R

A Tutorial on Loops in R – Usage and Alternatives

Posted on July 23rd, 2015.

Introduction

In this easy-to-follow R tutorial on loops we will examine the constructs available in R for looping, and how to make use of R's vectorization feature to perform your looping tasks more efficiently. We will present a few looping examples; then criticize and deprecate these in favor of the most popular vectorized alternatives (amongst the very many) available in the rich set of libraries that R offers.

In general, the advice of this R tutorial on loops would be: learn about loops, because they offer a detailed view of what it is supposed to happen (what you are trying to do) at the elementary level as well as an understanding of the data you are manipulating. And then try to get rid of those whenever the effort of learning about vectorized alternatives pays in terms of efficiency.

Want to learn even more on loops? Start the DataCamp interactive R programming tutorial for free.

A Short History on the Loop

'Looping', 'cycling', 'iterating' or just replicating instructions is quite an old practice that originated well before the invention of computers. It is nothing more than automating a certain multi step process by organizing sequences of actions ('batch' processes) and grouping the parts in need of repetition. Even for 'calculating machines', as computers were called in the pre-electronics era, pioneers like **Ada Lovelace**, **Charles Babbage** and others, devised methods to **implement such iterations**.

In modern – and not so modern – programming languages, where a program is a sequence of instructions, labeled or not, the loop is an evolution of the 'jump' instruction which asks the machine to jump to a predefined label along a sequence of instructions. The beloved and nowadays deprecated ***goto*** found in Assembler, Basic, Fortran and similar programming languages of that generation, was a mean to tell the computer to jump to a specified instruction label: so, by placing that label before the location of the goto instruction, one obtained a repetition of the desired instruction a loop.

Yet, the control of the sequence of operations (the program flow control) would soon become cumbersome with the increase of goto and corresponding labels within a program. Specifically, one risked of loosing control of all the gotos that transferred control to a specific label (e.g. “from how many places did we get here”?) Then, to improve the clarity of programs, all modern programming languages were equipped with special constructs that allowed the repetition of instructions or blocks of instructions. In the present days, a number of variants exist:

- Loops that execute for a prescribed number of times, as controlled by a counter or an index, incremented at each iteration cycle; these pertain to the ***for*** family;
- Loops based on the onset and verification of a logical condition (for example, the value of a control variable), tested at the start or at the end of the loop construct; these variants belong to the ***while*** or ***repeat*** family of loops, respectively.

Using Loops (in R)

Every time some operation/s has to be repeated, a loop may come in handy. We only need to specify how many times or upon which conditions those operations need execution: we assign initial values to a control loop variable, perform the loop and then, once finished, we typically do something with the results.

But when are we supposed to use loops? Couldn’t we just replicate the desired instruction for the sufficient number of times?

Well, our personal and arbitrary rule of thumb is that if you need to perform an action

(say) three times or more, then a loop would serve you better; it makes the code more compact, readable and maintainable and you may save some typing. Say you discover that a certain instruction needs to be repeated once more than initially foreseen:

: of a variable in the

And yet, the peculiar nature of R suggests **not to use loops at all(!)**: that is, whenever alternatives exist. And surely there are some, because R enjoys a feature that few programming language do, which is called **vectorization**...

Loops in the R language

According to the R base manual, among the control flow commands (e.g. the commands that enable a program to branch between alternatives, or, so to speak, to 'take decisions') the loop constructs are for, while and repeat, with the additional clauses break and next. You may see these by invoking `?Control` at the R Studio command line. A flow chart format of these structures is shown in figure 1.

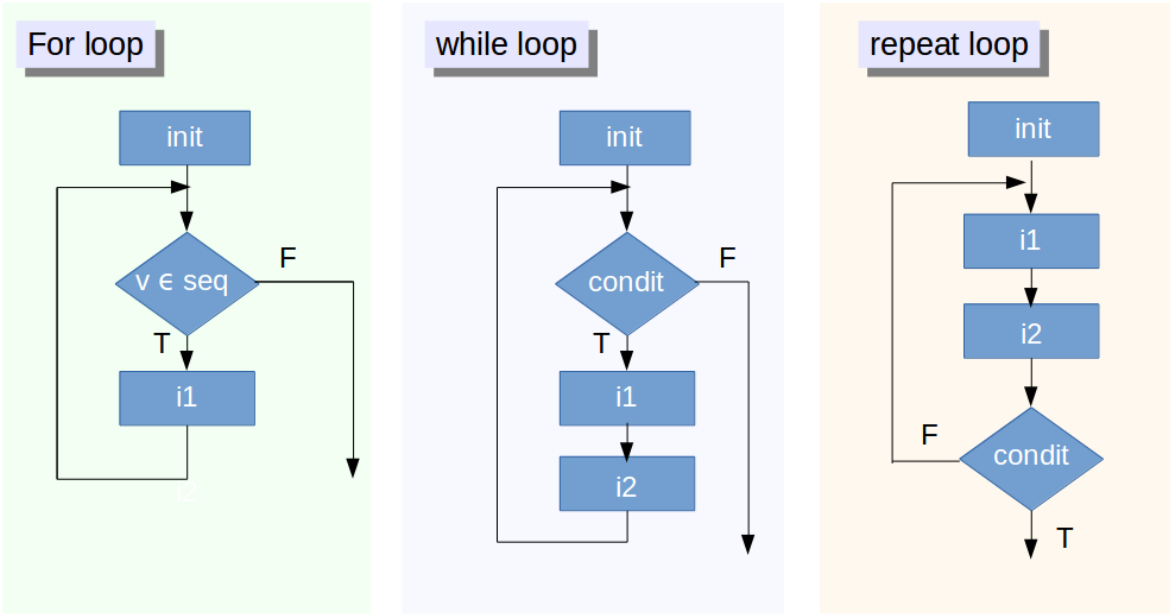

Figure 1

Let's have a look at each of these, starting from left to right.

The For loop in R

General

This loop structure, made of the rectangular box 'init' (for initialization), the diamond or rhombus decision, and the rectangular box (i1 in the figure), is executed a known number of times. In flow chart terms, rectangular boxes mean something like do something which does not imply decisions; while rhombi or diamonds are called 'decision' symbols and translate into questions which only have two possible logical answers, True (T) or False (F).

One or more initialization instructions within the init rectangle are followed by the evaluation of the condition on a variable which can assume values within a specified sequence. In the diamond within the figure, the symbols mean: Does the variable v's

current value belong to the sequence seq? Thus we are testing whether v's current value is within a specified range. This range is typically defined in the initialization, with something like (say) 1:100 in order to ensure that the loop is started.

If the condition is not met (logical false, F in the figure) the loop is never executed, as indicated by the loose arrow on the right of the for loop structure. The program will then execute the first instruction found after the loop block.

If the condition is verified, a certain instruction – or block of instructions- i1 is executed. Perhaps this is even another loop (a construct known as a nested loop). Once this is done, the condition is evaluated again, as indicated by the lines going from i1 back to the top, immediately after the init box. In R (and in Python) it is possible to express this in almost plain English, by asking whether our variable belongs to a range of values, which is handy. We note that in other languages, for example in C, the condition is made more explicit, by basing on a logical operator (greater or less than, equal to etc).

Here is an example of a simple for loop (a bit artificial, but it serves for illustration purposes).

```
u1 <- rnorm(30) # create a vector filled with random normal values
print("This loop calculates the square of the first 10 elements of
vector u1")

usq<-0
for(i in 1:10)
{
  usq[i]<-u1[i]*u1[i] # i-th element of u1 squared into i-th position
of usq
```

```
of usq
  print(usq[i])
}
print(i)
```

So, the for block is contained within curly braces. These may be placed either immediately after the test condition or beneath it, preferably followed by an indentation. None of this is compulsory, but it enhances readability and allows to easily spot the block of the loop and potential errors within.

[Note the initialization of the vector of the squares, usq. In effect, this would not be necessary in plain RStudio code, but in the markup version, knitr would not compile because a reference to the vector is not found before its use in the loop, thus throwing an error within RStudio].

Nesting For loops

For loops may be nested, but when and why would we be using this? Suppose we wish to manipulate a bi-dimensional array by setting its elements to specific values; we might do something like this:

```
# nested for: multiplication table
mymat = matrix(nrow=30, ncol=30) # create a 30 x 30 matrix (of 30 rows
and 30 columns)

for(i in 1:dim(mymat)[1])  # for each row
{

  for(i in 1:dim(mymat)[2]) # for each column
```

```
  for(j in 1:dim(mymat)[2]) # for each column
  {
    mymat[i,j] = i*j      # assign values based on position: product of
two indexes
  }
}
```

Two nested for, thus two sets of curly braces, each with its own block, governed by its own index (i runs over the lines, j over the columns). What did we produce? Well we made the all too familiar multiplication table that you should know by heart (here we limited to the first 30 integers):

```
mymat[1:10,1:10] # show just the upper left 10x10 chunk
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    4    6    8   10   12   14   16   18    20
##  [3,]    3    6    9   12   15   18   21   24   27    30
##  [4,]    4    8   12   16   20   24   28   32   36    40
##  [5,]    5   10   15   20   25   30   35   40   45    50
##  [6,]    6   12   18   24   30   36   42   48   54    60
##  [7,]    7   14   21   28   35   42   49   56   63    70
##  [8,]    8   16   24   32   40   48   56   64   72    80
##  [9,]    9   18   27   36   45   54   63   72   81    90
## [10,]   10   20   30   40   50   60   70   80   90   100
```

We may even produce a sequence of multiplication tables with a variable and

increasing third index in an additional outer loop (try this as an exercise); or let the user

increasing third index in an additional outer loop (try this as an exercise), or let the user choose an integer and then produce a table according to her choice : you need to acquire an integer and then assign it to one variable (if the table is square) or two (if the table is rectangular), which serve as upper bounds to the indexes i and j.

We show it for the square case, by preceding the previous loop with the user defined function (or udf) readinteger:

```
readinteger <- function()
{
  n <- readline(prompt="Please, enter an integer: ")
}
nr<-as.integer(readinteger())
```

Before setting the integer as upper bound for both the for loops, note that because the value entered is used inside the udf, we ask the latter to return its value in a different variable, nr, that can be referenced anywhere in the code (try using the variable 'n' outside of the function: RStudio will complain that no such thing is defined. This is an example of visibility (scope) of variables. (More on this, perhaps in a forthcoming post).

Also, note that to prevent the user from deluging the screen with huge tables, we put a condition at the end, in order to print either the first 10 x 10 chunk, if the user asked for an integer greater than 10; or (else), an n x n chunk, if the value entered was less or equal than 10.

The complete code looks like:

```
readinteger <- function()
```

```
readinteger <- function()
{
  n <- readline(prompt="Please, enter an integer: ")
  # return(as.integer(n))
}
nr<-as.integer(readinteger())
mymat = matrix(0,nr,nr) # create a n x n matrix with zeroes (n rows
and n columns)

for(i in 1:dim(mymat)[1])  # for each row
{
  for(j in 1:dim(mymat)[2]) # for each column
  {
    mymat[i,j] = i*j     # assign values based on position: product of
two indexes
  }
}
if (nr>10)
{
  mymat[1:10,1:10]  # for space reasons, we just show the first 10x10
chunk
} else mymat[1:nr,1:nr]  # ...or the first nr x nr chunk
```

Conclusion for loop

The for loop is by far the most popular and its construct implies that the number of iterations is fixed and known in advance,. E.g. generate the first 200 prime numbers or, enlist the 10 most important customers and things the like.

But, what if we do not know or control the number of iterations and one or several

conditions may occur which are not predictable beforehand? For example, we may want to count the number of clients living in an area identified by a specified post-code, or the number of clicks on a web page banner within the last two days, or similar unforeseen events. In cases like these, the while loop (and its cousin repeat) may come to the rescue.

While loop in R

The while loop, in the midst of figure 1, is made of an init block as before, followed by a logical condition which is typically expressed by the comparison between a control variable and a value, by means of greater/less than or equal to, although any expression which evaluates to a logical value, T or F is perfectly legitimate.

If the result is false (F), the loop is never executed as indicated by the loose arrow on the right of the figure. The program will then execute the first instruction it finds after the loop block.

If it is true (T) the instruction or block of instructions i1 is executed next. We note here that an additional instruction (or block) i2 has been added: this serves as an update for the control variable, which may alter the result of the condition at the start of the loop (although this is not necessary); or perhaps an increment to a counter, used to keep trace of the number of iterations executed. The iterations cease once the condition evaluates to false.

The format is `while(cond) expr`, where cond is the condition to test and expr is an

expression (possibly a statement). For example, the following loop asks the user to

enter the correct answer to **the universe and everything question** (from the hitchhiker's guide to the galaxy), and will continue to do so until the user gets it right:

```
readinteger <- function()
{
  n <- readline(prompt="Please, enter your ANSWER: ")
}
response<-as.integer(readinteger())

while (response!=42)
{
  print("Sorry, the answer to whaterver the question MUST be 42");
  response<-as.integer(readinteger());
}
```

As a start, we use the same function employed in the for loop above to get the user input, before entering the loop as long as the answer is not the expected '42'. This is why we use the input once, before the loop. Firstly because if we did not, R would complain about the lack of an expression providing the required True/False (in fact, it does not know 'response' before using it in the loop). Secondly, because if we answer right at first attempt, the loop will not be executed at all.

Repeat loop in R

The repeat loop at the far right of the picture in figure 1 is similar to the while, but it is made so that the blocks of instructions i1 and i2 are executed at least once, no matter

what the result of the condition, which in fact, is placed at the end. Adhering to other

languages, one could call this loop ‘repeat until’, in order to emphasize the fact that the instructions i1 and i2 are executed until the condition remains false (F) or, equivalently, becomes true (T), thus exiting; but in any case, at least once.

As a variation of the previous example, we may write:

```
readinteger <- function()
{
  n <- readline(prompt="Please, enter your ANSWER: ")
}

repeat
{
  response<-as.integer(readinteger());
  if (response==42)
  {
    print("Well done!");
    break
  } else print("Sorry, the answer to whaterver the question MUST be
42");

}
```

After the now familiar input function, we have the repeat loop whose block is executed at least once and that will terminate whenever the if condition is verified.
We note here that we had to set a condition within the loop upon which to exit with the clause break. This introduces us to the notion of exiting or interrupting cycles within loops.

Interruption and exit from loops in R

So how do we exit from a loop? In other terms, aside of the 'natural' end of the loop, which occurs either because we reached the prescribed number of iterations (for) or we met a condition (while, repeat), can we stop or interrupt the loop and if yes, how?

The break statement responds to the first question: we have seen this in the last example.

Break

When the R interpreter encounters a break, it will pass control to the instruction immediately after the end of the loop (if any). In case of nested loops, the break will permit to exit only from the innermost loop.

Here's an example. This chunk of code defines an m x n matrix of zeros and then enters a nested-for loop to fill the locations of the matrix, only if the two indexes differ. The purpose is to create a lower triangular matrix, that is a matrix whose elements below the main diagonal are non zero, the others are left untouched to their initialized zero value. When the indexes are equal (if condition in the inner loop, which runs over j, the column index), a break is executed and the innermost loop is interruped with a direct jump to the instruction following the inner loop, which is a print; then control gets to the outer for condition (over the rows, index i), which is evaluated again. If the indexes differ, the assignment is performed and the counter is incremented by 1. At the end, the program prints the counter ctr, which contains the number of elements that were assigned

```
# make a lower triangular matrix (zeroes in upper right corner)
m=10; n=10;
ctr=0;   # used to count the assignemnt
mymat = matrix(0,m,n) # create a 10 x 10 matrix with zeroes
for(i in 1:m) {
  for(j in 1:n)
  {
    if(i==j)
    {
      break;
    } else
      {
      mymat[i,j] = i*j   # we assign the values only when i<>j
      ctr=ctr+1
      }
  }
  print(i*j)
}
print(ctr)  # print how many matrix cells were assigned
```

[Note that we are a bit over-cautious in putting curly brackets even when non strictly necessary, just to make sure that whatever is opened with a {, is also closed with a }. So if you notice unmatched numbers of { or }, you know there is an error, although the opposite is not necessarily true!].

Next

Next discontinues a particular iteration and jumps to the next cycle (in fact, to the

evaluation of the condition holding the current loop). In other languages we may find the (slightly confusing) equivalent called continue, which means the same: wherever you are, upon the verification of the condition, jump to the evaluation of the loop you are in. A simpler example of keeping the loop ongoing whilst discarding a particular cycle upon the occurrence of a certain condition is:

```
m=20;
for (k in 1:m)
{
  if (!k %% 2)
    next
    print(k)
}
```

The effect here is to print all uneven numbers within the interval 1:m (here m=20) , that is, all integers except the ones with non zero remainder when divided by 2 (thus the use of the modulus operand %%), as specified by the if test. Even numbers, whose remainder is zero, are not printed, as the program jumps to the evaluation of the i in 1:m condition, thus ignoring the instruction that follows, in this case the print(k).

Wrap-up: The use of loops in R

1. Try to put as little code as possible within the loop by taking out as many instructions as possible (remember, anything inside the loop will be repeated several times and perhaps it is not needed).

2. Careful when using repeat: ensure that a termination is explicitly set by testing

a condition, or an infinite loop may occur.

3. If a loop is getting (too) big, it is better to use one or more function calls within the loop; this will make the code easier to follow. But the use of a nested for loop to perform matrix or array operations is probably a sign that things are not implemented the best way for a matrix based language like R.
4. Growing' of a variable or dataset by using an assignment on every iteration is not recommended (in some languages like Matlab, a warning error is issued: you may continue but you are invited to consider alternatives). A typical example is shown in the next section.
5. If you find out that a vectorization option exists, don't use the loop as such, learn the vectorized version instead.

Vectorization

Simply put: 'Vectorization' is the operation of converting repeated operations on simple numbers ('scalars'), into single operations on vectors or matrices. A vector is the elementary data structure in R and is "a single entity consisting of a collection of things (from the R base manual). So, a collection of numbers is a numeric vector. If we combine vectors (of the same length) we obtain a matrix. We can do this vertically or horizontally, using different R instructions. Thus in R a matrix is seen as a collection of horizontal or vertical vectors. By extension, we can vectorize repeated operations on vectors.

Now then, many of the above loop constructs can be made implicit by using vectorization. 'Implicit', because they do not really disappear; at a lower level, the

alternative vectorized form translates into code which will contain one or more loops in the lower level language the form was implemented and compiled (Fortran, C, or C++). These (hidden to the user) are usually faster than the equivalent explicit R code, but unless you plan implementing your own R functions using one of those languages, this is totally transparent to you.

The most elementary case one can think of, is the addition of two vectors v1 and v2 into a vector v3, which can be done either element-by-element with a for loop

```
for (i in 1:n)
{
    v3[i] <-v1[i] + v2[i]
}
```

or using the 'native' vectorized form:

```
v3 = v1 + v2
```

['native' because R can recognize all the arithmetic operators as acting on vectors and matrices].

Similarly, for two matrices A and B, instead of adding the elements of A[i,j] and B[i,j] in corresponding positions, for which we need to take care of two indexes i and j, we tell R to do the following:

```
C= A + B
```

and this is very simple indeed!

Vectorization Explained

Why would vectorization run faster, given that the number of elementary operations is seemingly the same? This is best explained by looking at the internal nuts and bolts of R, which would demand a separate post, but succinctly:

In the first place, R is an ***interpreted language*** and as such, all the details about variable definition (type, size, etc) are taken care by the interpreter. You do not need to specify that a number is a floating point or allocate memory using (say) a pointer in memory. The R interpreter 'understands' these issues from the context as you enter your commands, but it does so on a command-by-command basis. Therefore, it will need to deal with such definitions (type, structure, allocation etc) every time you issue a given command, even if you just repeat it.

A compiler instead, solves literally all the definitions and declarations at compilation time over the ***entire code***; the latter is translated into a compact and optimized binary code, before you try to execute anything. Now, as R functions are written in one of these lower-level languages, they are more efficient (in practice, if one looked at the low level code, one would discover calls to C or C++, usually implemented within what is called a wrapper code).

Secondly, in languages supporting vectorization (like R or Matlab) every instruction

making use of a numeric datum, acts on an object which is natively defined as a vector, even if only made of one element. This is the default when you define (say) a single

even if only made of one element. This is the default when you define (say) a single numeric variable: its inner representation in R will always be a vector, albeit made of one number only. Under the hood, loops continue to exist, but at the lower and much faster C/C++ compiled level. The advantage of having a vector means that the definitions are solved by the interpreter only once, on the entire vector, irrespective of its size (number of elements), in contrast to a loop performed on a scalar, where definitions, allocations, etc, need to be done on a element by element basis, and this is slower.

Finally, dealing with native vector format allows to utilize very efficient Linear Algebra routines (like BLAS), so that when executing vectorized instructions, R leverages on these efficient numerical routines. So the message would be, if possible, process whole data structures within a single function call in order to avoid all the copying operations that are executed.

But enough with digressions, let's make a general example of vectorization, and then in the next subsection we'll delve into more specific and popular R vectorized functions to substitute loops.

Vectorization, an example

Let's go back to the notion of 'growing data', a typically inefficient way of 'updating' a dataframe. First we create an m x n matrix with `replicate(m, rnorm(n))` with m=10 column vectors of n=10 elements each, constructed with rnorm(n), which creates random normal numbers. Then we transform it into a dataframe (thus 10 observations of 10 variables) and perform an algebraic operation on each element using a nested for loop: at each iteration, every element referred by the two indexes is incremented by a

loop: at each iteration, every element referred by the two indexes is incremented by a sinusoidal function [The example is a bit artificial, but it could represent the addition of a signal to some random noise]:

```
########## a bad loop, with 'growing' data
set.seed(42);
m=10; n=10;
mymat<-replicate(m, rnorm(n)) # create matrix of normal random numbers
mydframe=data.frame(mymat)    # transform into data frame
for (i in 1:m) {
  for (j in 1:n) {
    mydframe[i,j]<-mydframe[i,j] + 10*sin(0.75*pi)
    print(mydframe)
  }
}
```

Here, most of the execution time consists of copying and managing the loop. Let's see how a vectorized solution looks like:

```
#### vectorized version
set.seed(42);
m=10; n=10;
mymat<-replicate(m, rnorm(n))
mydframe=data.frame(mymat)
mydframe<-mydframe + 10*sin(0.75*pi)
```

Which looks simpler: the last line takes the place of the nested for loop (note the use of the set.seed to ensure the two implementations give exactly the same result).

Let's now quantify the execution time for the two solutions. We can do this by using R system commands, like system.time() to which a chunk of code can be passed like this (just put the code you want to evaluate in between the parentheses of the system.time function):

```
# measure loop execution
system.time(
  for (i in 1:m) {
    for (j in 1:n) {
      mydframe[i,j]<-mydframe[i,j] + 10*sin(0.75*pi)
    }
  }
)
```

```
##    user  system elapsed
##   0.011   0.006   0.023
```

```
# measure vectorized execution
system.time(
  mydframe<-mydframe + 10*sin(0.75*pi)
)
```

```
##    user  system elapsed
##   0.003   0.001   0.003
```

In the code chunk above, we do the job of choosing m and n, the matrix creation and its transformation into a dataframe only once at the start, and then evaluate the 'for'

chunk against the 'one-liner' of the vectorized version with the two separate call to system.time. We see that already with a minimal setting of m=n=10 the vectorized version is 7 time faster, although for such low values, it is barely important for the user. Differences become noticeable (at the human scale) if we put m=n=100, whereas increasing to 1000 causes the for loop look like hanging for several tens of seconds, whereas the vectorized form still performs in a blink of an eye.

For m=n=10000 the for loop hangs for more than a minute while the vectorized requires 2.54 sec. Of course, these measures should be taken ***cum grano salis*** and will depend on the hardware and software configuration, possibly avoiding overloading your laptop with a few dozens of open tabs in your internet browser, and several applications running in the background; but are illustrative of the differences [In fairness, the increase of m and n severely affects also the matrix generation as you can easily see by placing another system.time() call around the replication function].

You are invited to play around with m and n to see how the execution time changes, by plotting the execution time as a function of the product m x n (which is the relevant indicator, because it expresses the dimension of the matrices created and thus also quantifies the number of iterations necessary to complete the task via the nseted for loop).
So this is an example of vectorization. But there are many others. In **R News, a newsletter for the R project** at p46, there are very efficient functions for calculating sums and means for certain dimensions in arrays or matrices, like: rowSums(),

colSums(), rowMeans(), and colMeans().

Furthermore, ...the functions in the apply family, named [s,l,m,t]apply, are provided to apply another function to the elements/dimensions of objects. These apply functions provide a compact syntax for sometimes rather complex tasks that is more readable and faster than poorly written loops.

The apply family: just hidden loops?

The very rich and powerful family of apply functions is made of intrinsically vectorized functions. If at first sight these do not appear to contain any loop, looking carefully under the hood this feature becomes manifest.
The apply command or rather family of commands, pertains to the R base package and is populated with a number of functions (the [s,l,m,r, t,v]apply) to manipulate slices of data in the form of matrices or arrays in a repetitive way, allowing to cross or traverse the data and avoiding explicit use of loop constructs. The functions act on an input matrix or array and apply a chosen named function with one or several optional arguments (that's why they pertain to the so called 'functionals' (as in Hadley Wickham's advanced R page. The called function could be an aggregating function, like a simple mean, or another transforming or sub-setting function.

It should be noted that the application of these functions does not necessarily lead to faster execution (differences are not huge); rather it avoids the coding of cumbersome loops, reducing the chance of errors.

The functions within the family are: apply, sapply, lapply, mapply, rapply, tapply, vapply

But when and how should we use these? Well, it is worth noting that a package like plyr covers the functionality of the family; although remembering them all without resorting

to the official documentation might feel difficult, still these functions form the basic of more complex and useful combinations.

The first three are the most frequently used.

1. apply

We wish to apply a given function to the rows (index 1) or columns (index 2) of a ***matrix***.

2. lapply

We wish to apply a given function to ***every element of a list and obtain a list*** as result (thus the l in the function name).

3. sapply

We wish to apply a given function to ***every element of a list but we wish to obtain a vector*** rather than a list.

Related functions are sweep(), by() and aggregate() and are occasionally used in conjunction with the elements of the apply family.
We limit the discussion here to the apply function (a more extended discussion on this topic might be the focus of a future post).

Given a Matrix M, the call:

`apply(M,1,fun)` or `apply(M, 2,fun)` will apply the specified function fun to the rows of M, if 1 is specified; or to the columns of M, when 2, is specified. This numeric argument is called 'margin', and is limited to the values 1 and 2 because the function operates on a matrix (but we could have an array instead, with up to 8 dimensions). The

function may be any valid R function, but it could be a user defined function, even coded inside the apply, which is handy.

Apply: an example

```
#### elementary example of apply function

# define matrix mymat by replicating the sequence 1:5 for 4 times and
transforming into a matrix
mymat<-matrix(rep(seq(5), 4), ncol = 5)

# mymat sum on rows
apply(mymat, 1, sum)

## [1] 15 15 15 15

# mymat sum on columns
apply(mymat, 2, sum)

## [1] 10 11 12 13 14

# with user defined function within the apply
# that adds any number y to the sum of the row
# here chosen as 4.5
apply(mymat, 1, function(x, y) sum(x) + y, y=4.5)

## [1] 19.5 19.5 19.5 19.5

# or produce a summary column wise (for each column)
apply(mymat, 2, function(x, y) summary(mymat))
```

```
##       [,1]             [,2]             [,3]             [,4]
##  [1,] "Min.   :1.00  " "Min.   :1.00  " "Min.   :1.00  " "Min.
:1.00  "
##  [2,] "1st Qu.:1.75  " "1st Qu.:1.75  " "1st Qu.:1.75  " "1st
Qu.:1.75  "
##  [3,] "Median :2.50  " "Median :2.50  " "Median :2.50  " "Median
:2.50  "
##  [4,] "Mean   :2.50  " "Mean   :2.50  " "Mean   :2.50  " "Mean
:2.50  "
##  [5,] "3rd Qu.:3.25  " "3rd Qu.:3.25  " "3rd Qu.:3.25  " "3rd
Qu.:3.25  "
##  [6,] "Max.   :4.00  " "Max.   :4.00  " "Max.   :4.00  " "Max.
:4.00  "
##  [7,] "Min.   :1.00  " "Min.   :1.00  " "Min.   :1.00  " "Min.
:1.00  "
##  [8,] "1st Qu.:1.75  " "1st Qu.:1.75  " "1st Qu.:1.75  " "1st
Qu.:1.75  "
##  [9,] "Median :2.50  " "Median :2.50  " "Median :2.50  " "Median
:2.50  "
## [10,] "Mean   :2.75  " "Mean   :2.75  " "Mean   :2.75  " "Mean
:2.75  "
## [11,] "3rd Qu.:3.50  " "3rd Qu.:3.50  " "3rd Qu.:3.50  " "3rd
Qu.:3.50  "
## [12,] "Max.   :5.00  " "Max.   :5.00  " "Max.   :5.00  " "Max.
:5.00  "
## [13,] "Min.   :1.00  " "Min.   :1.00  " "Min.   :1.00  " "Min.
:1.00  "
## [14,] "1st Qu.:1.75  " "1st Qu.:1.75  " "1st Qu.:1.75  " "1st
Qu.:1.75  "
```

```
Qu.:1.75
## [15,] "Median :3.00  " "Median :3.00  " "Median :3.00  " "Median
:3.00  "
## [16,] "Mean   :3.00  " "Mean   :3.00  " "Mean   :3.00  " "Mean
:3.00  "
## [17,] "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd
Qu.:4.25  "
## [18,] "Max.   :5.00  " "Max.   :5.00  " "Max.   :5.00  " "Max.
:5.00  "
## [19,] "Min.   :1.00  " "Min.   :1.00  " "Min.   :1.00  " "Min.
:1.00  "
## [20,] "1st Qu.:2.50  " "1st Qu.:2.50  " "1st Qu.:2.50  " "1st
Qu.:2.50  "
## [21,] "Median :3.50  " "Median :3.50  " "Median :3.50  " "Median
:3.50  "
## [22,] "Mean   :3.25  " "Mean   :3.25  " "Mean   :3.25  " "Mean
:3.25  "
## [23,] "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd
Qu.:4.25  "
## [24,] "Max.   :5.00  " "Max.   :5.00  " "Max.   :5.00  " "Max.
:5.00  "
## [25,] "Min.   :2.00  " "Min.   :2.00  " "Min.   :2.00  " "Min.
:2.00  "
## [26,] "1st Qu.:2.75  " "1st Qu.:2.75  " "1st Qu.:2.75  " "1st
Qu.:2.75  "
## [27,] "Median :3.50  " "Median :3.50  " "Median :3.50  " "Median
:3.50  "
## [28,] "Mean   :3.50  " "Mean   :3.50  " "Mean   :3.50  " "Mean
:3.50  "
## [29,] "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd Qu.:4.25  " "3rd
Qu.:4.25  "
```

```
## [30,] "Max.   :5.00  " "Max.   :5.00  " "Max.   :5.00  " "Max.
:5.00  "
##       [,5]
##  [1,] "Min.   :1.00  "
##  [2,] "1st Qu.:1.75  "
##  [3,] "Median :2.50  "
##  [4,] "Mean   :2.50  "
##  [5,] "3rd Qu.:3.25  "
##  [6,] "Max.   :4.00  "
##  [7,] "Min.   :1.00  "
##  [8,] "1st Qu.:1.75  "
##  [9,] "Median :2.50  "
## [10,] "Mean   :2.75  "
## [11,] "3rd Qu.:3.50  "
## [12,] "Max.   :5.00  "
## [13,] "Min.   :1.00  "
## [14,] "1st Qu.:1.75  "
## [15,] "Median :3.00  "
## [16,] "Mean   :3.00  "
## [17,] "3rd Qu.:4.25  "
## [18,] "Max.   :5.00  "
## [19,] "Min.   :1.00  "
## [20,] "1st Qu.:2.50  "
## [21,] "Median :3.50  "
## [22,] "Mean   :3.25  "
## [23,] "3rd Qu.:4.25  "
## [24,] "Max.   :5.00  "
## [25,] "Min.   :2.00  "
## [26,] "1st Qu.:2.75  "
## [27,] "Median :3.50  "
```

```
## [28,] "Mean   :3.50  "
## [29,] "3rd Qu.:4.25  "
## [30,] "Max.   :5.00  "
```

Often we use dataframes: in this case, we must ensure that the data have the same type or else, forced data type conversions may occur, which probably is not what you want (For example, in a mixed text and number dataframe, numeric data will be converted to strings or characters).

Summary and conclusions

So now, this journey brought us from the fundamental loop constructs used in programming to the (basic) notion of vectorization and to an example of the use of one of the apply family of functions, which come up frequently in R. In terms of code flow control, we dealt only with iterative constructs ('loops'): for, while, repeat and the way to interrupt and exit these. As the last subsections hint that loops in R should be avoided, you may ask why on earth should we learn about them. Now, in our opinion, you ought to learn these programming structures because:

1. It is likely that R will not be your only language in data science or elsewhere, and grasping general constructs like loops is a useful thing to put in your own skills bag. Their syntax may vary depending on the language, but once you master those in one, you'll readily apply them to any other language you come across.

2. The R universe is huge and it is very difficult if not impossible to be wary of all R existing functions. There are many ways to do things, some more efficient or

elegant than others and your learning curve will be incremental; upon use, in time you will start asking yourself about more efficient ways to do things and will eventually land on functions that you never heard of before. By the time you read this, a few new libraries will be developed, others released, and so the chance of knowing them all is slim, unless you spend your entire time as an R specialist.

3. Finally, at least when not dealing with particularly highly dimensional datasets, a loop solution would be easy to code and read. And perhaps, as a data scientist, you may be asked to prototype a one-off job that just works. Perhaps you are not interested in sophisticated or elegant ways of getting a result. Perhaps, in the data analysis work-flow, you just need to show domain expertise and concentrate on the content. Somebody else, most likely a back-end specialist, will take care of the production phase (and perhaps she might be coding it in a different language!)

About the author: Carlo Fanara

Carlo Fanara – After a career in IT, Carlo spent 20 years in physics, first gaining an Msc in Nuclear Physics in Turin (Italy), and then a PhD in plasma physics in the United Kingdom. After several years in academia, in 2008 Carlo moved to France to work on R&D projects in private companies and more recently as a freelance. His interests include technological innovation and programming for Data Mining and Data Science.

View Comments (0) ...

DataCamp posts can be found at **R-bloggers.com**