R functional programming

Alberto Garfagnini

Università di Padova

Advanced R 02

Function fundamentals

- R functions can be broken into 3 components:
  - arguments : the list of arguments that describe how to call the function
  - body : the code inside the function
  - environment : the data structure that tell us how the fucntion finds the values associated with the name

```
mysum <- function(x, y) {
  # Compute the sum of 2 vectors
  x + y
}
```

```
> formals(mysum)
#> $x
#> $y
```

```
body(mysum)
#> {
#>     x + y
#> }
```

```
environment(mysum)
#> <environment: R_GlobalEnv>
```

- functions, as objects, can have `attributes`

```
attributes(mysum)
#> $srcref
#> function(x, y) {
#>   # Compute the sum of 2 vectors
#>   x + y
#> }
```

```
attr(mysum, "srcref")
#> function(x, y) {
#>   # Compute the sum of 2 vectors
#>   x + y
#> }
```

Primitive functions

- are those found in the base package
- are primarily written in C, so their `formals()`, `body` and `environment()` are all `NULL`

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")

formals(sum)
#> NULL

body(sum)
#> NULL

environment(sum)
#> NULL

typeof(sum)
#> [1] "builtin"
```


Creating functions

A "named" function

1) create a function object with `function`
2) bind it to a name with `<-`

```
mym <- function(x) {
  sin(1 / x ^ 2)
}
mym(1:4)
#> [1] 0.84147098 0.24740396 0.11088263 0.06245932
```

Anonymous functions

- it is done when a function name (i.e. binding) is not given

```
integrate(function(x) sin(x) ^ 2, 0, pi)
#> 1.570796 with absolute error < 1.7e-14
```

List of functions

- functions can be put in a list

```
lfuns <- list(
            half = function(x) x/2,
            double = function(x) x*2)
lfuns$half(10)
#> [1] 5
lfuns$double(10)
#> [1] 20
```

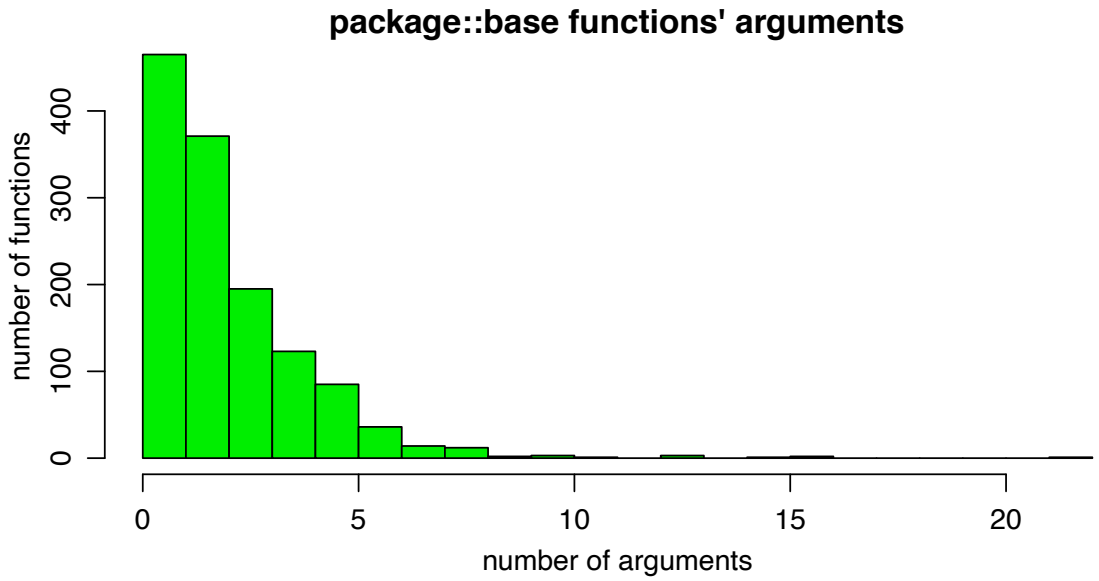
Exercise

- the following code create a list of all functions in the base package

```
objs <- mget(ls("package:base", all=TRUE), inherits=TRUE)
bfuns <- Filter(is.function, objs)
```

1➜ Determine the number of arguments for all functions and plot the distributions

2➜ How to restrict the search only to primitive functions ?



Functions calling

- R functions are normally invoked by placing the arguments in parentheses:

```
x <- c(1:3, NA, 5:10)
mean(x, na.rm=TRUE)
#> [1] 5.666667
```

- in case the functions arguments are inside a data structure
- the `do.call()` function can be called, instead:

```
x <- c(1:3, NA, 5:10)
args <- list(x, na.rm=TRUE)
do.call(mean, args)
#> [1] 5.666667
```

Functions composition

- let's imagine we need to call several functions:

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
x <- runif(10^3)
```

- we can nest the function calls

```
sqrt(mean(square(deviation(x))))
#> [1] 0.2925719
```

Functions calling (2)

- we could also store intermediate results as vectors

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
#> [1] 0.2925719
```

- but we could also use the pipe operator, %>%

```
library(magrittr)

x %>%
   deviation()  %>%
   square()  %>%
   mean()  %>%
   sqrt()
#> [1] 0.2925719
```

- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`

Lazy evaluation

- all function arguments are lazy evaluated

```
hstop <- function(x) { 10 }

hstop(1)
#> [1] 10

hstop(stop("This␣is␣an␣error!"))
#> [1] 10

stop("This␣is␣an␣error!")
#> Error: This is and error!
```

Promises

- unevaluated argument is called a promise, or a thunk.
- a promise is made up of two parts:
  - an expression, line `x + y` which gives rise to delayed computation
  - an environment, where the expression should be evaluated

Function arguments : default values

- function arguments can have default values

```
f <- function(a = 1, b = 2) c(a, b)
f()
#> [1] 1 2
```

- since arguments are evaluated lazily, default arguments can be defined in terms of other arguments

```
g <- function(a = 1,  b = a * 2) c(a, b)
g()
#> [1] 1 2
g(10)
#> [1] 10 20
```

- if an argument was supplied or not can be seen with the `missing()` function

```
i <- function(a, b) { c(missing(a), missing(b)) }
i()
#> [1] TRUE TRUE
i(a=1)
#> [1] FALSE  TRUE
i(b=1)
#> [1]  TRUE FALSE
i(1,2)
#> [1] FALSE FALSE
```


The ... (dot-dot-dot) function argument

- it is a special argument called `...`
- it will match any arguments not otherwise matched, and can be easily passed on to other functions
- one relatively sophisticated user of `...` is the base `plot()` function
- `plot()` is a generic method with arguments `x`, `y` and `...`
- simple invocations of `plot()` end up calling `plot.default()` which has many more arguments (including `...`). In this way, plot() accepts graphical parameters which are listed in the help of par()

```
plot(1:5, col = "red")
plot(1:5, cex = 5, pch = 20)

# The following allows to capture the arguments
f <- function(...) {
  names(list(...))
}
f(alpha=1, slope=3)
[1] "alpha" "slope"
```

Every operation is a function call

Golden rules

- everything that exists in R is an object
- but everything that happens is a function call
- this includes infix operators like +, control flow operators like `for`, `if`, and `while`, subsetting operators like `[]` and $, and even the curly brace {
- the backtick lets us refer to functions or variables that have otherwise reserved or illegal names

```
x <- 10; y <- 5;  x + y
[1] 15

`+`(x, y)
[1] 15

for (i in 1:2) print(i)
[1] 1
[1] 2

`for`(i, 1:2, print(i))
[1] 1
[1] 2

> { print(1)}
[1] 1
> `{`(print(1))
[1] 1
```


Every operation is a function call

- this allows to override the definitions of these special functions
- usually it is a bad idea, but it allows you to do something that would have otherwise been impossible
- example: we need to add 3 to every element of a list
- option 1: define a function `add()` and use `sapply()`:

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
[1] 4 5 6 7 8 9 10 11 12 13
```

- but we can also get the same effect using the built-in + function:

```
sapply(1:5, `+`, 3)
[1] 4 5 6 7 8

sapply(1:5, "+", 3)
[1] 4 5 6 7 8
```

- the second version works as well, because `sapply()` can be given the name of a function instead of the function itself
- it uses `match.fun()` to find functions given their names

Function arguments

- it is useful to distinguish between
- formal arguments ➜ a property of the function
- actual arguments ➜ can vary each time you call the function
- when calling a function, arguments can be specified by
- position, complete name, partial name
- arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position

```
f <- function(alpha, beta1, beta2) {
   list(a = alpha, b1 = beta1, b2 = beta2)
}
str(f(1,2,3))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(2,3,alpha=1))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(2,3,al=1))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(1,2,beta=3))
Error in f(1, 2, beta = 3) : argument 3 matches multiple formal arguments
```


Special calls: Infix functions

- most functions in R are *prefix* operators: the name of the function comes before the arguments
- infix functions are those where the function name comes in between its arguments (for instance '+' or '-')
- all user created infix functions must start and end with %
- R comes with the following infix functions predefined: %%, %*%, %/%, %in%, %o%, %x%
- the complete list of built-in infix operators that don't need % is: ::, :::, $, @, ^, *, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, <<-
- we could create a new operator that pastes together strings:

```
`%+%` <- function(a, b) paste(a, b, sep = "")
"new" %+% "␣string"
[1] "new␣string"
```

- as far as R is concerned there is no difference between these two expressions:

```
"new" %+% "␣string"
[1] "new␣string"
`%+%`("new", "␣string")
[1] "new␣string"
```

Special calls: replacement calls

- they act like they modify their arguments in place, and have the special name `xxx <-`
- they typically have two arguments (x and value), although they can have more, and they must return the modified object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:5
second(x) <- 0
x
[1] 1 0 3 4 5
```

- when R evaluates the assignment `second(x) <- 5`, it notices that the left hand side of <- is not a simple name, so it looks for a function named `second<-` to do the replacement
- if additional arguments are needed, they go in between x and value

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- -5
x
[1] -5  0  3  4  5
```


Functions : additional topics

Return values

- the last expression evaluated in a function becomes the return value

```
f <- function(x) {
  if ( x < 10 ){ 0 } else { 10 }
}
f(5)
[1] 0
```

- functions can return only a single object
- this is not a limitation because they can return a list containing any number of objects

Invisible values

- functions can return invisible values, which are not printed out by default when you call the function

```
f1 <- function() 1
f2 <- function() invisible(1)
f1()
[1] 1
f2()
```

```
f1() == 1
[1] TRUE
f2() == 1
[1] TRUE
```

- the most common function that returns invisibly is <-

Functions: `on.exit()` trigger

- functions can set up other triggers to occur when the function is finished using `on.exit()`
- the code inside `on.exit()` is always run, regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body

```
in_dir <- function(dir, code) {
   old <- setwd(dir)
   on.exit(setwd(old))
   force(code)
 }

getwd()
[1] "/Users/alberto/Documents/didattica/PhysicsOfData/R_code"

in_dir("~", getwd())
[1] "/Users/alberto"

getwd()
[1] "/Users/alberto/Documents/didattica/PhysicsOfData/R_code"
```