

Funciones

Oscar Perpiñán Lamigueiro

Universidad Politécnica de Madrid

① Conceptos Básicos

② Lexical scope

③ Funciones para ejecutar funciones

④ Debug

⑤ Profiling

⑥ Ejercicios

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 en 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 en 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...')
```

Aviso:

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.

- ① Conceptos Básicos
- ② Lexical scope
- ③ Funciones para ejecutar funciones
- ④ Debug
- ⑤ Profiling
- ⑥ Ejercicios

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)
```


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"      "b"      "foo"    "hello"  "myFun"  "power"  "pwrSum" "suma10" "x"
```

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: 0x562f60fa83a0>  
[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: 0x562f60fb1808>
```

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

```
[1] 53
```

Funciones que devuelven funciones

```
class(myFoo)
```

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

```
environment(myFoo)
```

```
<environment: 0x562f60fb1808>
```

```
ls()
```

```
[1] "a"           "anidada"    "b"          "constructor" "foo"        "hello"
[7] "myFoo"      "myFun"      "power"      "pwrSum"     "suma10"    "x"
```

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

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

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

```
[1] 10
```

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

```
[1] 3
```

- ① Conceptos Básicos
- ② Lexical scope
- ③ Funciones para ejecutar funciones
- ④ Debug
- ⑤ Profiling
- ⑥ Ejercicios

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] -0.4018482
```

```
[1] 52.28576
```

```
[1] 104.5826
```

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

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

```
$a  
[1] -0.4018482
```

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] 156.4665
```

Mejoramos *un poco* con with:

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

```
[1] 156.4665
```

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

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

```
[1] 156.4665
```


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,] -0.8190044 -1.721570  1.627311 9.498532e-02 -1.257119
[2,]  0.6707682  2.963802  2.648140 9.022211e-03  1.580348
[3,] -0.5493621 -5.102390  4.309345 8.569776e-04 -1.986686
[4,]  0.4499300  8.784120  7.012643 8.140030e-05  2.497501
[5,] -0.3684947 -15.122473 11.411748 7.731833e-06 -3.139657
```

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 Fibonacci

```
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
```

- ① Conceptos Básicos
- ② Lexical scope
- ③ Funciones para ejecutar funciones
- ④ **Debug**
- ⑤ Profiling
- ⑥ Ejercicios

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] 16.95814
```

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

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

```
traceback()
```

```
2: sumSq(y, ...) at #3
```

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

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)
```

Debugging con RStudio

- ▶ Artículo
- ▶ Vídeo
- ▶ *Debugging* explicado por H. Wickham
- ▶ Ejemplo: grabar en un fichero y usar *source*

```
sumSq <- function(x, ...)  
  sum(x ^ 2, ...)  
  
sumProd <- function(x, y, ...){  
  xs <- sumSq(x, ...)  
  ys <- sumSq(y, ...)  
  xs * ys  
}  
  
sumProd(rnorm(10), letters[1:10])
```

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

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)
```

- ① Conceptos Básicos
- ② Lexical scope
- ③ Funciones para ejecutar funciones
- ④ Debug
- ⑤ Profiling**
- ⑥ Ejercicios

¿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	V4
Min. :-5.202214	Min. :-4.982552	Min. :-5.016460	Min. :-5.389166
1st Qu.: -0.675160	1st Qu.: -0.675040	1st Qu.: -0.674668	1st Qu.: -0.676015
Median : 0.000027	Median : -0.001732	Median : -0.000605	Median : -0.001593
Mean :-0.000386	Mean :-0.000591	Mean : 0.000364	Mean :-0.000379
3rd Qu.: 0.675092	3rd Qu.: 0.674308	3rd Qu.: 0.675066	3rd Qu.: 0.675137
Max. : 4.814386	Max. : 4.803650	Max. : 5.147807	Max. : 4.680457
V5	V6	V7	V8
Min. :-5.061293	Min. :-5.281600	Min. :-5.040673	Min. :-4.632549
1st Qu.: -0.673658	1st Qu.: -0.676251	1st Qu.: -0.676437	1st Qu.: -0.675616
Median : 0.000872	Median : -0.000929	Median : 0.000929	Median : -0.001373
Mean : 0.000412	Mean :-0.001469	Mean : 0.000884	Mean :-0.000274
3rd Qu.: 0.676455	3rd Qu.: 0.672981	3rd Qu.: 0.676361	3rd Qu.: 0.674433
Max. : 4.701428	Max. : 4.738270	Max. : 4.710082	Max. : 4.637167
V9	V10	V11	V12
Min. :-5.023138	Min. :-4.688680	Min. :-5.013429	Min. :-4.898556
1st Qu.: -0.673479	1st Qu.: -0.673074	1st Qu.: -0.673827	1st Qu.: -0.672667
Median : 0.002535	Median : -0.000031	Median : -0.000611	Median : 0.001502

Diferentes formas de sumar

`system.time` mide el tiempo de CPU que consume un código¹.

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

```
user system elapsed  
0.984  0.005  0.989
```

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

```
user system elapsed  
1.742  0.112  1.855
```

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

```
user system elapsed  
0.230  0.000  0.229
```

¹Para 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"      1.06  45.69      1.92   82.76
"FUN"        0.50  21.55      0.50   21.55
"aperm.default" 0.30  12.93      0.30   12.93
"rowSums"    0.22   9.48      0.22   9.48
"sum"        0.18   7.76      0.18   7.76
"unlist"     0.04   1.72      0.04   1.72
"lengths"    0.02   0.86      0.02   0.86
```

```
$by.total
      total.time total.pct self.time self.pct
"apply"      1.92   82.76      1.06   45.69
"FUN"        0.50   21.55      0.50   21.55
"aperm.default" 0.30   12.93      0.30   12.93
"aperm"      0.30   12.93      0.00    0.00
"rowSums"    0.22   9.48      0.22   9.48
"sum"        0.18   7.76      0.18   7.76
"unlist"     0.04   1.72      0.04   1.72
"lengths"    0.02   0.86      0.02   0.86
```

```
$sample.interval
```

```
[1] 0.02
```

- ① Conceptos Básicos
- ② Lexical scope
- ③ Funciones para ejecutar funciones
- ④ Debug
- ⑤ Profiling
- ⑥ Ejercicios

Áreas de figuras geométricas

Escribe una función que calcule el área de un círculo, un triángulo o un cuadrado. La función empleará, a su vez, una función diferente definida para cada caso.

Conversión de temperaturas

Escribe una función para realizar la conversión de temperaturas. La función trabajará a partir de un valor (número real) y una letra. La letra indica la escala en la que se introduce esa temperatura. Si la letra es 'C', la temperatura se convertirá de grados centígrados a Fahrenheit. Si la letra es 'F' la temperatura se convertirá de grados Fahrenheit a grados Centígrados. Se usarán 2 funciones auxiliares, `cent2fahr` y `fahr2cent` para convertir de una escala a otra. Estas funciones aceptan un parámetro (la temperatura en una escala) y devuelven el valor en la otra escala.

Nota: La relación entre ambas escalas es $T_F = 9/5 \cdot T_C + 32$

Tablas de multiplicar

Construye un programa que muestre por pantalla las tablas de multiplicar del 1 al 10, a partir de dos funciones específicas. La primera función debe devolver el producto de dos valores numéricos enteros dados como parámetros. La segunda función debe mostrar por pantalla la tabla de multiplicar de un número dado como parámetro.

Números combinatorios

Escribe una función que calcule y muestre en pantalla el número combinatorio a partir de los valores n y k .

$$n k = \frac{n!}{(n - k)! \cdot k!}$$

Esta función debe estar construida en base a dos funciones auxiliares, una para calcular el factorial de un número, y otra para calcular el número combinatorio.

Fibonacci

Escribe una **función recursiva** que genere los n primeros términos de la serie de Fibonacci. Esta función aceptará el número entero n como argumento. Este valor debe ser positivo, de forma que si el usuario introduce un valor negativo la función devolverá un error.

Nota: En la serie de Fibonacci los dos primeros números son 1, y el resto se obtiene sumando los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, ...

Serie de Taylor

Escribe un conjunto de funciones para calcular la aproximación de e^{-x} mediante el desarrollo de Taylor:

$$e^{-x} = 1 + \sum_{i=1}^{\infty} \frac{(-x)^n}{n!}$$

La función principal acepta como argumentos el valor del número real x y el número de términos deseados. Se basará en otras tres funciones:

- ▶ `factorial` calcula el factorial de un número entero n .
- ▶ `potencia` calcula la potencia n de un número real x .
- ▶ `exponencial` calcula la aproximación anterior de un número real x usando n términos de la serie de Taylor.