Table of Contents

A deeper look at functions

In this session we take a deeper look at functions and the apply family of functions.

R makes extensive use of the *functional* programming paradigm (style); what this means in practice is that getting things done in R involves using functions. You can check out the reminder below (section 1) or in the first notebook (R1) for a definition of a function.

We will also take another look at the **apply** family of functions. These functions are used to achieve iteration in R and are basically an alternative way of looping over stuff.

Finally, it is worth working seeking out at a definition of different programming paradigms (functional, imperative, object orientated) to see where R sits in the universe of computer languages.

1. What is a function?

A function is a set of instructions (lines of codes) that you need to repeat or that you want to have self-contained to limit the complexity of your program

in R, function are build along the following principal:

function.name <- function(arguments) { body of function i.e. set of operations/instructions involving the arguments }

arguments are the elements that you import from the general program and on which you want to apply the function variables defined inside the function exists only within the function environment.

example 1

```
f1<-function (x,y) {
  x+y
}
f1(10,5)
```

```
## [1] 15
```

```
f1(2,4)
```

```
## [1] 6
```

simple, isn't it?

remember, our first function was very simple. But what happen when you have a more complex functions producing several output

```
f2<- function(x,y) {
  z1<-x+y
  z2<-x-y
  z3<-x*y
}
```

```
f2(10,5)
```

what to do to tackle this issue? You can include a return statement, with the list of output you wan to have

```
f3<- function(x,y) {
  z1<-x+y
  z2<-x-y
  z3<-x*y
  return(c(z1,z2,z3))
}
```

```
f3(10,5)
```

```
## [1] 15  5 50
```

you can now call those results,

```
f3(10,5)[[1]]
```

```
## [1] 15
```

```
f3(10,5)[[3]]
```

```
## [1] 50
```

```
f3(10,5)[[2]]
```

```
## [1] 5
```

but remember the variables inside the function (here z1, z2, and z3) only exist within the function

```
f3(10,5)$z1
```

```
## Error in f3(10, 5)$z1: $ operator is invalid for atomic vectors
```

it can be convenient to name the output

```
f4<- function(x,y) {
  z1<-x+y
  z2<-x-y
  z3<-x*y
  list(result1=z1,result2=z2,result3=z3)
}
```

```
f4(10,5)
```

```
## $result1
## [1] 15
##
## $result2
## [1] 5
##
## $result3
## [1] 50
```

now you can use those output name

```
f4(10,5)$result1
```

```
## [1] 15
```

it is also often convenient to store the output of the function into a object. For example, if there are going to be used for further operations

```
obj1<-f4(10,5)
```

what is the nature of obj1 and can you now extract the results from it?

it is also possible to set default values for the arguments

```
f5<- function(x=20,y=1) {
  z1<-x+y
  z2<-x-y
```

```
  list(result1=z1,result2=z2)
}
f5()

## $result1
## [1] 21
##
## $result2
## [1] 19

f5(10,)$result1

## [1] 11
```

finally, the arguments can be vectors or matrices.

```
# vectors

v1<-seq(1:5)
v2<-c(2,4,6,8,10)
v3<-seq(1:4)

#matrix

m1<-matrix( c(1, 2, 3, 4, 5, 6), ncol=2)
m2<-matrix( c(2, 4, 6, 8, 10, 12), ncol=2)
m3<-matrix( c(1, 2, 3, 4, 5, 6, 7, 8), ncol=2)
```

now create a function that: (1) subtract x from y, and (2) square the result apply it to v1 and v2 as well as m1 and m2 export those results to an new object and print see what happens when you use v3 or m3 instead of v1 and m1

Be careful that if the different arguments are all vectors and matrices they should be of the same length

functions can be used in loops, but R has a better way. This is the *apply* family. In comparison to a for loop, apply is typically easier to read and results in less side effects (unintended consequences). The efficiency (speed) of apply Vs a for loop will depend on the particular task at hand, you can read more about that here:

https://stackoverflow.com/questions/42393658/lapply-vs-for-loop-performance-r

2. apply-ing functions

There is a all family of function based on apply(), the idea is the same though: manipulate slices of matrix, array, list, dataframe in a repetitive way (*iterate*) and thus avoid the use of loops

apply operates on array or matrix (array of dimension 2) X

apply is written as apply(X,MARGIN,FUN)

where margin specify how the function is applied, 1 is on row and 2 is on column

FUN is the function to apply

```
X <- matrix(rnorm(30), nrow=5, ncol=6)
print(X)
```

```
##             [,1]       [,2]       [,3]      [,4]         [,5]       [,6]
## [1,] -0.2430682 -0.2321973 -1.7474844 1.7406033 -0.736532336  0.9134031
## [2,] -0.2062890  1.0827686 -0.8357843 0.9055297  2.191044726  1.0394131
## [3,] -0.7669125 -0.7751872  1.5299468 1.0312771 -0.162034550 -0.6878966
## [4,] -0.2836406  1.0183318  1.1521713 0.9900358  0.004463097  0.8340394
## [5,]  0.4085151 -0.9883006 -0.6758801 0.3769301  1.243341385  1.0820966
```

```
apply(X,2,sum)
```

```
## [1] -1.0913951  0.1054152 -0.5770306  5.0443759  2.5402823  3.1810556
```

we have summed the values of each column do the same with the lines

when you want to use a similar approach but with dataframes, lists or vectors, there is the lapply() function or sapply function

lapply returns a list sapply retruns a vector

```
#cars: The data give the speed of cars and the distances taken to stop. Note that the data were r
# [,1]   speed   numeric     Speed (mph)
# [,2]   dist    numeric     Stopping distance (ft)

#we calculate the average speed and breaking distance
CarDat<-cars
speed_av_l<-lapply(CarDat,mean)
speed_av_v<-sapply(CarDat,mean)

speed_av_l
```

```
## $speed
## [1] 15.4
##
## $dist
## [1] 42.98
```

```
speed_av_v
```

```
## speed  dist
## 15.40 42.98
```

now, find out the min and max of these values

tapply computes a measure (e.g., mean or max) of a object X (usually a vector) as a function of a factor contained in another variables.

we will use the iris data (in R), This famous (Fisher's or Anderson's) iris data set gives the measurements in centimetres of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

now we want to calculate the mean petal length per species

```
flowerDat<-iris
Petal_l_av<- tapply(flowerDat$Petal.Length, flowerDat$Species, mean)
Petal_l_av
```

```
##     setosa versicolor  virginica
##      1.462      4.260      5.552
```

to the same for the other variables

Now wouldn't it be convenient to do be able to do it for multiple variables at the same time?

mapply

That's were the mapply (m for multi) becomes handy

```
res<-mapply(mean,flowerDat[,c(1:4)])
res
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

but is it exactly what we want? no quite

```
res2<-mapply(function(x) tapply(x, flowerDat$Species, mean),flowerDat[,c(1:4)])
res2
```

```
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

aggregate

Now, there is another way to do that which I tend to prefer in such situation it relies on the aggregate function, which operate similarly as apply, but has the additional possibly to use "by" to specify on which sets of variable the function is apply independently

```
res3<-aggregate(flowerDat[,c(1:4)],by=list(flowerDat$Species),mean)
res3
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

3. What are R packages?

Before we finish, let's take a little detour into the world of packages. R packages are basically sets of functions, developed by the R user community to tackle a problem/coding need.

There are probably more than 10000 packages for R, usually available online. You can develop your own and share it.

A package usually includes some documentations, a set of functions (and possibly some data), a some test to see it works fine basic information are given the description file (available online, e.g., cran.r-project.org or stat.ethz.ch)

```
packageDescription("datasets")
```

```
## Package: datasets
## Version: 3.5.1
## Priority: base
## Title: The R Datasets Package
## Author: R Core Team and contributors worldwide
## Maintainer: R Core Team <R-core@r-project.org>
## Description: Base R datasets.
## License: Part of R 3.5.1
## Built: R 3.5.1; ; 2018-07-03 02:29:26 UTC; unix
##
## -- File: /usr/lib/R/library/datasets/Meta/package.rds
```

```
#help(package="datasets"), might not work here
```

packages are usually on repositories from where they can be installed (CRAN, GitHub,...)

to install a package from R: "might not work form notebook" "install.packages("ggplot2")"

to install from CRAN "install.packages("ggplot2", repo="https://ftp.acc.umu.se/mirror/CRAN/")"

packages can be removed with remove.packages(always specify which package) packages can be updated with update.packages(), if not specify all packages will be updated

In the notebooks we install packages ahead of time. To use the packages, you can either load it in the memory

```
library(ggplot2)
```

or you can just use one function with the following syntax *package::function*

In Rstudio, you can install and load packages from the packages tab in the lower right corner.

note: do not mix library (a command to load packages) with the packages

4. An additional exercise to test your understanding.

Here we will try to use what we have learnt to datadownloaded from SMARTSMEAR.

Read (either) data file from https://github.com/OptPhotLab/EnvDataSciNotebooks/tree/master/data

```
# hint, take a look in notebook R3-databasics
```

read the other file, and combine the two files together into a single dataframe:

create a function to calculate a single column mean. Do not use 'mean' built in, but write the formula yourself:

```
# hint, the input argument should be a vector of values
```

apply this function to both columns:

plot both daily means using ggplot