NOVA IMS Information Management School

Group Project Seminar 2014/2015

3rd lecture, 3rd Feb 2015

Júlio Caineta

jcaineta@novaims.unl.pt

Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Today we are going to learn to

- Use and define functions
  - Recursive functions
  - Built-in functions
- Apply functions to data structures
- Find, install and use packages
- Get out of the mud (!?)

NOVA IMS Information Management School

Functions

- A function is an object in R
- Take some input objects, called arguments
- Return an output object
- Everything in R is done by functions!
- Even statements are translated to functions

Syntax

- Functions are defined with the **function** keyword
- `function(arguments) body`
- *arguments* is a set of symbol names (and, optionally, default values) that will be defined within the *body* of the function
- *body* is an R expression

Arguments

- The function definition includes the names of the arguments
- You may include default values
- If an argument have a default value, then that argument is optional
- You can override a default value

```
> f = function(x, y) {x + y}
> g = function(x, y=10) {x + y}
```

Functions inside functions

- You can define a function using another function previously defined

```
> func1 = function(i) i^2
> func2 = function(i) sqrt(func1(i))
```

- That is why it is useful to specify a variable number of arguments with an ellipsis (…)

```
> v = c(sqrt(1:100))
> f = function(x,...) {print(x); summary(...)}
> f("Here is the summary for v.", v, digits=2)
```

- Now try to define a function that sums whatever elements it receives

Return values

- R will return the last evaluated expression as the result of the function

```
> f = function(x) {x^2 + 3}
> f(3)
[1] 12
```

- But you may also specify the value returned by the function

```
> f = function(x) {return(x^2 + 3)}
> f(3)
[1] 12
```

Functions as Arguments

- A function can take other functions as arguments
- Rewrite the previous example of a function inside a function

```
> func1 = function(i) i^2
> func2 = function(i) sqrt(func1(i))
```

- Functions do not need to have a name, they can be... anonymous!

```
> apply.to.three = function(f) {f(3)}
> apply.to.three(function(x) {x * 7})
[1] 21
> (function(x) {x+1})(1)
[1] 2
```

Properties

- R provides you a set of tools to manage the properties of functions, including to
  - See the function signature with the **args** function
  - Modify the function signature with the **formals** and **alist** functions
  - Even change the function **body**

Argument Order and Named Arguments

- When you define a functions, you give a name to each argument
- Inside the body of the function, you can access the arguments by name
- When you call a function, you can specify the arguments either by their exact names or order

Side effects

- Besides returning a value, a function may do other things:
  - Change variables in an environment (e.g., with **<<-** operator)
  - Plot graphics
  - Load or save files
  - Access the network
- They are called *side effects*

Applying functions

- Now we you know that, in R, we can pass functions as arguments. But when is that useful?
- As an alternative to control structures, e.g, **for** loop

```
> v = 1:20
> w = NULL
> for (i in 1:length(v)) {w[i] = v[i]^2}
> w
[1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

```
> v = 1:20
> w = sapply(v, function(i) {i^2})
> w
[1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

Applying functions

- Now we you know that, in R, we can pass functions as arguments. But when is that useful?
- As an alternative to control structures, e.g, **for** loop
- And it is faster! Check it with **system.time**
- Big family: apply, lapply, sapply, mapply, tapply

Recursive functions

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).

Graham et al., 1990

Recursive functions

Here is an example, let's define a function to compute the factorial of a given natural number

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Iterative vs recursive

```
factIter = function(n) {
  f = 1
  for (i in 2:n) f = f * i
  f
}
```

```
fact = function(n) {
  if ( n <= 1 ) 1
  else n * fact(n-1)
}
```

Packages

- A *package* is a related set of functions, help files, and data files that have been bundled together
- Why R? R offers an enormous number of packages:
  - display graphics
  - performing statistical tests
  - for industries and applications:
    - analysing microarray data
    - modelling credit risks
    - social sciences

How to get packages?

- Some of these packages are included with R: you just have to tell R that you want to use them
- Other packages are available from public package repositories
- You can even make your own packages

Using a package

- First need to make sure that it has been installed into a local *library*
- By default, packages are read from one system-level library, but you can add additional libraries

… wait, but how do I know what packages I can use?

List packages

- Loaded by default

```
> getOption("defaultPackages")
[1] "datasets" "utils" "grDevices" "graphics" "stats"
[6] "methods"
```

- Currently loaded

```
> (.packages())
[1] "stats" "graphics" "grDevices" "utils" "datasets"
"methods" [7] "base"
```

- All packages available

```
> (.packages(all.available=TRUE))
```

- List with description

```
> library()
```

Using packages

- Ok, so now we know how to use packages. Let's try the **stats** package

```
> library(stats)
```

- What can this package do for me?

```
> library(help=stats)
```

Now try some function in this package.

Get new packages

- The ultimate source for R packages is the CRAN: Comprehensive R Archive Network http://cran.r-project.org
- Currently, the CRAN package repository features 6264 available packages
- It is easy to install new packages

```
> install.packages("vardiag")
```

- List installed packages

```
> installed.packages()
```

Learn to get out of the mud

- It happens to all of us, we get stuck while trying to solve some problem
- But you can get rid of it by yourself... look for help!
- There are a lot of resources on the web, here are two to get you started
  - **Inside-R** A Community Site for R http://www.inside-r.org
  - **Stack Overflow** A question and answer site for professional and enthusiast programmers http://stackoverflow.com

Stack Overflow

- Hmmm... is it that good? Let's try...
- Remember those functions with similar names and purposes? **apply**, **lappy**, **sapply**, ***what-else*-apply**...