R functional programming

Alberto Garfagnini

Università di Padova

Advanced R 07

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)          body(mysum)
#> $x                     #> {
#> $y                     #>     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
```

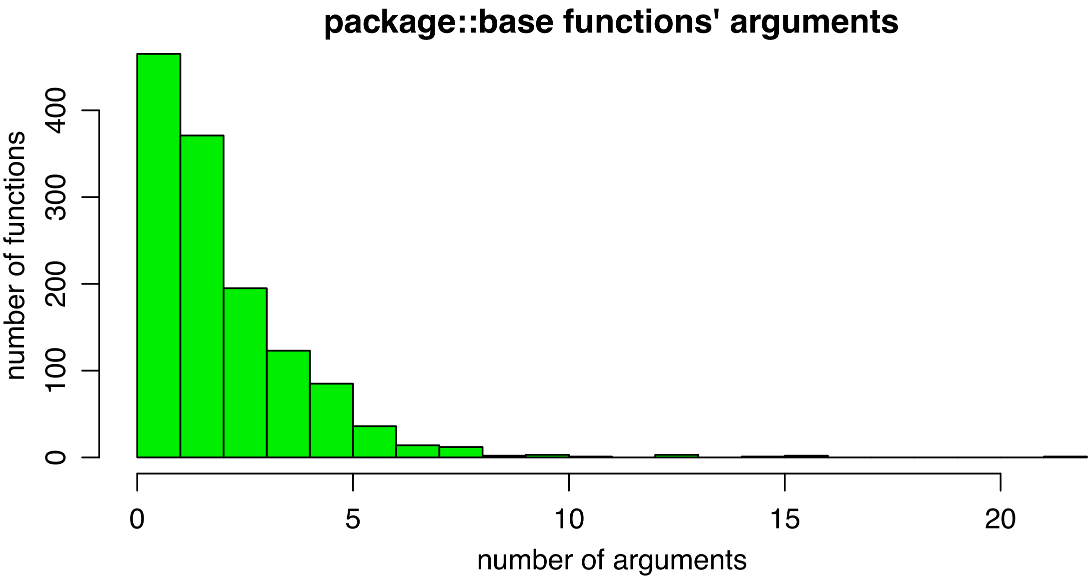
Exercise

- 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"
```

Functionals basics

Definition

a FUNCTIONAL is a function that
takes FUNCTION as INPUT
and returns a VECTOR as OUTPUT

- example:

```
randomize <- function(f) f(runif(10^3))

randomize(mean)
#> [1] 0.4954407
randomize(mean)
#> [1] 0.491658

randomize(sum)
#>[1] 507.5148
```

- typical examples in base R:

  `lapply()`, `apply()` and `tapply()`

- other example: `integrate()`

```
integrate(dnorm, -Inf, Inf)
#> 1 with absolute error < 9.4e-05
```

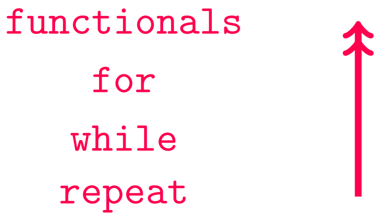
Functionals : replacement for `loops` ?

a common use of functionals is as alternative to for loops

NOTE

- `for` loops are not slow by themselves
- what makes them slow is what programmers do inside the for loop body

ex: modifying a data structure makes the loop slow because each modification creates a copy: `copy-on-modify`



`functionals`

`for`

`while`

`repeat`

- switching from loop to functional is a pattern matching exercise:

  goal: find a functional that matches the basic loop form

Our first functions: `purrr::map()`

- it takes a vector and a function
- it calls the function for each vector element
- it returns the results in a list

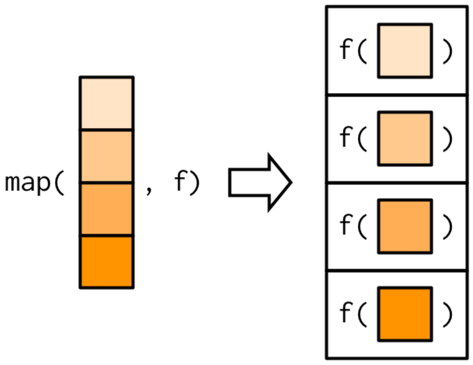
```
purrr::map(1:10, f)
```

is equivalent to

```
list(f(1), f(2), ..., f(10))
```

```
double <- function(x) x*2
xd <- purrr::map(1:10, double)
str(xd)
#>List of 10
#> $ : num 2
#> $ : num 4
...
#> $ : num 18
#> $ : num 20

unlist(xd)
#> [1]  2  4  6  8 10 12 14 16 18 20
```



Example 1

- we have a tibble with different data sets

```
dt <- tibble( a1 = rnorm(10), b1 = runif(10),
              c1 = rpois(10, 3.7), d1 = rbeta(10, 0.3, 5) )
```

- we want to evaluate the median of each colum

```
omed <- vector("double", ncol(dt))
omed
#> [1] 0 0 0 0
for (i in seq_along(dt)) {
    omed[[i]] <- median(dt[[i]])
}
omed
#> [1] 0.165312063 0.487255521 4.000000000 0.009203981
```

- it's possible to wrap up for loops in a function, and call that function instead of using the for loop directly

```
purrr::map_dbl(dt, median)
#>          a1          b1          c1          d1
#> 0.165312063 0.487255521 4.000000000 0.009203981
```

- all the map_*() functions use ... to pass along additional arguments to .f each time it's called

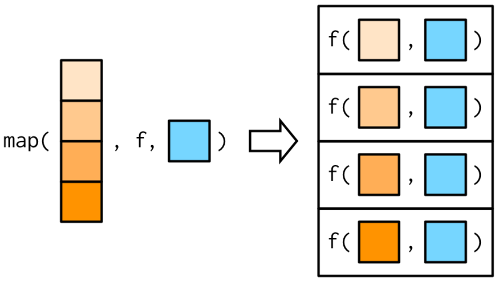
```
purrr::map_dbl(dt, mean, trim=0.5)
#>          a1          b1          c1          d1
#> 0.165312063 0.487255521 4.000000000 0.009203981
```

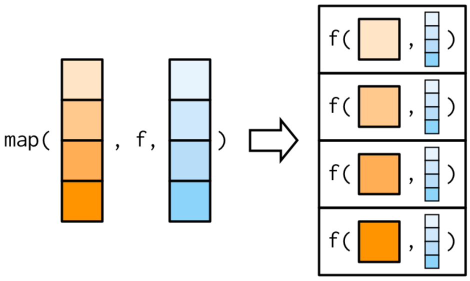
map()

the function map() returns a list:

```
     a list or vector
    ||
    \/
map(.x, .f, ...)
        /\    |-----> pass additional arguments to .f each
        ||            time it is called
        ||
        a function, formula or vector
```

- map_lgl(), map_int(), map_dbl() and map_chr() return a vector of specific type (logical, integer, double or character)
- map_dfr() and map_dfc() return a data frame created by row or by column
- any arguments that come after f in the call to map() are inserted after the data in individual calls to f()





Example 2: map()

- we generate 10 sets of random numbers from a probability distribution

```
1:10 %>% map(rnorm, n=20) -> l1
```

- this can be done using an anonyous function

```
1:10 %>% map(function(x) rnorm(n=20, x)) -> l2
```

- or by using a one-sided formula

```
1:10 %>% map( ~ rnorm(n=20, .x) ) -> l3
```

- there are a few shortcuts that you can use with .f in order to save a little typing

- .x and .y are used for two argument functions, and ..1, ..2, ..3, ... for all the additional arguments

- map() can be chained:

```
 1:10 %>%
      map(rnorm, n=20) %>%
      map_dbl(mean)
#> [1]  0.8355395  2.0266397  3.1451209  3.9854774  5.0977312
#> [6]  6.0904780  6.9547342  8.4906865  8.9917292 10.1268192
```

Mapping over multiple arguments: map2()

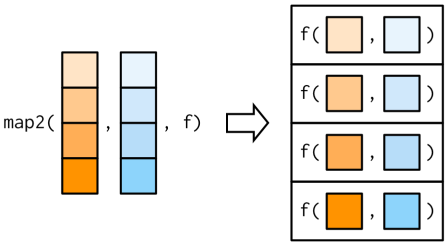
- as an example we want to generate several data sets from a normal distribution with different mean and variance

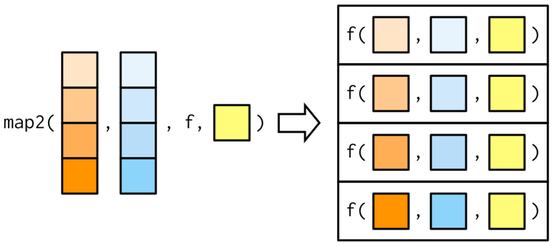
```
mus <- list(5, 10, -3)
sigmas <- list(1, 5, 10)
map2(mus, sigmas, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.17 5.24 5.54 4.8 5.44
#>  $ : num [1:5] 12.71 7.01 9.56 7.25 10.74
#>  $ : num [1:5] -8.72 9.89 -14.54 3.51 -9.49
```

- the same results could have done iterating over indices

```
seq_along(mus) %>%
          map( ~rnorm(5, mus[[.]], sigmas[[.]]) ) %>% str()
#> List of 3
#>  $ : num [1:5] 4.73 6.52 2.68 5.42 5.35
#>  $ : num [1:5] 14.913 -0.695 11.702 17.911 1.795
#>  $ : num [1:5] -4.14 -1.08 -7.85 5.79 13.73
```

- but the code with map2() is simpler and cleaner

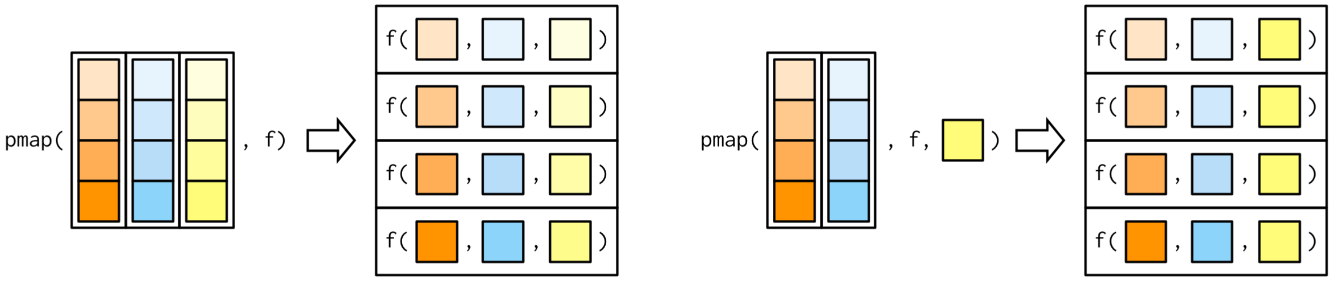




additional functions: pmap() and imap()

- in case of multiple arguments, purrr provides pmap() which takes a list of arguments
- if you don't name the list's elements, pmap() will use positional matching when calling the function. This makes the code harder to read ➜ use named arguments:

```
args2 <- list(mean = c(5, 10, -3),
              sd = c(1, 5, 10), n = 5)
args2 %>%
          pmap(rnorm) %>%
                str()
#> List of 3
#>  $ : num [1:5] 5.38 4.54 3.85 5.44 5.42
#>  $ : num [1:5] 17.038 6.072 15.107 0.697 6.488
#>  $ : num [1:5] -10.7 2.99 -1.12 17.47 13.08
```



Invoking different functions: invoke_map()

- a setp up in complexity is to invoke different functions with different parameters (values and meanings):

```
fgen <- c("runif", "rnorm", "rpois")

fpar <- list(
           list(min = -1, max = 1),
           list(sd = 3),
           list(lambda = 7.5))

invoke_map(fgen, fpar, n = 5) %>% str()

#> List of 3
#>  $ : num [1:5] -0.7744 -0.0524 0.7523 0.5074 0.5284
#>  $ : num [1:5] 3.162 -2.766 -0.298 -2.849 -2.638
#>  $ : int [1:5] 5 7 10 6 4
```

Example: reading different files and combine in one data.frame

- our data is organized in text files accoring to different years:
- data_2020_Italy.csv, data_2021_Italy.csv
- we want to read the data and combine them in one data.frame

```
read_my_csv <- function(year, country) {
    filename <- paste0(year, "_", country, ".csv")
    mobdata_dir <- "./Region_Mobility_Report_CSVs"
    filepath <- file.path(mobdata_dir, filename)
    message(paste("Reading␣from␣file:", filepath))
    read_csv(filepath)
}

years <- 2020:2021
country <- "Italy"

mbdata <- map_df(years, read_my_csv, country)

Reading from file: ./Region_Mobility_Report_CSVs/2020_IT.csv
...
Reading from file: ./Region_Mobility_Report_CSVs/2021_IT.csv
...
```