Funciones

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

Outline

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

Conceptos Básicos

Lexical scope

Debug

Profiling

Miscelánea

Fuentes de información

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

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

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

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

Mi primera función

- Definición

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

- Argumentos

```
formals(myFun)
```

```
$x


$y
```

- Cuerpo

```
body(myFun)
```

```
{
    x + y
}
```


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

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
```


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

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
```


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

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
```


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

Funciones sin argumentos

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

```
hello()
```

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


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

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
```


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

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
```


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

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
```


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

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") (from #3) : arguments must be numeric.
```


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

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
```


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

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"          "addTask"    "anidada"    "b"          "constructor"
 [6] "createTask" "createToDo" "fib"        "foo"        "hello"
[11] "lista"      "ll"         "lmFertEdu"  "myFoo"      "myFun"
[16] "myList"     "myListOops" "myToDo"     "noise"      "power"
[21] "print"      "pwrSum"     "suma10"     "sumNoise"   "sumProd"
[26] "sumSq"      "task1"      "task2"      "tmp"        "valida"
[31] "x"          "xyplot"     "zz"
```


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

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: 0x428b178>
[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
```


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

Funciones que devuelven funciones

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

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

```
function(x)
    {
        m*x + n
    }
<environment: 0x4285600>
```

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

```
[1] 53
```


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

Funciones que devuelven funciones

```
class(myFoo)
```

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

```
environment(myFoo)
```

```
<environment: 0x4285600>
```

```
ls()
```

```
 [1] "a"          "addTask"    "anidada"    "b"          "constructor"
 [6] "createTask" "createToDo" "fib"        "foo"        "hello"
[11] "lista"      "ll"         "lmFertEdu"  "myFoo"      "myFun"
[16] "myList"     "myListOops" "myToDo"     "noise"      "power"
[21] "print"      "pwrSum"     "suma10"     "sumNoise"   "sumProd"
[26] "sumSq"      "task1"      "task2"      "tmp"        "valida"
[31] "x"          "xyplot"     "zz"
```

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

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

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

```
[1] 10
```

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

```
[1] 3
```


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

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] 87.69971
```

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

```
Error in x^2 (from #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])
```


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

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)
```


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

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
        }
```


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

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)
```


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

Más recursos

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

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

¿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
0.176   0.008   0.181
```


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

¿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)
```


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

¿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"      0.18       75       0.18        75
"array"      0.06       25       0.06        25

$by.total
                 total.time total.pct self.time self.pct
"sapply"               0.24       100      0.00        0
"sumNoise"             0.24       100      0.00        0
"rnorm"                0.18        75      0.18       75
"FUN"                  0.18        75      0.00        0
"lapply"               0.18        75      0.00        0
"array"                0.06        25      0.06       25
"simplify2array"       0.06        25      0.00        0

$sample.interval
[1] 0.02

$sampling.time
[1] 0.24
```


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

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] 147.4239
```

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

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

```
[1] 147.4239
```


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

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,] -0.385555865 -1.186135 -1.215231 0.326677461 -0.6441565
[2,]  0.148653325  1.406916  1.476786 0.106718163  0.4149377
[3,] -0.057314161 -1.668792 -1.794635 0.034862419 -0.2672848
[4,]  0.022097811  1.979413  2.180896 0.011388766  0.1721733
[5,] -0.008519941 -2.347850 -2.650292 0.003720453 -0.1109065
```

- Este mismo ejemplo puede resolverse con `sapply`

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

```
           [,1]      [,2]        [,3]       [,4]         [,5]
[1,] -0.3855559 0.1486533 -0.05731416 0.02209781 -0.008519941
[2,] -1.1861349 1.4069160 -1.66879215 1.97941261 -2.347850369
[3,] -1.2152307 1.4767856 -1.79463525 2.18089584 -2.650291569
[4,]  0.3266775 0.1067182  0.03486242 0.01138877  0.003720453
[5,] -0.6441565 0.4149377 -0.26728481 0.17217326 -0.110906533
```

Reduce

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

```
## (((1+2)+3)+4)+5
Reduce('+', 1:5)
## equivalente a
## sum(1:10)
```

```
[1] 15
```


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

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
```


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

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
```


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