Funciones

Oscar Perpiñán Lamigueiro
http://oscarperpinan.github.io

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 in foo(1:10, "a") : 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 in foo(2, "a") : arguments must be numeric.
```

Mensajes para el usuario

`stop` para la ejecución y emite un mensaje de error

```
stop('Algo no ha ido bien.')
```

```
Error: Algo no ha ido bien.
```

`warning` no interfiere en la ejecución pero añade un mensaje a la cola de advertencias

```
warning('Quizás algo no es como debiera...')
```

```
Warning message:
Quizás algo no es como debiera...
```

`message` emite un mensaje (**no usar `cat` o `print`**)

```
message('Todo en orden por estos lares.')
```

```
Todo en orden por estos lares.
```

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"
 [5] "diamonds"    "fib"         "foo"         "hello"
 [9] "i"           "lista"       "ll"          "lmFertEduCatAgr"
[13] "m"           "M"           "makeNoise"   "myFoo"
[17] "myFun"       "power"       "pwrSum"      "suma1"
[21] "suma10"      "suma2"       "suma3"       "sumProd"
[25] "sumSq"       "tmp"         "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: 0x55e3ae6bec90>
[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: 0x55e3ae654500>
```

```
## 10*5 + 3
myFoo(5)
```

```
[1] 53
```

Funciones que devuelven funciones

```
class(myFoo)
```

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

```
environment(myFoo)
```

```
<environment: 0x55e3ae654500>
```

```
ls()
```

```
 [1] "a"           "anidada"     "b"           "constructor"
 [5] "diamonds"    "fib"         "foo"         "hello"
 [9] "i"           "lista"       "ll"          "lmFertEduCatAgr"
[13] "m"           "M"           "makeNoise"   "myFoo"
[17] "myFun"       "power"       "pwrSum"      "suma1"
[21] "suma10"      "suma2"       "suma3"       "sumProd"
[25] "sumSq"       "tmp"         "x"           "zz"
```

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

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

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

lapply

Supongamos que tenemos una lista de objetos, y queremos aplicar a cada elemento la misma función:

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

Podemos resolverlo de forma repetitiva...

```
sum(lista$a)

sum(lista$b)

sum(lista$c)
```

```
[1] -0.5135241
[1] 53.80815
[1] 85.80208
```

O mejor con `lapply` (lista + función):

```
lapply(lista, sum)
```

```
$a
[1] -0.5135241

$b
```

do.call

Supongamos que queremos usar los elementos de la lista como argumentos de una función.
Resolvemos de forma directa:

```
sum(lista$a, lista$b, lista$c)
```

```
[1] 139.0967
```

Mejoramos *un poco* con `with`:

```
with(lista, sum(a, b, c))
```

```
[1] 139.0967
```

La forma recomendable es mediante `do.call` (función + lista)

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

```
[1] 139.0967
```

do.call

Se emplea frecuentemente para adecuar el resultado de `lapply` (entrega una lista):

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

```
          [,1]      [,2]       [,3]     [,4]     [,5]
[1,] -0.7138158 0.9317848 0.56947704 1.092146 1.219154
[2,]  0.5095330 0.8682229 0.32430410 1.192784 1.486337
[3,] -0.3637127 0.8089969 0.18468374 1.302694 1.812074
[4,]  0.2596239 0.7538110 0.10517315 1.422733 2.209197
[5,] -0.1853236 0.7023897 0.05989369 1.553833 2.693352
```

Reduce

Combina sucesivamente los elementos de un objeto aplicando una función binaria

```
## (((1+2)+3)+4)+5
Reduce('+', 1:5)
```

```
[1] 15
```

Reduce

```
## (((1/2)/3)/4)/5
Reduce('/', 1:5)
```

```
[1] 0.008333333
```

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

```
[1] 4.212948
```

```
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

Ejemplo: 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
```

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] 40.00681
```

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

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

```
traceback()
```

Analizar antes de que ocurra: `debug`

`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
}
<bytecode: 0x55e3a6184de0>

## (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)
```

Más recursos

- Debugging en RStudio
  - Artículo
  - Vídeo
- *Debugging* explicado por H. Wickham

¿Cuánto tarda mi función? `system.time`

Defino una función que rellena una matriz de 10^6 filas y `n` columnas con una distribución normal:

```
makeNoise <- function(n){
    sapply(seq_len(n), function(i) rnorm(1e6))
}
```

```
M <- makeNoise(100)
summary(M)
```

```
       V1                  V2                  V3
 Min.   :-4.734116   Min.   :-4.598497   Min.   :-4.737993
 1st Qu.:-0.672882   1st Qu.:-0.674885   1st Qu.:-0.673589
 Median : 0.002169   Median : 0.000201   Median : 0.000154
 Mean   : 0.001107   Mean   :-0.000276   Mean   : 0.000282
 3rd Qu.: 0.675470   3rd Qu.: 0.673820   3rd Qu.: 0.674238
 Max.   : 5.196789   Max.   : 5.751186   Max.   : 4.540216
       V4                  V5                  V6                  V7
 Min.   :-4.746465   Min.   :-5.138755   Min.   :-4.98647   Min.   :-4.757454
 1st Qu.:-0.674177   1st Qu.:-0.672330   1st Qu.:-0.67461   1st Qu.:-0.676046
 Median : 0.001735   Median : 0.000239   Median :-0.00053   Median :-0.002339
 Mean   : 0.000301   Mean   :-0.000064   Mean   :-0.00004   Mean   :-0.001178
 3rd Qu.: 0.674728   3rd Qu.: 0.673775   3rd Qu.: 0.67227   3rd Qu.: 0.674123
 Max.   : 4.834753   Max.   : 4.986100   Max.   : 4.86728   Max.   : 4.722494
       V8                  V9                 V10
 Min.   :-4.895707   Min.   :-5.513674   Min.   :-4.870733
 1st Qu.:-0.676299   1st Qu.:-0.674142   1st Qu.:-0.675979
```

Diferentes formas de sumar

`system.time` mide el tiempo de CPU que consume un código[1].

```
system.time({
    suma1 <- numeric(1e6)
    for(i in 1:1e6) suma1[i] <- sum(M[i,])
})
```

```
 user  system elapsed
1.425   0.000   1.428
```

```
system.time(suma2 <- apply(M, 1, sum))
```

```
 user  system elapsed
2.598   0.152   2.750
```

```
system.time(suma3 <- rowSums(M))
```

```
 user  system elapsed
0.308   0.000   0.308
```

[1] Para entender la diferencia entre `user` y `system` véase explicación aquí.

¿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

```
suma1 <- numeric(1e6)
for(i in 1:1e6) suma1[i] <- sum(M[i,])

suma2 <- apply(M, 1, FUN = sum)

suma3 <- rowSums(M)
```

¿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
"apply"              1.12    36.60       2.40     78.43
"aperm.default"      0.70    22.88       0.70     22.88
"FUN"                0.52    16.99       0.52     16.99
"sum"                0.34    11.11       0.34     11.11
"rowSums"            0.32    10.46       0.32     10.46
"unlist"             0.04     1.31       0.04      1.31
"lengths"            0.02     0.65       0.02      0.65

$by.total
                total.time total.pct self.time self.pct
"apply"               2.40     78.43      1.12    36.60
"aperm.default"       0.70     22.88      0.70    22.88
"aperm"               0.70     22.88      0.00     0.00
"FUN"                 0.52     16.99      0.52    16.99
"sum"                 0.34     11.11      0.34    11.11
"rowSums"             0.32     10.46      0.32    10.46
"unlist"              0.04      1.31      0.04     1.31
"lengths"             0.02      0.65      0.02     0.65

$sample.interval
[1] 0.02
```