Lists and functional programming

Matteo Fasiolo

List

We have seen vectors and matrices as data structures in R.

List is another important data structure in R.

It combines objects **with different types**.

Matrix/vector only supports a single type of data.

Similar to the `struct` in C programming.

See ART: Section 4.

Creating List

```
matteo_f <- list(name = "matteo",
                 staff = TRUE,
                 salary = 10)
```

This creates a list contains three elements:

- Character data: `name`
- Logical data: `staff`
- Numeric data: `salary`

`name`, `staff` and `salary` are called “tags” for values `"matteo"`, `TRUE`, `10` respectively.

Displaying List

You can list all tags and their corresponding values by simply typing the name of the list on the console:

```
matteo_f
```

```
$name
[1] "matteo"

$staff
[1] TRUE

$salary
[1] 10
```

Indexing List

To obtain the value bound to a specific tag we can use the $ sign:

```
matteo_f$staff
```

```
[1] TRUE
```

or similarly `matteo_f[["staff"]]`.

Or you can index a list without using tags:

```
matteo_f[[2]]
```

```
[1] TRUE
```

but note

```
matteo_f[2]
```

```
$staff
[1] TRUE
```

In particular note:

```
matteo_f[[3]] + 1
```

```
[1] 11
```

while

```
matteo_f[3]
```

```
$salary
[1] 10
```

```
class(matteo_f[3])
```

```
[1] "list"
```

So we can not do

```
matteo_f[3] + 1
# Error in matteo_f[3] + 1 : non-numeric argument
# to binary operator
```

You can create a list with missing tags:

```
matteo_f <- list(name = "matteo", T, salary = 10)
```

Then you will have to access untagged elements using `matteo_f[[2]]`.

You can subset a list by doing:

```
matteo_f[c("name", "salary")]
```

```
$name
[1] "matteo"


$salary
[1] 10
```

or `matteo_f[c(1, 3)]`.

But you cannot do

```
matteo_f[[c(1, 3)]]
# Error in matteo_f[[1:3]] : subscript out of bounds
```

To add elements do

```
matteo_f <- list(name = "matteo", staff = T, sal = 10)
matteo_f$department <- "math"
matteo_f[[5]] <- 1985
matteo_f
```

```
$name
[1] "matteo"

$staff
[1] TRUE

$sal
[1] 10

$department
[1] "math"

[[5]]
[1] 1985
```

To remove an element do

```r
matteo_f$department <- NULL
matteo_f[[1]] <- NULL
matteo_f
```

```
$staff
[1] TRUE

$sal
[1] 10

[[3]]
[1] 1985
```

NOTE: before R 3.1.0 modifying a list triggered a copy of all elements, this is **not** true anymore! Modifying lists is much more efficient now.

To concatenate lists do:

```r
more_info <-  list(height = 182, employer = "UoB")
matteo_f_2 <- c(matteo_f, more_info)
matteo_f_2
```

```
$staff
[1] TRUE

$sal
[1] 10

[[3]]
[1] 1985

$height
[1] 182

$employer
[1] "UoB"
```

Note that this does something different:

```r
( matteo_f_2 <- list("l1" = matteo_f, "l2" = more_info) )
```

```
$l1
$l1$staff
[1] TRUE

$l1$sal
[1] 10

$l1[[3]]
[1] 1985



$l2
$l2$height
[1] 182

$l2$employer
[1] "UoB"
```

To create a nested list from scratch do:

```r
studs <- list(list(name = "Jack", age = 21),
              list(name = "Tim", age = 20))
studs
```

```
[[1]]
[[1]]$name
[1] "Jack"

[[1]]$age
[1] 21


[[2]]
[[2]]$name
[1] "Tim"

[[2]]$age
[1] 20
```

Then we can do:

```r
course <- list(code = "MATH0017", year = "1",
               students = studs)
course[[1]]
```

```
[1] "MATH0017"
```

```r
course[[3]][[2]]
```

```
$name
[1] "Tim"

$age
[1] 20
```

```r
course$students[[2]]$name
```

```
[1] "Tim"
```

Functional Programming

So far, we have introduced two programming paradigms

- **Procedural Programming (PP)**: Your program is divided into several subtasks and you write **functions** for each subtask.
- **Object Oriented Programming (OOP)**: Your program is divided into several pieces called "objects" and objects contain **data as well as procedures**.

PP and OOP divide the program **by features** thus is suitable for developing apps with complicated logics and components.

Functional Programming

However, most data science program has a simple programming pipeline:

- Data1 -> Op1 -> Data2 -> Op2 -> ... -> Final Result
- Apply(Op1, Data1) -> Data2-> Apply(Op2, Data2) -> Data3 -> ... -> Final Result

Functional Programming (FP) views our program as a pipeline, focusing on writing data-operating functions and applying such functions to our data.

R supports functional programming natively.

- C/C++ also supports functional programming via some advanced language features.
- Function pointers, templates, etc.

A Simple FP Example

Write a simple data operating function:

```
add <- function(x) { return( x + 1 ) }
```

Applying this function on some dummy data:

```
l <- list(1,2)
lapply(l, add)
```

```
[[1]]
[1] 2

[[2]]
[1] 3
```

Here, `lapply` applies the `add` function to each element of the list `l`, producing a new list.

The input and output of `lapply` are both lists.

We can also convert the list output to a vector by using `unlist`:

```
l <- list(1,2)
unlist(lapply(l, add))
```

```
[1] 2 3
```

More conveniently, we can use `sapply`

```
l <- list(1,2)
sapply(l, add)
```

```
[1] 2 3
```

Looking at `lapply` more in detail:

```
args(lapply)
# function (X, FUN, ...)
```

the `"..."` (aka ellipsis) means that `lapply` accepts more arguments.

These will be passed to `FUN`. Example:

```
a_plus_b <- function(a, b){ a + b }
```

```
lapply(1:3, a_plus_b, b = 10)
```

```
[[1]]
[1] 11

[[2]]
[1] 12

[[3]]
[1] 13
```

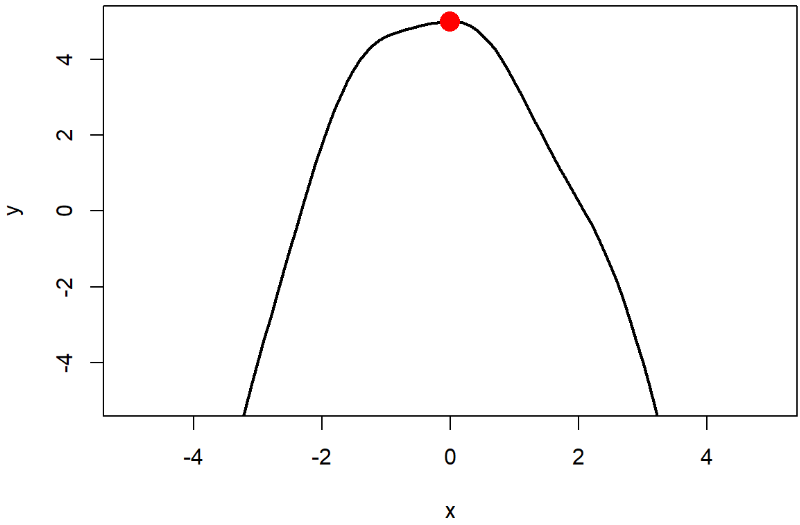
Functions are Variables

In FP, functions are variables too, **thus they can be passed to other functions as input arguments.**

In the previous example, `add` is a function that was passed to the `lapply` function as an input argument.

This property allows us to write clean and more readable code.

Gradient Ascent Rivisited



```r
f <- function (x){
  return(-sin(x)^3-x^2+ 5)
}
```

```r
df <- function(x){
  return(-3*sin(x)^2*cos(x) - 2*x)
}
```

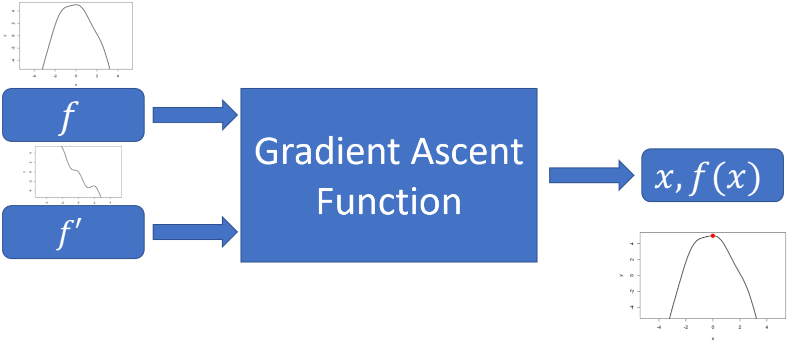
```r
x <- -1.5
while( abs(df(x)) > 0.01){
  x <- x + 0.1 * df(x)
}
x
```

```
[1] -0.004884895
```

How can we wrap the gradient ascent algorithm using a function?

Gradient Ascent Function

Gradient ascent algorithm depends on `f` and `df`, thus this gradient ascent function should take **two functional** inputs.



Here is our function:

```r
grad_asc <- function(x, f, df){

  while( abs(df(x)) > .01){

     x <- x + .1*df(x)

  }
  return(list(x, f(x)))
}
```

```r
grad_asc(x = -1.5, f, df)
```

```
[[1]]
[1] -0.004884895

[[2]]
[1] 4.999976
```

Gradient Ascent Function

In this example, the initial search point `x` is data and `f` and `df` are data operating functions.

`grad_asc` tells the program how `f` and `df` are applied to the data and produces the final outcome.

Functions are variables so we can put them in a list and simplify:

```r
grad_asc <- function(problem){
  f <- problem$func
  df <- problem$deri
  x <- problem$x

  while( abs(df(x)) > .01){
    x <- x + .1*df(x)
  }

  return(list(x, f(x)))
}
```

```r
problem <- list(x = -1.5, func = f, deri = df)

unlist( grad_asc(problem) )
```

```
[1] -0.004884895  4.999976254
```

Conclusion

1. List in R can contain data with different types.
2. Lists can be nested or concatenated using `c()`.
3. It's important to be clear about `[[]]` vs `[]`.
4. FP focuses on data operating functions and how these functions are applied on data.
5. In FP, functions can be used as variables.