R programming

José R. Valverde
`jrvalverde@cnb.csic.es`
CNB/CSIC

Functions and packages

"Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice."
Sun FORTRAN Reference Manual

2024

To write or not to write

- Before writing a new function check if it already exists
  - If the function will carry out some generic, well-known, often needed task, chances are that someone will already have done it
  - And shared it, so you do not need to do it yourself
  - Search the Internet for existing implementations
  - If nothing pops-up
  - Search the Net for alternative names!

What is a function?

- A function is nothing but a piece of code (a list of commands) to carry out a specific task
- Besides grouping code, we can also “abstract” the operations specified by it.
- E.g.: a very simple function:

```
hello <- function()
    print('Hello world!')
hello()
```

A function explained

- A function is just another kind of R object.
- We create one with the command “**function( )**”
- Just like with any other function, we can assign the result to a variable with “<-” or “=”
  - In the example we assigned it to “hello”
- From then on, we will refer to this function using the name of the variable and the parentheses
  - In the example, we call it as “hello( )”
    - because it takes no arguments.

So, what does it do?

- We define the function with the command **function()** followed by the command that this function will execute.
  - In the example it is **print('Hello world!')**
- However, remember that we can also use "*compound statements*", that is, whenever we enclose a list of commands in curly brackets, they are considered as one single "virtual" command.

Defining a function

```
name <- function(arg1, arg2, …)
command
```

- Or, alternatively (which is the same)

```
name <- function(arg1, arg2, …) {
        command1
        command2
        …
}
```

Arguments

- We can specify none, one or more arguments...
  - These are variable names that will be used **inside** the function
  - They will be assigned any value used when calling the function

```
miles.to.km <- function(miles)
   miles*8/5 # note the unneeded indentation
miles.to.km(60)
miles.to.km( miles=c(60, 70, 90, 100, 120) )
# c(60, 70, 90, 100, 120) is one vector
```

Arguments explained

- Here, we have assigned first one numerical value (60) to variable “miles”
- Then, the function has executed the command “miles * 8 / 5” using the value we assigned
- In the second call, we created a vector and assigned to “miles” the vector we just created with “c()”
- Then the function has executed the command using the vector we assigned to variable “miles”.

Polymorphism

- In the previous example 'miles.to.km' works for numbers, vectors, matrices... and gives a different result for each
- That is called "polymorphism"
- Polymorphism can produce more elaborate (and differing) outputs
  - plot(x) ; plot(x, y); plot(model); ...
  - print(x); print(matrix); print(plot(x)); pint(t.test(x,y)) ...

Defining your own functions

- There are many functions already available
- But sometimes we'll want to define our own ones:

```
f <- function(x) return(2*x^2 -
                         0.9*x - 1)
f(0)
f(1.5)
```

Functions produce values

- When you use a function you get its value: you can use it as you would use any variable, e.g. when calling another function.
  - If your function produces a value, it is the same as a variable that contains a value, or as the value itself:

```
plot(f)
plot(f, from=-1, to=2)
```

Exercise

- Define the function

$$g(x) = -4x^2 + 0.2x + 4$$

- And plot it with

```
plot(g, from=-1, to=2, add=T,
            col="red")
```

Minima and maxima

- We can see that f(x) has a minimum around 0.25 and g(x) a maximum around 0
- We can find the minimum with

```
optimize(f, interval=c(-1,2))
```

- Exercise:
  - Find the maximum of g(x) using optimize

Minima and maxima

- We can see that f(x) has a minimum around 0.25 and g(x) a maximum around 0
- We can find the minimum with
  - `optimize(f, interval=c(-1,2))`
- Exercise:
  - Find the maximum of g(x) using optimize
  - **Which function has a minimum at the same point where g(x) has a maximum?**
    - **DEFINE AND CHECK IT**

Functions with several arguments

$$f(x) = -5 - 3x_2 + 4x_2 + x_1^2 - x_1 x_2 + x_2^2$$

- We can define it in several ways
  - As a function taking <u>two</u> argumens (x_1 and x_2)
  - As a function taking <u>one</u> argument, which is a vector with two components ($x = (x_1, x_2)$)
    - val = c(x1, x1) ---> val[1] = x_1, val[2] = x_2
  - As a function taking <u>one</u> argument: a list with two components ($x = x_1, x_2$))
    - val = list(x1, x1) --> val[[1]] = val$x1 = x_1, val[[2]] = val$x2 = x_2

- Try to make a function "fun" that takes two arguments (x1 and x2)

Solution

```
fun <- function(x1,x2) return(
    -5 - 3*x1 + 4*x2 + x1^2 -
    x1*x2 + x2^2)
fun(0,0)
fun(1,2)
```

Contour plot

- For functions of one argument we plot a line, for two arguments (dimensions) it is convenient to draw a contour plot.
  - The axis are X1 and X2
  - We draw lines/curves corresponding to different values of the function
    - All values of x1, x2 in a line give the same result for the function

Contour plot (2)

```
x1 <- seq(0,2,length=51)
x1
x2 <- seq(-3,1, length=51)
x2
f.vals <- outer(x1,x2,fun)
dim(f.vals)
contour(x1,x2,f.vals)
```

Contour plot (3)

- We want a plot with 50 grid points in X1 and X2
  - So we need 51 points, from 0 to 50
- Function **outer( )** will compute the function in all grid points
- And then we use function **contour( )** to draw the plot.

Another dimension

- Define a function 'fun' that calculates (x^2 + y^2)
- now try:

```
x <- seq(-10, 10, length= 30)
y <- x
z <- outer(x, y, fun)# calculate fun for all x,y points
par(bg = "white")   # set a white background
persp(x, y, z,         # plot values in x,y space
      theta = 30, phi = 30,     # help(persp)
      expand = 0.5,
      col = "lightblue")
```

default arguments

Another example

- Dataset “Florida” contains the votes for various US candidates in year 2000 elections, county by county in Florida

```
install.packages('car', dependencies=TRUE)
library(car)

data(Florida)

plot(x=Florida$BUSH, y=Florida$BUCHANAN,
     xlab="Bush", ylab="Buchanan")
```

Want to know more?

- You tried one command and saw that it worked (plot)
- Now you want to know more
  - You can type in more additional plot commands
  - But if it is more than one command, it will grow dangerously boring
    - It's never advisable to drive sleepy
  - You can group that one command (or all the commands) with { } and give it a name
    - when you use the name, it is as if you copy/paste what was between the { and } instead.

A first try

```
compare <- function() {
  plot(x=Florida[ , "BUSH"],
       y=Florida[ , "BUCHANAN"],
       xlab="Bush", ylab="Buchanan")
}
```

- `compare()`
- `compare()`
- `compare()`

Making it general

- Using arguments will allow you to make it more useful
  - We want to compare any two candidates
  - We are out of imagination
  - We'll call each candidate 'x' and 'y' (who goes on the x and who goes on the y axes)
  - If we assign the candidate names to 'x' and 'y' when calling the function then, inside the function we need to work with 'x' and with 'y' instead of the actual names.

A function to plot any pair of candidates

```
plot.florida <- function(x, y){
    x.axis <- Florida[,x] # column x
    y.axis <- Florida[,y] # column y
    plot(x.axis, y.axis, xlab=x,ylab=y)
    mtext(side=3, line=1.75,
    'Votes in Florida, by county, in the 2000 US
Presidential election') # this is one line!!!
}
plot.florida("BUSH", 'BUCHANAN')
plot.florida("BUSH")
```

Memorizing values

- Humans are, at best, sloppy with names. Remembering all the variables required by a function and their names is **highly error prone**.
- If, in addition, we require that the meaning and validity of variables be remembered, we have **a recipe for disaster**.
- A safer solution is to provide “default” values for function arguments. These are values to be used **iff** no value was provided when the function was called.

Default arguments explained

- In our example, we defined a function that takes two arguments: 'x' and 'y'.
- If we forget to specify any of them, then the function would try to make a plot with a variable that has no value (the one we forgot)
- If we provide default values and forget to give a value for any of them, then the default will be used.
- Default values are specified after an “=” sign.

A (safer) function to plot any pair of candidates

```
plot.florida <- function(x='BUSH',
                     y='BUCHANAN'){
    x.axis <- Florida[,x]
    y.axis <- Florida[,y]
    plot(x.axis, y.axis, xlab=x,ylab=y)
    mtext(side=3, line=1.75,
    'Votes in Florida, by county, in the 2000 US
Presidential election')
}
plot.florida()
plot.florida(y='GORE')
```

Default arguments at work

```
plot.florida()
```

- If we forget the values of both, 'x' and 'y', “BUSH” and “BUCHANAN” will be used.

```
plot.florida('GORE')
```

- We give a value for the first variable (x), but forgot the second, so “BUCHANAN” is used

```
plot.florida("GORE", "HARRIS")
```

- If we give both, then 'x' gets the first value and 'y' the second value.

Function results

Compute the standard deviation

```
std <- function( x ) sqrt( var( x ) )
# see? no need for indentation, it's up to you

data <- c(1, 2, 3, 3, 4, 4, 4, 4, 5, 217)
std( data )
sd <- std(data)
sd
```

- Observe that the result of the function is the result of the last command executed, and that it can be assigned to another variable

T statistic

- Let's *simulate* the t-statistic for a normal sample with mean 10 and standard deviation 5 for n samples

```
sim.t <- function( n ) {
    mu <- 10
    sigma <- 5
    # no need to remember formulas with functions
    X <- rnorm(n,mu,sigma)
    (mean(X) - mu)/(sd(X)/n)
}
sim.t(4)
```

Exercise

- Modify the function so that it allows you to set mu and sigma
- Define default values for n, mu and sigma.

T statistic (2)

- What if we want to be able to change the mean and standard deviation?

```
sim.t <- function(n=1000,mu=10,sigma=5) {
    X <- rnorm(n,mu,sigma)
    (mean(X) - mu)/(sd(X)/n)
}
sim.t(4)
sim.t(4, 3, 10)
sim.t(4, 5)
sim.t(4, sigma=100)
sim.t(sigma=100, mu=1, n=4)
sim.t() ; sim.t(sigma=100, mu=1)
```

T statistic (3)

- In this example, we see again that if we give values, the values given are assigned in the order specified when the function was defined
- We also see that, if we forget the last arguments, default values are used. Note that arguments that have no default must always be given.
- And we also see that we can specify to which argument we want to assign each value, and then we need not follow (nor remember☺) the original order.

Error! Horror!

```
average <- function(x)    sum(x) / length(x)
average( c(1,2,3,4,3,2,1) )
average( c("Tom", "Dick", "Harry") )
```

- A function will return the result of the last command executed. If that command results in an error, an error will be returned.
- We *should* check argument validity.

Dealing with uncertainty

- It is a simple matter of making decisions on what to do, and to adapt calculations as needed.

```
average = function(x) {
    if ( ! is.numeric(x) ) {
        stop("STOP! Non numeric value in x.")
    }
    sum(x)/length(x)
}
var1 = 1:5
average(var1)
average("thisisnotanumber")
```

Explanation

- We test first if the required conditions are met or not (if the value of x is not numeric...)
- *If the value is invalid*, it does not make sense to continue: we stop the <u>program</u> here and produce <u>our own</u> error message to explain what's wrong, using **stop( )**.
  - I used a very short message to fit in a slide
  - <u>You</u> should use explanatory messages
- *If the value is valid*, we do the calculations.

Dealing with uncertainty (2)

- And, of course you can make any decisions any way you like:

```
average = function(x) {
   if ( is.numeric(x) ) {
      return( sum(x)/length(x) )    # return and end here
   }
   # no need for an else, this will not be reached
   return(NA)    # return an invalid value
}
var1 = 1:5
average(var1)
average('thisisnotanumber')
```

Explanation

- Here we have changed the test: instead of checking if the argument is not a number, we check if it actually is.
- If the argument (x) is a number then we will calculate the mean
  - But now, it is not the last command
  - Which is why we use **return( )** to indicate that the result of this calculation will be the value returned (the result) of the function.
- **YOU SHOULD ALWAYS USE return( )**

Scope
(peri, micro, tele...)

Compute the square

```
square <- function( x ) {
   sq <- x * x
}
square(3)
sq

square <- function( x ) {
   sq <- x * x
   return(sq)
}
square(3)
```

What???

- The first version computes the square and assigns it into a new variable
  - This variable exists inside the function
  - But **NOT outside** the function
  - The assignment does not produce anything
  - Therefore the function does not produce anything
- We need to specify what is to be the result using a command that produces a result

Local scope

- If we create a new variable inside a function, it will exist only **within** the function, and only as long as the function is executed
  - I.e. it will not be “visible” (accessible) from outside the function
  - But will be visible (usable) within the function
- These variables are called “**local**” variables (only accesible within the local *scope*).

Global scope

- R objects are visible at the level in which they are created.
- A script is an R object (a collection of commands)
- A function is an R object (a collection of commands) **within** another object (the script)

```
square <- function() x * x
x <- 5
square( )
```

Explanation

- x is created in the script
- ==> x is visible in all the script
- therefore, x is visible in the function, which is part of the script
- x is said to be a **global variable** as it is visible in all the script (including sub-sections of the script such as functions).
  - **Note: try to avoid relying on global variables inside functions**

Local vs. Global

```
square <- function( ) {
   sq <- x * x
   return(sq)
}
x <- 5
square()
x
sq
```

Explained

- x is created in the script, therefore it is visible in all the script, including the function, which is defined in the script as well
- sq is created in the function, and visible in the function (e.g. can be seen by “return( )”)
- While the function is a part of the script, the script is **not** a part of the function, so sq cannot be seen <u>outside</u> the function

Another example

```
x <- 3
sq <- function( ) {
    print(x)   # this is the global x
    x = x * x # a new x is created inside
    print(x)   # this is the local x
}
sq()
print(x)         # this is the global x
```

Returning many values

Returning multiple objects

- Up to now, we have returned the result of the last command
- We can also return many things at once using **list( )**.
- Remember, list members are referenced with double square brackets or by name with dollar ($) notation.

```
cmplx <- function(x, i) {
    # create a complex number
    return( list(z=x, i=i) )
}

cmplx_add <- function(x, y) {
    # add two complex numbers (x) and (y)
    z = x$z + y$z
    i = x$i + y$i
    return( list(z=z, i=i) )
}

c1 <- cmplx(10, 5)
print(c1)
c2 <- cmplx(-10, 5)
print(c2)
sum.1.2 <- cmplx_add(c1, c2)
print(sum.1.2)
```

Growing up

- Most of your functions will be rather simple
- But sooner or later they will grow.
- When they do, split them into smaller functions if possible!
- But sooner or later you will not be able
- Then, if you have an error in your code, it may be difficult to hunt it.
  - HINT: print is your best friend!

Hunting errors

- Programming errors are commonly called “**bugs**”
- Eliminating errors from a program is called “**debugging**”
- You *should* use print( ) commands to follow the progress of the calculations
- You can use conditionals (if..else..) to execute part of the calculations
- You can use stop( ) to check calculations up to some point

R debugging

- Most languages have a “debugger”, a utility to locate errors. R does too.
- R will show at which point in the function (in which command) did execution stop because of errors.
- You can use “**debug(***name.of.function***)**” and then call the function normally.
- R will now execute the function line by line and you can “print( )” any variable.
- Press <ENTER> to proceed to the next line.

A sample **bad** function

```r
matmult <- function(x, y) {
  print("multiplying two matrices")
  if (dim(x) != dim(y)) {
    print("multiplying")
    mult <- x %*% y
  }
  # return the result
  return(mult)
}
```

Test runs

```
m1 <- matrix(1:12, 3, 4)
m2 <- matrix(1:24, 4, 6)
m1 %*% m2
matmult(m1, m2)
debug(matmult)
matmult(m1, m2)
# we cannot test two values
undebug(matmult)
```

Fix it

```
matmult <- function(x, y) {
  print("multiplying two matrices")
  if (dim(x)[2] == dim(y)[1]) {
    print("multiplying")
    mult <- x %*% y
    return(mult)
  }
  else
      return(NULL)
}
```

Growing up (many times)

- As you solve more problems, you will collect a larger amount of scripts.
- It is wise to keep functions in separate files so you can use them in many scripts but only have one copy to modify.
- But then you'll need to “read” this file with all those functions to add them into R
- Use the “**source( )**”, Luke!

General advice

Best practices

- **Keep your functions "short"**
  - Remember that you can call other functions from within a function, so there is no reason not to split them if needed.
  - This makes it easier to read.. and to detect bugs!
- **Add comments to explain everything:**
  - what does the function do
  - what are the arguments and their meaning
  - How are things done and why
  - What is the result

Best practices (2)

- **Check for errors on the fly**
  - *Write incrementally* and check as you write
    - i.e. write one or a few lines and then test them
  - *Add tests* to verify all assumptions and produce error messages when an assumption is violated
  - *Include* simple *examples*
    - include commented out example calls *and outputs* in your source code

Controlling output

- You can redirect the output produced by R to a file using **sink( )**

```
sink("filename", split=TRUE) # to also show output
                             # onscreen
sink('filename') # to send output only to a file
sink( )          # to stop sending output to the last file
sink( )# if we do it more than once
                 # sink(file...) then we need
                 # to close each one by one
```

- You can use **cat( )** instead of **print()** to print to the screen or to a file

Sinking the Titanic

- You can check how many "diversions" you use with "sink.number()"
- Write a simple function "titanic" that
  - takes no arguments
  - gets the number of active "sinks" with sink,number()
  - uses a for loop to close all "sinks" by calling sink()
- Add an optional example

Solution

```
titanic <- function() {
  n.active.sinks <- sink.number()
  for (i in 1:n.active.sinks)
    sink()
}
# example (set to T==T for testing, T==F otherwise)
if (T == T) {
  sink('tit'); sink('anic')
  titanic()
}
```

functional programming

Functional programming

- Note the syntax used to create a function: a function is a collection of commands stored in a variable.
  - That's why using the variable name alone shows the function's content (the code)
- As a corolary, you can use functions as variables, assign them to vectors, lists, create them inside other functions and even return a function as the result of a function.

Closure

- When a function returns another function, we call this “**closure**”.
  - Functions “written” by other functions.
- They are useful in specific circumstances, but we are not going into more detail now.

Plotting to a PNG file

```
plot(runif(1:100))
plot.and.save.to.png <- function(FUN, file=NULL) {
    FUN
    if ( ! is.null(file) ) {
        print(paste("saving to file", file))
        dev.copy(png, file)
        dev.off()
    }
}
plot.and.save.to.png(plot(runif(1:100)), file="random.png" )
```

Getting up to speed...

- As you write functions, you will sometimes come up with various alternative solutions
- All is well as long as it is well
- But if you try one solution and it takes too long...
- You can try to implement each solution and see which is faster
  - **Sys.time( )** gives the current time
  - **difftime(stop, start)** gives the difference between two times

Benchmarking

- Is the process of comparing two implementations to see which is better.
- Some tasks may take very little time but be called so many times that they end up eating most of the time of the calculation.
- We can write a function to run another function, do it a large number of times, and test how long it took.

Benchmark n times

```
square <- function(x) x * x
square2 <- function(x) x^2

bench <- function(FUN, n=100, ...) {
    started.at <- Sys.time()
    for ( i in 1:n) {
        FUN(...)
    }
    finished.at <- Sys.time()
    difftime(finished.at, started.at)
}

bench(FUN=square, 1000000, 10)    # try also square2
```

What???

- We first define a function square that we want to benchmark
- Then we define a second function to see how long does it take to execute **any** function a number of times (by default 100)
- We finally call bench( ) asking it to run square( ) a million times...

Yeah, I see that, but ... WTF???

- … is a special kind of argument variable.
- These three dots mean “everything that follows” and we can use them (the three dots) to pass on all this “everything that follows” to an internal function, when we use the 'variable' “...” again.
- Here we use this trick in order to be able to execute any function:
  - **FUN** is the function name
  - **n** is the number of times we will run the function
  - … is(are) the argument(s) to the function
- This allows us to use any function as the funtion to test (FUN) no matter how many arguments it takes
  - We simply add the arguments and they will be passed along.

Anonymous (lambda) functions

- Sometimes, we may want to define functions "without a name"
- Normally we will do it to create a function that we want to pass as argument to another function
- We do this when we will never need to use again that function elsewhere or it is too simple to deserve a name

```
FUN <- function(x) x*x
bench(FUN=FUN, 1000000, 10)


bench(function(x) x*x, 1000000, 10)     # λ
bench(FUN=function(x) x*x, 100000, 10) # λ
```

Function properties

- Try with any of the functions we have created
- **formals(** *func.name* **)**
- **body(** *func.name* **)**
- **environment(** *func.name* **)**
- **args(** *func.name* **)**

Packages

Packages

- Packages are the fundamental unit of shared code
  - They bundle together code, data, documentation and tests
  - That can be shared with others, usually through CRAN, the **C**omprehensive **R** **A**rchive **N**etwork.
- CRAN is the public hub where you can find most packages written by others

Installation

- You install a package with
  - install.packages(“packagename”)
- It is often a good idea to use

  install.packages(“packagename”, dependencies=T)

  to ensure that if one package needs code from other packages, those others are automatically installed as well.
- R will contact CRAN to download and install the package.
  - Usually R will ask you which copy of CRAN is closer

Using a package

- Is very easy:
  - library(“package”)
- Alternatively, you may use
  - require(“package”)
  - This tells R that there is no point in executing any more commands if the package is not installed.

Getting help

- Simply use
  - library(help=”package.name”)
  - help(package = “package.name”)

- When you load a package, all of its components (code, functions, datasets, **documentation**, etc...) become automatically available.

Pre-installed vs. third-party

- By default, when you install R, there is a number of packages that are automatically installed
  - They provide the basics of a fully functional statistical environment
- There are literally thousands (and many thousands indeed) of 'packages' available to extend the functionality of R

Beyond CRAN

- Not all packages are available in CRAN
- Some packages/communities have grown so big that they have acquired an identity of their own
  - Rstudio
  - JGR
  - Bioconductor
  - USGS-R, etc...
- Some of these may be accessible through other packages (e.g. Deducer contains JGR)

Do I have it?

```
# check to see if a single package is installed, if not, install it, and use it.
usePackage <- function(p) {
    if (!is.element(p, installed.packages()[,1]))     # if the package is not installed
        install.packages(p, dependencies = TRUE)      # install it
    require(p, character.only = TRUE)                 # load the package
}
# check.packages function: install and load multiple R packages.
# Check to see if packages are installed. Install them if they are not,
# then load them into the R session.
use.packages <- function(pkgs){
    new.pkgs <- pkgs[!(pkgs %in% installed.packages()[, "Package"])]
    if (length(new.pkgs))                         # if there are uninstalled packages
        install.packages(new.pkgs, dependencies = TRUE)      # install them
    sapply(pkgs, require, character.only = TRUE)    # load all the packages
}
```

%in%

- Note the use of %in% in the second function.
- This tells R to give you the values from the first collection of data that are also present in the second argument.
  - This is useful whenever we want to find common items in collections of values
  - e.g. to find common genes in two listings/groups

Example

foreign

- This package provides functionality to read “foreign” data, that is, data produced by other statistical packages.
- It is one of the packages recommended for installation with R
- `library("foreign")`
- If that doesn't work, don't worry, it is also available in CRAN

```
install.packages("foreign",
                     dependencies=TRUE)
```

foreign: Give it a try

- Remember that we can open files over the web by giving their URL instead of the file name.
- Reading SPSS data files
  - **read.spss( )** reads SPSS data files

```
dat.spss <- read.spss(

"http://faculty.washington.edu/
tathornt/sisg/hsb2.sav",

to.data.frame=TRUE)
```

Vignettes

Vignettes

- Many packages contain “vignettes” in addition to the usual textual help.
- Vignettes give an overview of the package, usually including examples and tutorials
- You can read these “vignettes” for an installed (and loaded) package using
- **BrowseVignettes("package.name")**
- If the package contains any, R will open a web browser and show them

Example

- Package **car** is another package recommended to be installed with base R. You surely have it already

```
library(car)
browseVignettes("car")
help(package="car")
```

Shortcuts

- ?name
  - help(name)
- ??name
  - Search for “name” in the available help pages.
- library()
  - With no arguments will list all packages installed in your computer and ready for loading

Masking

Name clashes, code crashes

- With so many packages, and so little imagination, we will often run into trouble.
  - Two contributors may have decided to use the same name for one of their functions
    - Possibly doing different things (or doing things differently)
  - You will see a message when this happens after loading a new package that contains a function with the same name as a function that you already have
    - In that case only the new version is visible
    - The old version is **masked**

Example

- `library("plyr")`
- `library("reshape")`
- You get a message indicating that “rename” and “round_any” have been masked.
  - This means that when we loaded package “plyr”, it added -among others- two functions with these names
  - And when we loaded “reshape” it also added two functions with the same names.

How does R decide

- Remember variable assignments
  - X = 10 ; X = 20 ; X
- Remember how did we define a function
  - func < - function(....)
- A function is just another variable. It will contain the result of the last assignment
  - Therefore, only the last definition of a function will be valid.

How can <u>you</u> decide

- That's not the whole story. Remember that the function was defined in a package...
  - Remember variables in a dataset
    - cherry$Girth
    - attach(cherry) ; Girth, cherry$Girth
    - deattach(cherry) ; Girth ; cherry$Girth
  - With packages we have something similar: elements of a package can be referred to using '**::**'
    - We can always refer to a function in a package as **package.name::function( )**

Explanation

- In effect, library works as if we loaded the package and attached it afterwards:
  - Exit R and start over again

```
rename()
?rename                # get help on "rename()"
plyr::rename()
?plyr::rename          # get help
reshape::rename()
?reshape::rename       # get help
```

- This works like $ for variables in dataframes. Note that we have <u>not</u> loaded any package with "**library()**" yet.

Explanation (2)

```
library("plyr")    # works as attach()
rename()            # rename from plyr
?rename
library("reshape")
rename()           # reshape's rename
?rename
plyr::rename()
?plyr::rename
```

- Corollary: pay attention to warnings about read masking and be careful if you need to use a masked function

Housekeeping

- We can keep track of everything that we have loaded with **sessionInfo( )**

```
sessionInfo( )
```

- The info may contain warnings about package versions being outdated. You can use **update.packages( )** to update them.
- You may use also **new.packages( )** to see what's new in CRAN.

Thank you

Questions?