

```
[1] 10
```

```
get('n', env=environment(myFoo))
```

```
[1] 3
```

Contenidos

Conceptos Básicos

Lexical scope

Debug y profiling

Miscelánea

Post-mortem: traceback

```
sumSq <- function(x, ...){
   sum(x ^ 2, ...)
   }

sumProd <- function(x, y, ...){
   xs <- sumSq(x, ...)
   ys <- sumSq(y, ...)
   xs * ys
   }
```

```
sumProd(rnorm(10), runif(10))
```

```
[1] 88.97738
```

```
sumProd(rnorm(10), letters[1:10])
```

```
Error en x^2 : argumento no-numérico para operador binario
```

```
traceback()
```

```
3: x^2 at #2
2: sumSq(y, ...) at #3
1: sumProd(rnorm(10), letters[1:10])
```

Analizar antes de que ocurra: `debug`

- Activa la ejecución paso a paso de una función

```
debug(sumProd)
```

- Cada vez que se llame a la función, su cuerpo se ejecuta línea a línea y los resultados de cada paso pueden ser inspeccionados.
- Los comandos disponibles son:
  - `n` o intro: avanzar un paso.
  - `c`: continua hasta el final del contexto actual (por ejemplo, terminar un bucle).
  - `where`: entrega la lista de todas las llamadas activas.
  - `Q`: termina la inspección y vuelve al nivel superior.
- Para desactivar el análisis:

```
undebug(sumProd)
```

Analizar antes de que ocurra: `trace`

- `trace` permite mayor control que `debug`

```
trace(sumProd, tracer=browser, exit=browser)
```

```
[1] "sumProd"
```

- La función queda modificada

```
sumProd
```

```
Object with tracing code, class "functionWithTrace"
Original definition:
function(x, y, ...){
    xs <- sumSq(x, ...)
    ys <- sumSq(y, ...)
    xs * ys
    }

## (to see the tracing code, look at body(object))
```

```
body(sumProd)
```

```
{
    on.exit(.doTrace(browser(), "on exit"))
    {
        .doTrace(browser(), "on entry")
        {
            xs <- sumSq(x, ...)
            ys <- sumSq(y, ...)
            xs * ys
        }
    }
```

Analizar antes de que ocurra: `trace`

- Los comandos `n` y `c` cambian respecto a `debug`:
  - `c` o intro: avanzar un paso.
  - `n`: continua hasta el final del contexto actual (por ejemplo, terminar un bucle).
- Para desactivar

```
untrace(sumProd)
```

¿Cuánto tarda mi función? `system.time`

```
noise <- function(sd)rnorm(1000, mean=0, sd=sd)
```

```
sumNoise <- function(nComponents){
   vals <- sapply(seq_len(nComponents), noise)
   rowSums(vals)
   }
```

```
system.time(sumNoise(1000))
```

```
   user  system elapsed
  0.188   0.004   0.191
```

¿Cuánto tarda cada parte de mi función?: `Rprof`

- Usaremos un fichero temporal

```
tmp <- tempfile()
```

- Activamos la toma de información

```
Rprof(tmp)
```

- Ejecutamos el código a analizar

```
zz <- sumNoise(1000)
```

¿Cuánto tarda cada parte de mi función?: `Rprof`

- Paramos el análisis

```
Rprof()
```

- Extraemos el resumen

```
summaryRprof(tmp)
```

```
$by.self
         self.time self.pct total.time total.pct
"rnorm"        0.16    66.67       0.16     66.67
"array"        0.06    25.00       0.06     25.00
"unlist"       0.02     8.33       0.02      8.33

$by.total
                total.time total.pct self.time self.pct
"sapply"              0.24    100.00      0.00     0.00
"sumNoise"            0.24    100.00      0.00     0.00
"rnorm"               0.16     66.67      0.16    66.67
"FUN"                 0.16     66.67      0.00     0.00
"lapply"              0.16     66.67      0.00     0.00
"simplify2array"      0.08     33.33      0.00     0.00
"array"               0.06     25.00      0.06    25.00
"unlist"              0.02      8.33      0.02     8.33
"as.vector"           0.02      8.33      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
```

Contenidos

Conceptos Básicos

Lexical scope

Debug y profiling

Miscelánea

do.call

- Ejemplo: sumar los componentes de una lista

```
lista <- list(a=rnorm(100), b=runif(100), c=rexp(100))
with(lista, sum(a + b + c))
```

```
[1] 146.6529
```

- En lugar de nombrar los componentes, creamos una llamada a una función con `do.call`

```
do.call(sum, lista)
```

```
[1] 146.6529
```

do.call

- Se emplea frecuentemente con el resultado de `lapply`

```
x <- rnorm(5)
ll <- lapply(1:5, function(i)x^i)
do.call(rbind, ll)
```

```
          [,1]       [,2]         [,3]      [,4]     [,5]
[1,]  2.131384 -0.6716343 0.1708127647 0.8274226 1.451193
[2,]  4.542797  0.4510926 0.0291770006 0.6846281 2.105961
[3,]  9.682445 -0.3029693 0.0049838041 0.5664768 3.056156
[4,] 20.637007  0.2034846 0.0008512974 0.4687157 4.435072
[5,] 43.985385 -0.1366672 0.0001454125 0.3878259 6.436146
```

- Este mismo ejemplo puede resolverse con `sapply`

```
sapply(1:5, function(i)x^i)
```

```
           [,1]      [,2]         [,3]         [,4]          [,5]
[1,]  2.1313839 4.5427973  9.682444959 2.063701e+01 43.9853847104
[2,] -0.6716343 0.4510926 -0.302969290 2.034846e-01 -0.1366672164
[3,]  0.1708128 0.0291770  0.004983804 8.512974e-04  0.0001454125
[4,]  0.8274226 0.6846281  0.566476774 4.687157e-01  0.3878259343
[5,]  1.4511930 2.1059611  3.056156073 4.435072e+00  6.4361459209
```

Reduce

- Combina sucesivamente los elementos de un objeto aplicando una función binaria

```
Reduce('+', 1:10)
## equivalente a
## sum(1:10)
```

```
[1] 55
```

Reduce

```
Reduce('/', 1:10)
```

```
[1] 2.755732e-07
```

```
Reduce(paste, LETTERS[1:5])
```

```
[1] "A B C D E"
```

```
foo <- function(u, v)u + 1 /v
Reduce(foo, c(3, 7, 15, 1, 292), right=TRUE)
## equivalente a
## foo(3, foo(7, foo(15, foo(1, 292))))
```

```
[1] 3.141593
```

Funciones recursivas

- Serie de Fibonnaci

```
fib <- function(n){
   if (n>2) {
      c(fib(n-1),
        sum(tail(fib(n-1),2)))
   } else if (n>=0) rep(1,n)
   }
```

```
fib(10)
```

```
 [1]  1  1  2  3  5  8 13 21 34 55
```