R_Programming

Nicola Davide D'Avanzo

27/01/2015

This document concerns the statistical programming environment R.

Getting started

R is a dialect of S language, an internal statistical analysis environment initiated in 1976 and rewritten in C in 1988. R war created in 1991 and was made a free software through the GNU General Public License in 1995. R syntax is very similiar to S, while semantics are superficially similiar. R functionality is divided into modular packages. Graphics capabilities are sophisticated and better than other platforms. R contains a powerful prgramming language useful for developing new tools. Ther is a strong and active commmunity.

R is free. This makes you able to:

- run the program for any purpose;
- study how the program works and adapt it for your needs;
- redistribute copies;
- improve the program and release your improvements to the public.

Drawbacks of R:

- little built in support for 3-D graphics;
- functionality based on consumer demand;
- objects stored in physical memory;
- not ideal for all possible situations.

R system is divided into 2 conceptual parts:

- Base R, tha contains the Base Package (which includes the sub-packages *utils*, *stats*, *datasets*, *graphics*, *grDevices*, *grid*, *methods*, *tools*, *parallel*, *compiler*, *splines*, *tcltk*, *stats*);
- Everything else (expecially "Recommeded" packages *boot*, *class*, *cluster*, *codetools*, *foreign*, *KernSmooth*, *lattice*, *mgcv*, *nlme*, *rpart*, *survival*, *MASS*, *spatial*, *nnet*, *Matrix*).

In order to find answers on R you can try by:

- searching the archives of the forum;
- searching the Web;
- reading the manual;
- reading a FAQ;
- inspection and experimentation;
- asking a skilled friend (describe the gol not the step, be explicit about your question, provide the minimun amount of information necessary);
- reading the source code.

When having a problem do the right questions:

- what steps will reproduce the problem?

- what is the expected output?
- what do you see instead?
- what version of the product are you using?
- what operating system?
- additional information

R has five basic or "atomic" classes of objects:

- character;
- numeric (real numbers);
- integer (if you explicit want an integer, you need to specify the L suffix);
- complex;
- logical (True/False).

The most basic object is a vector. A vector can only contain objects of the same class. Empty vectors can be created with the *vector()* function. The one exception is a list, which is represented as a vector but can contain object of different classes. When different objects are mixed in a vector, coercion occurs so that every element in a vector is of the same class. Objects can be explicity coerced from one class to another using the *as.* function.

There is a special number *Inf* that represents infinity.

The missing values are denoted by *NA* and *NaN*. The value *NaN* represents an undefined number ("not a number", i.e. an undefined mathematical operation).

R objects can have attributes (accessible using the *attributes()* function):

- names, dimnames;
- dimensions (e.g. matrices, arrays);
- class;
- length;
- other metadata.

Matrices can be constructed using:

- *matrix()* function;
- from vectors by adding the dimension attribute with *dim()*;
- by column-binding or row-binding with *cbind()* and *rbind()*.

Factors are used to represent categorical data. One can think of a factor as an integer vector where each integer has a label. The order of the labels can be ordered by using the level attribute of *factor()* function. Factors are treated specially for modeling functions e.g. *lm()* and *glm()*.

Unlike matrices, data frames can store different classes of objects in each column.

R object can also have names, which is very useful for writing readable code and self-describing objects. List can also have names with *names()*, and also matrix with *dimnames()*.

Data type summary:

- atomic classes;
- vectors, lists;
- factors;
- missing values;

- data frames;
- names.

Reading data:

- *read.table*, *read.csv* for reading tabular data;
- *readLines*, for reading lines of a text file;
- *source*, for reading in R code files (inverse of *dump*);
- *dget*, for reading in R code files (inverse of *dput*);
- *load*, for reading in saved workspace;
- *unserialize*, for reading single R object in binary form.

Writing data:

- *write.table*
- *writeLines*
- *dump* (textual format, potentially recoverable because of preserving metadata)
- *dput* (textual format, potentially recoverable because of preserving metadata)
- *save*
- *serialize*

Reading in larger datasets:

Specifying the option *colClasses*, instead of using the default, can make *read.table* much faster, often twice as fast. In order to use this option, you have to know the class of each column in you data frame. So you can set the option *nrows* in order to set the number of rows and help the memory usage. You can use the Unix tool *wc* to calculate the number of lines in a file. It's useful to know a few things about your system:

- How much mermory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

In order to calculate the memory requirements of a data frame with numeric data you need to moltiplicate the number of rows and columns and 8 bytes/numeric. Then you need to divide the previous amount to 2^{30} bytes/GB.

Interfaces to the outside world:

- *file*, opens a connection to a file;
- *gzfile*, opens a connection to a file compressed in gzip;
- *bzfile*, opens a connection to a file compressed in bzip2;
- *url*, opens a connection to a webpage.

Programming with R

Control Structures:

- *if,else*: testing a condition;
- *for*: execute a loop a fixed number of times (most commonly used for iteration over the elements of an object);

- *while*: execute a loop while a condition (or more than one) is true;
- *repeat*: execute an infinite loop;
- *break*: break the execution af a loop;
- *next*: skip an iteraction of a loop;
- *return*: exit a function.

Infinite loops should be generally avoid, even if they are theoretically correct.

Functions:

- Functions are created using the *function()* directive and are stored as "first class objects", i.e. they must be trated much like any other R object.
- Functions can be passed as arguments to other functions.
- Functions can be nested, so that you can define a function inside another function.
- The return value of a function is the last expression in the function body to be evaluated.

Function arguments:

- formal arguments are those arguments included in the function definition;
- the *formals* function returns a list of all the arguments of a function;
- Function arguments can be missing or might have default values;
- R functions arguments can be matched positionally or by name.
- When an argument is matched by name, it is "taken out" of the argument list.

The order of operations when given an argument is:

- check for exact match for a named argument;
- check for a partial match;
- check for a positional match.

Lazy evaluation: arguments to functions are evaluated only lazily, so they are evaluated only as needed.

The "..." argument:

- indicate a variable number of arguments that are usually passed on to other functions;
- is often used when extending another function and you don't want to copy the entire argument list of the original function;
- is also necessary when the number of arguments passed to the function cannot be known in advance;
- any arguments that apppear after ... on the argument list must be named explicitly and cannot be partially matched.

Binding values to symbols:

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. The order is:

1. Search the global environment for a symbol name matching the one requested;
2. Search the namespaces of each of the packages of the *search()* list.

- The global environment is always the first element of the search;
- The order of the packages on the search list matters;

- When a userload a package with *library()*, the namespace of the package gets put in position 2 of the search list;
- R has separate namespace for fnctions and non-functions.

Environment:

- An environment is a collection of (symbols,value) pairs;
- Every environment ha a parent environment;
- The only environment without a parent is the empty environment;
- A function + an environment= a closure or function closure.

Scoping rules:

- The scoping rules detemine how a value is associated with a free variable in a function (free variable are not formal arguments and are not assigned insided the funciton body);
- R uses lexical scoping or static scoping. A common alternative is dynamic scoping;
- Related to the scoping rules is how R uses the search list to bind a value to a symbol.

Lexical scoping:

- If the value of the symbol is not found in the environment in which a function was defined, the search is continued in the parent environment;
- The search continues down the sequence of parent environments until we hit the top-level environment;
- After the top-level environment the search continues down the search list until we hit the empty environment.
- If a value for a given symbol cannot be found once the empty environment is arrived at, the an error is thrown.
- In R you can defined functions inside other functions: in this case the environment in which a function is defined is the body of another function;

Lexical vs Dynamic scoping:

- With lexical scoping the value of a symbol is looked up in the environment in which the function is defined;
- With dynamic scoping the value of a symbol is looked up in the environment from whic the function was called.

Coding standards for R:

- Always use text files/text editor;
- Indent your code (4 spaces at minimum, 8 spaces ideal);
- Limit the width of your code (line length fixed to 80 columns prevents lots of nesting and very long functions);
- Limit the length of individual functions.

Dates and times in R:

- Dates are respresented by the Date class and can be coerced from a character string using the *as.Date()* function;
- Times are represented by the POSIXct or POSIXlt class;

- POSIXct is just a very large integer under the hood;
- POSIXlt is a list underneath and it stores other important information like the day of the week, day of the year, month, day of the month;
- There are a number of generic functions that operate with times in R like weekdays, months, quarters.
- Times can be coerced from a character string using *as.POSIXct* or *as.POSIXlt* function;
- *strptime* function is useful when the date is written in a different format from "year-month-day hours:minuts:seconds"
- You can use mathematical operation on dates and times of the same format

Loop Functions and Debugging

There are some functions which implement loop to make life easier.

- *lapply*: loop over a list and evaluate function on each element. It takes a list and a function as inputs and always returns a list. Lapply and friends make heavy use of anonymous functions.
- *supply*: same as lapply but try to simplify the result. If the result is a list where every element is length 1, then a vector is returned; if the result is a list where every element is a vector of the same length (>1) then a matrix is returned; if it can't figure things out, a list id returned.
- *apply*: apply a function over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix.
- *tapply*: apply a function over subsets of a vector. It takes a vector and a list of factors as inputs.
- *mapply*: multivariate version of lapply. It applies a function in parallel over a set of arguments.
- *split*: take a vector or other objects and splits it into groups determined by factor or list of factors.

Indications that something is not right:

- *message*: a generic notification/diagnostic message produced by the *message* function; execution of the function continues.
- *warning*: an indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the *warning* function.
- *error*: an indication that a fatal problem has occurred; execution stops; produced by the *stop* function.
- *condition*: a generic concept for indicating that something unexpected can occur; programmers can create their own conditions.

How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were ypur expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Debugging tools in R:

- *traceback*: prints out the sequence of calls (in a function) that lead to an error; does nothing if there's no error.

- *debug*: flag a function for "debug" which allow you to step through execution of a function one line at a time.
- *browser*: suspends the execution of a function wherever it is called and puts the function in debug mode.
- *trace*: allows you to insert debugging code at chosen places in any function.
- *recover*: allows you to modify the error behavior so that you can browse the function call stack

Simulation in R

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- d for density
- r for random numbe generation
- p for cumulative distribution
- q for quantile function.
- Drawing samples from specific probability distributions can be done with r* functions
- Standard distribution are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc...
- The cumulative distribution for a standard Normal distribution is the inverse of the probability distribution.
- Always set the random number seed by *set.seed()* when conducting a simulation
- The sample function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

Profiling R code

- Profiling is a systematic way to examine how much time is spent in different parts of a program
- Useful when trying to optimize your code
- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time: this cannot be done without performance analysis or profiling.ù
- Genaral principle of optimization: design first, then optimize.

system.time()

- takes an arbitrary R expression as input and return the amount of time (in seconds) taken to evaluate the expression.
- If there is an error, gives time until the error occurred.
- Returns an object of class *proc_time* with *user time* and *elapsed time*
- *user time*: time charged to the CPU(s) for this expression
- *elapsed time*: "wall clock" time
- Usually, the *user time* and *elapsed time* are relatively close, for straight computing tasks
- *elapsed time* may be greater than *user time* if the CPU spends a lot of time waiting around
- *elapsed time* may be smaller than the *user time* if your machine has multiple cores/processors and is capable of using them

- Using *system.time()* allows you to test certain functions or code blocks to see if they are taking excessive amounts of time

R Profiler

- The *Rprof()* function starts the profiler in R
- The *summaryRprof()* function summarizes the output from *Rprof()*
- *Rprof()* keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent in each function
- The *summaryRprof()* function tabulates the R profiler output and calculates how much time is spent in which function
- There are two methods for normalizing the data: *by.total* and *by.self*
- *by.total* divides the time spent in each function by the total run time
- *by.self* does the sime but first subtracts out time spent in functions above in the call stack
- C and Fortran code called by R is not profiled