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" "fib"
 [6] "foo"        "hello"      "i"          "lista"       "ll"
[11] "M"          "makeNoise"  "myFoo"      "myFun"       "noise"
[16] "power"      "pwrSum"     "suma"       "suma1"       "suma10"
[21] "suma2"      "suma3"      "sumNoise"   "sumProd"     "sumSq"
[26] "tmp"        "vals"       "x"          "xx"          "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: 0x55d9bfced1a8>
[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: 0x55d9bfce8fe8>
```

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

```
[1] 53
```

Funciones que devuelven funciones

```
class(myFoo)
```

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

```
environment(myFoo)
```

```
<environment: 0x55d9bfce8fe8>
```

```
ls()
```

```
 [1] "a"          "anidada"    "b"          "constructor" "fib"
 [6] "foo"        "hello"      "i"          "lista"       "ll"
[11] "M"          "makeNoise"  "myFoo"      "myFun"       "noise"
[16] "power"      "pwrSum"     "suma"       "suma1"       "suma10"
[21] "suma2"      "suma3"      "sumNoise"   "sumProd"     "sumSq"
[26] "tmp"        "vals"       "x"          "xx"          "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] 4.656825
[1] 42.73277
[1] 103.4457
```

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

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

```
$a
[1] 4.656825

$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] 150.8353
```

Mejoramos *un poco* con `with`:

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

```
[1] 150.8353
```

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

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

```
[1] 150.8353
```

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,] -1.024612 -0.7763783  1.688847 0.53777883 -1.564490
[2,]  1.049829  0.6027632  2.852205 0.28920607  2.447630
[3,] -1.075667 -0.4679723  4.816939 0.15552890 -3.829293
[4,]  1.102141  0.3633235  8.135075 0.08364015  5.990891
[5,] -1.129267 -0.2820765 13.738899 0.04497990 -9.372691
```

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] 10.05877
```

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

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

```
traceback()
```

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
}
<bytecode: 0x55d9c47ba2a0>

## (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 `m` 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.   :-4.688361   Min.   :-5.077203   Min.   :-4.747337
 1st Qu.:-0.674881   1st Qu.:-0.676277   1st Qu.:-0.675707
 Median :-0.000478   Median :-0.001294   Median :-0.001410
 Mean   :-0.000623   Mean   :-0.001175   Mean   :-0.001552
 3rd Qu.: 0.674458   3rd Qu.: 0.673064   3rd Qu.: 0.673440
 Max.   : 4.815998   Max.   : 4.887105   Max.   : 4.797947
       V4                  V5                  V6
 Min.   :-4.490183   Min.   :-5.302761   Min.   :-5.095467
 1st Qu.:-0.674733   1st Qu.:-0.672828   1st Qu.:-0.676807
 Median :-0.000739   Median :-0.000008   Median :-0.001260
 Mean   :-0.000598   Mean   : 0.000366   Mean   :-0.001962
 3rd Qu.: 0.675618   3rd Qu.: 0.672595   3rd Qu.: 0.673771
 Max.   : 5.397561   Max.   : 5.037609   Max.   : 5.168260
       V7                  V8                  V9
 Min.   :-5.243775   Min.   :-4.736483   Min.   :-5.349814
 1st Qu.:-0.675340   1st Qu.:-0.674290   1st Qu.:-0.673804
```

Diferentes formas de sumar

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

```
 user  system elapsed
1.676   0.000   1.676
```

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

```
 user  system elapsed
2.210   0.000   2.209
```

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

```
 user  system elapsed
0.233   0.000   0.233
```