
Advanced Data Management in R

MATTHEW DENNY

FRIDAY 15TH NOVEMBER, 2013

This tutorial will follow the `example.R` provided in the `Advanced_Data_Management_in_R.zip` file included with this tutorial. All the data you need to follow the example script is included in the folder. You will simply need to change your working directory to the unzipped workshop folder you downloaded with this document in it.

1 Tips and Tricks

One suggestion I can make if you are starting to get into more advanced data management projects in R is that you use an integrated development environment (IDE) like **RStudio** (available here: <http://www.rstudio.com/>) for testing and writing your code, and then run the full production version of your work in a base R window. The advantages of this strategy are listed below:

1. Base R uses less Ram as overhead than and IDE like **RStudio** so if you find that memory becomes an issue on your computer, this may speed things up significantly and allow you to work with more data at a time on the same computer.
2. Being able to easily inspect the data objects you are creating is a really helpful feature of the work space window in **RStudio**. I suggest using this for testing your code as I have found tons of errors in my code by simply inspecting the data object it outputs for the right dimensions and type.
3. Running your code fresh from the start once you have written and tested it will help ensure that if you make it available to others as part of the supporting materials to a published paper, it will be more likely to run on a reviewer or colleague's computer.

You will also want to be sure that you keep detailed notes on the version of R and of any packages you are using in your analysis. The people who maintain and update these packages will not always think of how and update could affect the particular analysis you are running so it is helpful to include this information in a `README.txt` file for future users or for yourself if you come back to the code, so that you can replicate your results. I know several people who have experienced getting different results due to a change in the functionality of a package so just take the 5 minutes to jot down the version number of each package you use.

1.1 The `paste()` function

For a lot of what I am doing in this tutorial and what I do in the most complicated data management projects I work on, this single function is the most important and useful I have encountered. This function lets you glue together strings and pieces of text which is really useful for giving yourself print statement updates on where you are in a loop. For example:

```
print(paste("Currently on Iteration: ", i , sep = ""))
```

will tell you what iteration you have reached in a loop and is a good way to check your progression a very long or nested looping job. You can also use the `paste` function inside of the `assign()` function or the `get()` function to automate the process of creating R objects and accessing them. For example you may want to read in a whole bunch of `.csv` files as we do in our example. Then we can use a combination of the `paste` and `assign` functions to create sequentially named R objects for us.

```
temp <- read.csv(paste(cur,"_senmatrix.txt", sep = ""), stringsAsFactors = F, header = F)
assign(paste("Senate_Raw_Cosponsorship_",cur, sep = ""),temp)
```

On the flip side of the equation, we can also use the `get()` function inside of a loop to sequentially work on one part of our data at a time.

```
temp <- get(paste("Senate_Raw_Cosponsorship_",cur, sep = ""))
```

These functions working together form the heart of a lot of complicated data analysis projects. However if you have an extremely large data analysis project, you may want to use a list data structure.

1.2 Using lists

You can create a list object by using the `list()` function. Lists are useful because they can contain all sorts of objects of different types and can be easily added to using the `append()` function. It is sometimes useful to create lists of lists where the outer list indexes a year and the inner list indexes the objects that are relevant to that year. In the example below, I use the `append` function to simply add everything to one big outer list that can be accessed using the dollar sign operator.

```
Current_List <- list(temp_cosponsorship, temp_senator_attributes)
names(Current_List) <- c(paste("Cosponsorship_",cur,sep=""),
                        paste("Senator_Attributes_",cur,sep=""))
Master_List <- append(Master_List, Current_List)
```

Now we can access things off of the list by using one of two methods. We can either use the list subscripts or we can use the `$` operator:

```
Master_List$Cosponsorship_100 (gives us a cosponsorship matrix)
Master_List[[2]]$age (we can also use them together)
```

The point is that when you are working with hundreds of different objects it may make the most sense to just put them all in a list. These list objects can simplify the process of indexing objects so that we can loop over waves of a survey or years of data collection parsimoniously.

2 User Defined Functions

There are several advantages to defining your own functions:

1. They make it possible to share your techniques with coauthors or other scholars who may not be as familiar with your methods or with R coding as you are.
2. They can form the basis of an R package which is increasingly becoming professionally valuable.
3. Writing a function challenges you to think more abstractly about what you want to do and often can be used for more than its intended purpose.
4. Using functions is faster than rewriting or changing the text throughout a complicated nested loop or other process and will greatly speed up your analysis as it grows more complex.
5. Functions are necessary for use in apply statements and in parallel processing which can provide massive speed-ups to your work flow.

Essentially what you are doing by defining your own function is to hide away the mechanics of the operation you wish to complete and just get the result by typing in a simple and flexible command. A function takes the following form:

```
add <- function(agr1, arg2){
  result <- arg1 + arg2 #as an example
  return(result)
}
```

You give the function some data or arguments that you want it to do something with and then you tell it to return a single value which can be assigned to a variable. Alternatively, you do not need to actually have the function return anything, instead it can just print something or save something such as:

```
add <- function(agr1, arg2){
  result <- arg1 + arg2 #as an example
  print(result)
}
```

If you want to return more than one object from your function you have to put those objects in a list first. Otherwise you should get an error message saying that functions are not allowed to return mutiple arguments such as if we run the following function:

```
add <- function(agr1, arg2){
  result <- arg1 + arg2 #as an example
  return(result, arg1, arg2)
}
```

note that if you enter this function to the R console everything will seem to be fine, but if you try to run it you will get an error like this:

```
Error in return(result, arg1, arg2) : multi-argument returns are not permitted
```

2.1 Using the `source()` command and keeping things organized

Another really cool thing about functions is that you can call them using the `source` function, allowing you to keep your user defined functions in a separate file and then load them into your working memory with one simple command. So long as the R file containing the code that defines the function is in your current working directory, you can use the `source` command as follows:

```
source("My_Functions.R")
```

and this command will load all of the functions defined in this file into your working memory. Note that you can define as many functions as you want in the same file, although there are a few guidelines I suggest you stick to:

1. If you have a really big function with a bunch of lines of code in it that does something general, I suggest you give it its own R file and name it roughly the same as the function.
2. If you have several related smaller functions, putting them all in the same file with a commented index at the top that states what is in the file can make a lot of sense.
3. Make sure that you do not have anything in the function R files that is not a function as the `source` file simply runs every line of code in the file so if you have other things going on in there that actually do something to your data or test the functions, they will run too which can mess things up.
4. R will load your functions into memory a whole lot quicker if you access them using the `source` command than if you input them in the command line. This can save you a lot of time with 1000+ line functions.

Using the source command wisely can let you have a Master script that will run all of your analysis from a single R file kind of like a master DO file in Stata. One thing you should always make sure to do is save any changes you make in the file you define your functions in as soon as you make them because those changes will not be reflected when you use the source function again until they have been saved. This can be a problem if you had an error in a function or wanted to make a change to it but then forget to save the file before using the source function to load in the new functions again in your master script.

3 Looping

Looping is really the heart of this tutorial. While you may be able to do a lot of the same things using the apply family of functions (and sometimes faster), using for and while loops are often much easier to debug and for most applications will do things as fast as you could ever need to do them. There are also a number of things for which loops are the only way to go, so don't skip over this section, it is really important. There are two ways to write a loop – you can tell R to do something a set number of times using a `for()` loop, or you can tell R to do something for an unspecified number of times until some condition is met using a `while()` loop. While these are highly related and are most often functionally the same, I will go over the benefits of each below and why we would want to use one as opposed to the other below:

3.1 The `for()` loop

A for loop is generally the simplest and fastest to implement and should be used in any situation where we want to go through every element of a vector, matrix, array or list and do something or check something at each stop. The basic syntax for a `for()` loop is shown below:

```
for(i in 1:N){ # in english: do something N times where i is the current index
  #do something
}
```

In practice, we usually want to do something to every element of a vector so a flexible way to specify a loop is to use the length of the vector as the number of iterations to use in the for loop. Here is an example with addition, wrapped into a function:

```
Vector_Sum <- function(vector){
  sum <- 0
  for(i in 1: length(vector)){
    sum <- sum + vector[i]
  }
  return(sum)
}
```

We can also nest loops to go through a matrix by going through every row and then every column in every row. Here is another example wrapped into a function:

```
Matrix_Sum <- function(matrix){
  sum <- 0
  for(i in 1: length(matrix[,1])){
    for(j in 1:length(matrix[1,])){
      sum <- sum + matrix[i,j]
    }
  }
  return(sum)
}
```

```
}
```

3.2 The while() loop

While loops provide a good alternative to for loops that can help us speed up our code if we only need to look for one thing or if we need to do something an indeterminate number of times. A while loop works by continuing to iterate until some condition is reached. A simple example that mirrors the one in the section above simply does the incrementing of *i* manually

```
i <- 1
while(i < N){ # note that we use the < sign because on the last iteration we will
# set i = N and we don't want to add that last iteration.
  # do something
  i <- i + 1
}
```

This loop will run in the same amount of time as a for loop but takes a couple of extra lines of code to write. However, suppose we have a vector of length 10,000,000 where all entries are zero except for one which is equal to one, signifying some important event such as who sent a particular text message in a huge database. If we use a for loop to try and find the index of the person who sent the message, it will not stop even if the index is the first one, it will go through every element in the vector. We can write a while loop that will run in about half the time on average. below is a neat little example that demonstrates this:

```
#system.time() will let us measure the time it takes to run our loops
vector <- rep(0, 10000000)
random <- round(runif(1,1,10000000))
vector[random] <- 1

#now find the index of that number
forfind <- function(vector){
  for(i in 1:length(vector)){
    if(vector[i] == 1){
      index <- i
    }
  }
  return(index)
}

whilefind <- function(vector){
  found <- FALSE
  i <- 1
  while(!found){ # notice that if there is no entry that is equal to one,
# this will produce an error
    if(vector[i] == 1){
      index <- i
      found <- TRUE
    }
    i <- i + 1
  }
  return(index)
}
```

```
}

system.time(whilefind(vector))
system.time(forfind(vector))
```

A for loop is slightly more efficient in how it iterates so if the while loop has to go through the whole vector, a for loop will still be faster, but say the index we are looking for happens right at the beginning (at $i = 30,000$ in this example), then we can realize huge time savings.

```
> system.time(whilefind(vector))
  user system elapsed
0.037 0.000 0.037
> system.time(forfind(vector))
  user system elapsed
6.784 0.021 6.805
```

Of course in this particular example it is actually most efficient to use the `which` function, which does the same thing as our `forfind` function but does it using the underlying C code that R is built upon, making it blazing fast and really simple:

```
> system.time(which(vector == 1, arr.ind = T))
  user system elapsed
0.035 0.000 0.035
```

4 The Apply Family and Other Useful Functions

The `apply` family of functions does just what the name implies, they take a function and *apply* it to a specified subset of values (or all values in a vector, list, matrix, array or really kind of data structure you can think of. This can be very useful for taking things like the sum of rows in a matrix or for normalizing all values in a vector but can also do more complicated things like generating 30 graphs at once or running regressions on a bunch of data sets simultaneously. One of the nice things about the `apply` family of functions is that they take care of keeping track of values for you so you do not have to write a bunch of data structures to hold intermediate values while you are performing intermediate steps. We will walk through several commonly used functions in the `apply` family and finish by going over a few other useful functions in this vein.

You will find when you are talking to people who use R a lot that the conversation may naturally drift to whether or not you use the `apply` family of functions. This is kind of silly as some people see them as the only way to go and can be sort of *apply snobs*. I mostly use for loops in my own work as they tend to be easier to debug and fast enough for most of my needs. However the `apply` family of functions lets us write really elegant one-liners and takes advantage of the underlying C code in R to sometimes do things really really fast.

Here are a few additional references I have found helpful in the past:

1. <http://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/>
2. http://seananderson.ca/courses/12-plyr/plyr_2012.pdf

4.1 The `apply(X, margin, function)` function

The **`apply`** function is the most general in many ways as it allows you to apply a function over any margin of an array, meaning that (for example) if you have a 3 dimensional array, you could easily get slice sums,

row sums. column sums over time, etc. Lets look at a simple example to motivate things. Say we have a big matrix and we want to know the column sums:

```
mat <- mat(1:400, 20, 20)
apply(mat, 2, sum) # take the sum over the second margin (columns) of the matrix mat

[1] 210 610 1010 1410 1810 2210 2610 3010 3410 3810 4210 4610 5010
[14] 5410 5810 6210 6610 7010 7410 7810
```

We might also want to apply our own function instead of a built in one like sum. I wrote a function that takes a integer as an input, adds ten to it and returns it as output:

```
# Add ten to an entry
AddTen <- function(num){
  num <- num + 10
  return(num)
}
```

This brings up an interesting point about the apply function: it will just try to do its best, even if you give it something it was not expecting. In the case of the apply function, it is expecting a function that is applied to a whole vector, but this function is only applied to one element in the vector. So the **apply** function deals with this by applying the function to every value in the vector for every column vector in the matrix. This is something to be careful of if you only wanted to apply the AddTen function to the first column, for example. In that case you would want to use the **sapply** function which we will cover next. The standard way to apply the function to every element individually in a matrix is to apply over both margins. Check out these examples:

```
#this applies the function to every element in the matrix
apply(mat, 2, AddTen)
#this is the correct way to apply the function to every element in the matrix
apply(mat, 1:2, AddTen)

#here is where we would see a difference:
apply(mat, 2, mean)
apply(mat, 1:2, mean)
```

One of the simplest practical applications of this function is in social network analysis where we can calculate the number of ties a person sends or the number of ties they receive.

4.2 The sapply(X, function) function

The **sapply** function works much the same as the **apply** function but is really meant to work on a vector or a list. This can be very useful for doing something like transforming every value in a vector:

```
set.seed(1234)
changer <- function(x){
  x <- sqrt(exp(x + runif(1,0,100)))
  return(x)
}
lis <- list(c(1:100))
vec <- c(1:100)
sapply(lis,changer)
sapply(vec,changer)
```

The nice thing about the **sapply** function is that it returns a vector, even if it is applied to a matrix or a list, which can save you steps in getting the results in a workable form.

4.3 The **mapply**(function, X) function

The **mapply** function works similarly to the **apply** and **sapply** functions, but it lets us send multiple arguments to a function. It also takes its arguments in the opposite order with the function coming first and arguments coming after. We can look at a simple example of a multiplication function that takes two arguments. The **mapply** function will recycle the shorter vector and perform element-wise multiplication of these two vectors.

```
multiply <- function(arg1, arg2){
  result <- arg1*arg2
  return(result)
}

vec<- c(1:3)
vec2<- c(1:90)
mapply(multiply,vec,vec2)
```

The main difference between the **mapply** function and the other **apply** functions is that you can give the function multiple variables. Most of the time, this will get the job done for you but some times it may be unwieldy or you may want to vectorize over arguments that are difficult to get in the right structure for the **mapply** function. A nice little workaround for this is to use a wrapper function. A **wrapper** function (as the name implies) simply warps the function you want to call, allowing you to add in additional arguments and use the **apply** or **sapply** functions. Here is an example:

```
random_multiply <- function(arg1){
  result <- multiply(arg1,runif(1,1,100))
  return(result)
}

vec2<- c(1:90)
sapply(vec2, random_multiply) #note the order of arguments goes back to (data, function)
```

4.4 The reshape library: melt and cast

Check out this introduction by the package author <http://had.co.nz/reshape/introduction.pdf> for more examples and other cool functions in the **reshape** package not covered here. The basic idea with this package is that we can take a dataset with some id variables or particular variables of interest and "melt" it into a big long dataset that contains every value of the id variable and the attending values of every one of the other variables. We can then "cast" this new data structure into an infinite number of shapes. This kind of package is better played around with than explained in words so we will try out this example in the workshop:

```
library(reshape)

#generate a toy dataset quality
a <- c(1:100)
b <- rep(c(1:4),25)
c <- rep(c(1:10),10)
d <- runif(100,1,100)
```



```
e <- seq(from = 1.01, to = 2, by = 0.01)

data <- as.data.frame(cbind(a,b,c,d,e))
names(data) <- c("id","color","age","taste","quality")

meltdat <- melt(data, id = c("age","color"))

# cast(data, formula, function)
cast(meltdat, color~variable, mean)
cast(meltdat, age~variable, sum)
cast(meltdat, age~variable, AddTen)
cast(meltdat, age + color~variable, mean)
```