

# Handout 4: Introduction to **R**

Dave Armstrong  
University of Wisconsin – Milwaukee  
Department of Political Science

e: [armstrod@uwm.edu](mailto:armstrod@uwm.edu)

w: [www.quantoid.net/ICPSR.php](http://www.quantoid.net/ICPSR.php)

## Contents

<b>1</b>	<b>Basic Function Writing</b>	<b>2</b>
1.1	Data Storage . . . . .	6
1.2	Saving the Results of <code>robse</code> . . . . .	7
1.3	Printing Stars . . . . .	8
1.4	One- vs Two-sided p-values . . . . .	9
1.4.1	If/then statements . . . . .	10
1.4.2	Using if/then statements for two-sided p-values . . . . .	11
1.5	Some Other Important Facts About Functions . . . . .	13
<b>2</b>	<b>Repeated Calculations</b>	<b>13</b>
2.1	The for loop . . . . .	13
2.2	Saving results . . . . .	15
2.3	Apply and its Siblings . . . . .	17
2.3.1	<code>apply</code> . . . . .	17
2.3.2	<code>sapply</code> . . . . .	19
2.4	By and Aggregate . . . . .	19
2.4.1	Aggregate . . . . .	20
2.4.2	<code>by</code> . . . . .	21

# 1 Basic Function Writing

There are a few different situations in which we might want to write our own functions. First, we might want to be able to execute multiple commands with a single command. Second, we might want to use **R**'s ability to do repeated calculations with `apply`, `by` or `for` to do something many times with a single command. Finally, we might want to make complicated graphs with the `lattice` package which will require writing a function.

The function `function()` in **R** is what you need to write functions. The most important thing when writing a function is that you know what you want **R** to do. The *very* basic structure of a function is

```
myfun <- function(x){
  do something to/with/using x
}
```

Not surprisingly, it's the `do something` part that you need to come up with. We will start out with some relatively simple examples, but these can get arbitrarily complicated.

You might want to know if you can make a function that sets parameters of another function so we don't have to. Let's take, for instance, the `CrossTable` function from the `gmodels` package. Let's say that we only want column percentages and cell counts along with the chi-squared statistic and its p-value. If we look at the help file for the `CrossTable` command, we can see what arguments we need to set.

```
CrossTable(x, y, digits=3, max.width = 5, expected=FALSE, prop.r=TRUE,
  prop.c=TRUE, prop.t=TRUE, prop.chisq=TRUE, chisq = FALSE,
  fisher=FALSE, mcnemar=FALSE, resid=FALSE, sresid=FALSE,
  asresid=FALSE, missing.include=FALSE,format=c("SAS","SPSS"),
  dnn = NULL, ...)
```

Notice here, we will probably want to change some of the options that are `TRUE` by default to `FALSE`.

In **R**, the defaults are set with the equal sign (=) in the command. For example in the `CrossTable` command, `x` and `y` have no defaults. That is to say, the command will not substitute a value for you if you do not provide arguments `x` and `y`. However, if you do not specify the `digits` argument, it defaults to 3. The argument `prop.r` defaults to `TRUE`, but we want to set it to `FALSE`.

```
CrossTable2 <- function(x,y){
  requires("gmodels")
  print(
    CrossTable(x,y, prop.r=F, prop.t=F, prop.chisq=F,chisq=T,
      format="SPSS")
  )
}
```

If we run the command now, we can see the results:

```
> CrossTable2(Duncan$ed.cat, Duncan$type)
```

```

Cell Contents
|-----|
|              Count |
|      Column Percent |
|-----|

```

Total Observations in Table: 45

x	y			Row Total
	bc	prof	wc	
(6.91,30.2]	17 80.952%	0 0.000%	0 0.000%	17
(30.2,53.5]	4 19.048%	2 11.111%	2 33.333%	8
(53.5,76.8]	0 0.000%	3 16.667%	3 50.000%	6
(76.8,100]	0 0.000%	13 72.222%	1 16.667%	14
Column Total	21 46.667%	18 40.000%	6 13.333%	45

Statistics for All Table Factors

Pearson's Chi-squared test

```
-----
Chi^2 = 46.42857      d.f. = 6      p = 2.432300e-08
```

You can see that we've controlled the output and presented exactly what we wanted. Notice that we've simply included x and y as arguments and we simply pass them to the original command. It's a bit more explicit what is going on if we do the following:

```

CrossTable2 <- function(a,b){
require("gmodels")
  xt <- CrossTable(x=a,y=b, prop.r=F, prop.t=F, prop.chisq=F,
  format="SPSS", chisq=T)
  print(xt)

```

```
}
```

Here, `a` and `b` are standing in for `x` and `y` in the new function `CrossTable2`. Here, we pass those along to the main command in the function with `x=a` and `y=b`. There are two other pieces of this command.

- `require('gmodels')` - does the following. If the `gmodels` library is loaded, then it does nothing. If it is not loaded, it loads it.
- `print(xt)` prints the object that was created on the previous line.

We can do more complicated things as well. As I suggested before, we can do this for arbitrarily long commands. We could make a function that presents the robust standard errors, since there is no “robust” option in **R**'s `lm` command. Here are the steps we would have to go through:

1. Make sure the `car` package is loaded.
2. Save the model coefficients.
3. Generate and save the standard errors.
4. Use 1 and 2 to calculate t-statistics
5. Use 3 to calculate a p-value.
6. Present the results.

As I said before, if you understand what you want to do, then it will be much easier to get **R** to do it for you. We could do something like the following:

```
1 robse <- function(model){
2   require("car")
3   coefs <- coefficients(model)
4   rses <- sqrt(diag(hccm(model, "hc0")))
5   tstat <- coefs/rses
6   pvals <- 2*(1-pt(abs(tstat), model$df.residual))
7   mat <- cbind(coefs, rses, tstat, pvals)
8   rownames(mat) <- names(coefs)
9   colnames(mat) <- c("Coefficient", "Robust_SE", "t-stat", "p-value")
10  round(mat,5)
11 }
```

This command follows exactly the steps that we specified above. Below is a brief description of what each line does:

1. Open the function
2. Make sure the `car` package is loaded

3. Save the model coefficients.
4. Save the robust standard errors
5. Calculate the t-statistics
6. Calculate the p-values
7. Aggregate all of the results in a matrix
8. Make the row-names of the matrix the names of the coefficients (for prettier printing)
9. Set the column names of the matrix (for prettier printing)
10. Print the matrix to the console.
11. Close the function.

We can use this to motivate discussion in a couple of different directions. First let's see what happens when we try to send the results of the `robse` command to an object:

```
> robsum <- robse(mod)
> robsum
```

	Coefficient	Robust_SE	t-stat	p-value
(Intercept)	-0.18503	3.27240	-0.05654	0.95519
income	0.59755	0.12193	4.90086	0.00002
education	0.34532	0.11186	3.08693	0.00366
typeprof	16.65751	7.77408	2.14270	0.03828
typewc	-14.66113	5.29984	-2.76633	0.00854

Let's imagine now that we also wanted this function to print the original model summary. To do this, we would simply have to put `print(summary(model))` in the function. I've made two functions: one where it was put before `round(mat, 5)` called `robse2a` and one where it was put after `round(mat, 5)` called `robse2b`. We might hope that these two functions behave in roughly the same way, but let's see what happens when we assign their output to objects - `robsum2a` and `robsum2b`, respectively.

```
> robsum2a
```

	Coefficient	Robust_SE	t-stat	p-value
(Intercept)	-0.18503	3.27240	-0.05654	0.95519
income	0.59755	0.12193	4.90086	0.00002
education	0.34532	0.11186	3.08693	0.00366
typeprof	16.65751	7.77408	2.14270	0.03828
typewc	-14.66113	5.29984	-2.76633	0.00854

```
> robsum2b
```

```
Call:
lm(formula = prestige ~ income + education + type, data = Duncan)
```

```
Residuals:
```

```
      Min       1Q   Median       3Q      Max
-14.890  -5.740  -1.754   5.442  28.972
```

```
Coefficients:
```

```
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.18503    3.71377  -0.050  0.96051
income         0.59755    0.08936   6.687 5.12e-08 ***
education     0.34532    0.11361   3.040  0.00416 **
typeprof      16.65751    6.99301   2.382  0.02206 *
typepwc      -14.66113    6.10877  -2.400  0.02114 *
---
```

```
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

```
Residual standard error: 9.744 on 40 degrees of freedom
```

```
Multiple R-squared: 0.9131, Adjusted R-squared: 0.9044
```

```
F-statistic: 105 on 4 and 40 DF, p-value: < 2.2e-16
```

There are a couple of different things to notice here. First, if we look back at all of the output from the commands, without more modification, both only print the original model summary to the console. To change this behavior, we also have to put the `print()` command around the `round(mat,5)` command to force it to print the matrix to the console. The second difference is that the save output from the first function is only the robust summary and the saved output from the second function is only the original model summary. This is because when you're assigning the output, only the last command that prints to the console gets saved in the object output. How can we change this?

The `return()` command will allow you to return a list of model output rather than just the last piece of information that was printed to the console.

## 1.1 Data Storage

Since we need to use a `list` to save multiple pieces of output, it is probably worth talking about the different kinds of data storage that we haven't talked about already. Remember, we have talked about data frames, vectors and matrices already. The two things we haven't talked about are lists and arrays.

A list is simply a collection of (usually named) pieces of data. What separates a list from all other data storage types is that the different elements of the list do not have to have the same size. The output from a linear model command is a list. If we look at `names(mod)` (from the model used above in the robust summary commands), we will see the following:

```
> names(mod)
 [1] "coefficients" "residuals"    "effects"      "rank"
 [5] "fitted.values" "assign"       "qr"           "df.residual"
```

```

[9] "contrasts"      "xlevels"      "call"        "terms"
[13] "model"

```

We refer to each of these 13 different pieces of information, each as an *element* of the list. It is possible (actually likely) that each element of the list will itself contain many pieces of information. For example the `coefficients` element of the model object contains the vector of model parameter estimates. It is a vector of length 5 in this case. The `residuals` element is a vector of model residuals and since there is one residual for each observation, it has a length of 45. The element `df.residual` is a scalar giving the residual degrees of freedom. Finally (for our purposes), the element `qr` is, itself the result of a command called a QR decomposition. Further, this element of the list is itself a list. You can see the power and flexibility of this storage type. Let's say you want to save two elements in a list (A and B) and you want them to have names "elementA" and "elementB", you can do the following `obj <- list(elementA=A, elementB=B)`

An array is another form of storage that is the three-dimensional extension of a matrix. You can think of an array as a collection of matrices of the same size. Arrays also have a three-dimensional index `[r,c,f]` where the `f` stands for "face". Each face of an array is a matrix. Let's see how this works:

```

> my.array <- array(1:24, dim=c(4,3,2))
>
> my.array
, , 1
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24

```

As you can see, there is a considerable amount of structure here. If we want to see only the first  $4 \times 3$  matrix, we could index it in the following way: `my.array[, , 1]`.

## 1.2 Saving the Results of `robse`

Let's say that we want to return both the original summary and the robust summary from the `robse` command to the object (rather than only the last one). We can do this by using the `return()` command. Specifically, we can add this as the last line of the command: `return(list(robust = round(mat,5), orig=summary(mod)))`, in place of

both of the print commands. Now, the resulting model object (I called it `robsum2c`) is a list that has two elements `robust` which could be accessed by `robsum2c$robust` and `orig`, which could be accessed by `robsum2c$orig`.

### 1.3 Printing Stars

As yet another step in the program, what if we want to add the ever-important stars to our robust summary. If we follow the convention, we want to add one star if  $0.01 \leq p < 0.05$ , we want to add 2 stars if  $0.001 \leq p < 0.01$  and we want to add three stars if  $p < 0.001$ . To accomplish this, we can use the `which` command. This command tells us which values of a vector fulfill some conditional expression. We want to evaluate the each coefficient in the robust output on these three conditions mentioned above. We could do this as follows:

```
one.star <- which(pvals >= 0.01 & pvals < 0.05)
two.star <- which(pvals >= 0.001 & pvals < 0.01)
three.star <- which(pvals < 0.001)
```

Then we will have to find some way to attach that many stars to each standard error. The simplest way of doing this is to simply make another column for the results matrix that will have the correct number of stars for each coefficient. To do this, we can add:

```
star.vec <- rep("", length(coefs))
star.vec[one.star] <- "*"
star.vec[two.star] <- "**"
star.vec[three.star] <- "***"
```

Now, since we have both strings and numbers, we can make `mat` in the code a data frame, rather than a matrix, because the printing will be prettier.

```
mat <- data.frame(Coefficients = round(coefs, 5), Robust_SE = round(rses, 5),
  t_stat = round(tstat, 5), p_value = round(pvals, 5), "sig" = star.vec)
```

Now, we can print the data frame `mat` to the console with `print` command and we can use `cat` (another printing command that prints some objects (like vectors) in a more aesthetically pleasing manner) to print a legend for the table to the console as well.

```
print(mat)
cat("* p < 0.05, ** p < 0.01, *** p < 0.001\n")
```

Finally, to ensure that we get the data frame `mat` as output, we use the command `invisible` which is just like `return`, except that it does not print the result to the console, it only sends it out to the object.

```
> robse <- function(model){
+   require("car")
+   coefs <- coefficients(model)
+   rses <- sqrt(diag(hccm(model, "hc0")))
```

```

+   tstat <- coefs/rses
+   pvals <- 2*(1-pt(abs(tstat), model$df.residual))
+   one.star <- which(pvals >= 0.01 & pvals < 0.05)
+   two.star <- which(pvals >= 0.001 & pvals < 0.01)
+   three.star <- which(pvals < 0.001)
+   star.vec <- rep("", length(coefs))
+   star.vec[one.star] <- "*"
+   star.vec[two.star] <- "**"
+   star.vec[three.star] <- "***"
+   mat <- data.frame(Coefficients = round(coefs, 5),
+     Robust_SE = round(rses, 5), t_stat = round(tstat, 5),
+     p_value = round(pvals, 5), "sig" = star.vec)
+   print(mat)
+   cat("* p < 0.05, ** p < 0.01, *** p < 0.001\n")
+   invisible(mat)
+ }
>
> robsum <- robse(mod)
      Coefficients Robust_SE   t_stat p_value sig
(Intercept)    -0.18503   3.27240 -0.05654 0.95519
income           0.59755   0.12193  4.90086 0.00002 ***
education       0.34532   0.11186  3.08693 0.00366 **
typeprof        16.65751   7.77408  2.14270 0.03828 *
typewc         -14.66113   5.29984 -2.76633 0.00854 **
* p < 0.05, ** p < 0.01, *** p < 0.001
> robsum
      Coefficients Robust_SE   t_stat p_value sig
(Intercept)    -0.18503   3.27240 -0.05654 0.95519
income           0.59755   0.12193  4.90086 0.00002 ***
education       0.34532   0.11186  3.08693 0.00366 **
typeprof        16.65751   7.77408  2.14270 0.03828 *
typewc         -14.66113   5.29984 -2.76633 0.00854 **

```

## 1.4 One- vs Two-sided p-values

To motivate a discussion of if/then statements, let's consider allowing us to choose whether we want one-sided or two-sided p-values. To allow this, we will have to allow another argument to our command. Since there are only two possible options here, it is probably easiest to make this a logical argument where if the argument is `TRUE`, we get two-sided p-values and if it is `FALSE`, we get one-sided p-values. We can change the arguments to our function as follows:

```
robse <- function(model, two.sided=T){
```

We have the choice here to allow the command to have a default (as it does above) or require users to specify the argument each time. If we leave it as it is, it defaults to `TRUE`, if we change it to `two.sided=F`, it defaults to `FALSE` and if we simply put `two.sided`,

then the user must specify the argument each time or get an error. I will leave it as it is, but when you're playing around with this you can change the argument if you like.

What else do we have to change? Well, we also have to change the way `pvals` is calculated because right now it is only doing two-sided p-values. There are a bunch of ways we could accomplish this task, but one way is through an if/then statement.

### 1.4.1 If/then statements

- `ifelse(condition, if T, if F)` can be used to do two relatively simple things - one thing when a conditional expression evaluates to `TRUE` and another relatively simple thing when it evaluates to `FALSE`. For example:

```
> x <- 2
> ifelse(x > 1, "X is greater than 1", "X is not greater than 1")
[1] "X is greater than 1"
> x <- 0
> ifelse(x > 1, "X is greater than 1", "X is not greater than 1")
[1] "X is not greater than 1"
```

- The more flexible way of doing this is with two statements `if(cond){if T}` and then `else{if F}`. The same thing as above could be accomplished as follows:

```
> x <- 2
> {if(x > 1){
+   "X is greater than 1"
+ }
+ else{
+   "X is not greater than 1"
+ }
+ }
[1] "X is greater than 1"
>
> x <- 0
> {if(x > 1){
+   "X is greater than 1"
+ }
+ else{
+   "X is not greater than 1"
+ }
+ }
[1] "X is not greater than 1"
```

What, then, are the differences between these two methods? Well, perhaps most importantly, in the second method, the “what you do if `TRUE`” and “what you do if `FALSE`” pieces can be arbitrarily long and complicated. This is not really the case with the `ifelse()` statement. The other important difference is that `ifelse()` is “vectorized”, but using `if()` and `else` is not:

```

> x <- c(-1,0,1,2,3,4)
> ifelse(x > 1, "X is greater than 1", "X is not greater than 1")
[1] "X is not greater than 1" "X is not greater than 1"
[3] "X is not greater than 1" "X is greater than 1"
[5] "X is greater than 1"      "X is greater than 1"
>
> {if(x > 1){
+   "X is greater than 1"
+ }
+ else{
+   "X is not greater than 1"
+ }
+ }
[1] "X is not greater than 1"
Warning message:
In if (x > 1) { :
  the condition has length > 1 and only the first element will be used

```

#### 1.4.2 Using if/then statements for two-sided p-values

The reason we engaged in this discussion of if/then statements was to facilitate the extension of our summary program to produce both one- and two-sided p-values. Let's think about how we could write this function.

```

{if(two.sided == TRUE){
  pvals <- 2*(1-pt(abs(tstat), model$df.residual))
}
else{
  pvals <- (1-pt(abs(tstat), model$df.residual))
}
}

```

This is one of probably 5 or 6 different ways you could accomplish this task. If you want to practice your **R** programming, you might try to think of some other ways that you could accomplish this goal. At least one of them could be with `ifelse()`, another is using `math` and `as.numeric` and still another uses `math` and the indexing we talked about last week.

Should we change anything else? Well, we should probably change the legend so that it indicates whether the significance is with one-sided or two-sided p-values. We can use a similar if/then statement to do this:

```

{if(two.sided=T){
  cat("* p < 0.05, ** p < 0.01, *** p < 0.001 (two-sided)\n")
}
else{
  cat("* p < 0.05, ** p < 0.01, *** p < 0.001 (one-sided)\n")
}
}

```

Putting this all together, we would get something that looks like the following:

```

> robse <- function(model, two.sided=T){
+   require("car")
+   coefs <- coefficients(model)
+   rses <- sqrt(diag(hccm(model, "hc0")))
+   tstat <- coefs/rses
+   {if(two.sided == TRUE){
+     pvals <- 2*(1-pt(abs(tstat), model$df.residual))
+   }
+   else{
+     pvals <- (1-pt(abs(tstat), model$df.residual))
+   }
+ }
+   one.star <- which(pvals >= 0.01 & pvals < 0.05)
+   two.star <- which(pvals >= 0.001 & pvals < 0.01)
+   three.star <- which(pvals < 0.001)
+   star.vec <- rep("", length(coefs))
+   star.vec[one.star] <- "*"
+   star.vec[two.star] <- "**"
+   star.vec[three.star] <- "***"
+   mat <- data.frame(Coefficients = round(coefs, 5),
+     Robust_SE = round(rses, 5), t_stat = round(tstat, 5),
+     p_value = round(pvals, 5), "sig" = star.vec)
+   print(mat)
+   {if(two.sided == T){
+     cat("* p < 0.05, ** p < 0.01, *** p < 0.001 (two-sided)\n")
+   }
+   else{
+     cat("* p < 0.05, ** p < 0.01, *** p < 0.001 (one-sided)\n")
+   }
+ }
+   invisible(mat)
+ }
>
> robsum <- robse(mod)
      Coefficients Robust_SE   t_stat p_value sig
(Intercept)   -0.18503   3.27240 -0.05654 0.95519
income         0.59755   0.12193  4.90086 0.00002 ***
education     0.34532   0.11186  3.08693 0.00366 **
typeprof     16.65751   7.77408  2.14270 0.03828 *
typewc      -14.66113   5.29984 -2.76633 0.00854 **
* p < 0.05, ** p < 0.01, *** p < 0.001 (two-sided)
>
> robsum <- robse(mod, two.sided=F)
      Coefficients Robust_SE   t_stat p_value sig
(Intercept)   -0.18503   3.27240 -0.05654 0.47760
income         0.59755   0.12193  4.90086 0.00001 ***
education     0.34532   0.11186  3.08693 0.00183 **
typeprof     16.65751   7.77408  2.14270 0.01914 *
typewc      -14.66113   5.29984 -2.76633 0.00427 **
* p < 0.05, ** p < 0.01, *** p < 0.001 (one-sided)

```

## 1.5 Some Other Important Facts About Functions

There are a couple of other things to note here. First, the function will not look in the workspace for its arguments. Think back to the robust standard error function. Its first argument is `model`. If we omit the first argument, even if there is a model object in the workspace called `model`, the command will not use that as the argument.

Another interesting feature is that the way the command is written right now, the vectors (`coefs`, `rses`, `tstat`, `pvals` and `star.vec` as well as the data frame `mat` are only written locally to the command. That is to say they are available to any command operating inside the `function()` environment, but will not be written out to the workspace unless you ask to do that explicitly. To write something from inside a function out to the workspace, you need to specify a different assignment character. Here, you need `<<-`.

Hopefully, this gives you a very basic introduction to how you might be able to write your own functions. Again, I can't make this point strongly enough - this becomes much easier when you know what you want **R** to do. It often helps to write out on paper the steps you want to accomplish - then it is simply a matter of translating those steps from Human to **R**.

## 2 Repeated Calculations

One of **R**'s strengths is its incredible flexibility in doing repeated calculations. There are a number of functions that make this easier, but the most intuitive of these is the `for` loop. We will talk about these first and then talk about other commands that do some things better or at least more efficiently. Two themes discussed above maintain their importance here. First, with these repeated calculations, they can be arbitrarily complicated. They can be nested inside user-written functions or you can nest user-written functions inside loops. The second is that it is important the you know what you want to have **R** do.

### 2.1 The for loop

The `for` loop is a very useful tool when dealing with aggregating over units or performing operations multiple times either on the same set of data or on different sets of data. To show the basic structure of a loop, consider the following example:

```
for(i in 1:10){
  print(mean(dat[,i]))
}
```

Here, the solitary character `i` holds the place for the number. It would be equivalent to do the following:

```
mean(dat[,1])
mean(dat[,2])
mean(dat[,3])
mean(dat[,4])
mean(dat[,5])
mean(dat[,6])
```

```
mean(dat[,7])
mean(dat[,8])
mean(dat[,9])
mean(dat[,10])
```

Though you often see `i` used as an index, you could perform the same task with:

```
for(fred in 1:10){
  print(mean(dat[,fred]))
}
```

The other piece of the loop provides the scope. Remember, we can make a sequence of integer values using the colon, for example:

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 2:4
[1] 2 3 4
```

In the loop above, the command was a function of `i` or `fred`. However, that need not be the case. If you simply wanted to do something 10 times (where that something was not a function of the iteration), you could do that, too

```
> for(apple in 1:10){
+   print(mean(rnorm(100)))
+ }
[1] 0.03391467
[1] -0.04944169
[1] 0.01672039
[1] 0.063206
[1] -0.00811317
[1] 0.1559169
[1] -0.02408411
[1] -0.0254934
[1] 0.1099721
[1] 0.02074514
```

Notice that `apple` does not appear anywhere inside the loop. This set of commands draws a random-normal variate with length 100 and prints its mean each time.

Let's make the following matrix and then we can think about iterating some calculations on the matrix:

```
set.seed(10)
mat2 <- matrix(rnorm(1000), ncol=10)
```

## 2.2 Saving results

Since results don't make it out from inside the loop, we generally have to *initialize* an object outside of the loop and then fill it up inside the loop. For example, if I want to save the mean of each column of `mat2`, I know that I need to save 10 numbers (one for each column). If I make a vector that has 10 values, I can iteratively replace each value of the vector with the required number from inside the loop. Here is how I might do that:

```
> colmeans <- rep(NA, 10)
> for(lemon in 1:10){
+   colmeans[lemon] <- mean(mat2[, lemon])
+ }
>
> colmeans
[1] -0.13654894 -0.09496258  0.02875184  0.19883876 -0.05044530
[6] 0.06953555 -0.03247968 -0.12705238  0.04282867  0.21528147
```

Notice that I used the index `lemon` twice here - once to tell **R** which value of `colmeans` I wanted to replace and once to tell **R** the column of `mat2` for which I wanted to calculate the mean. Since I defined `colmeans` outside the loop, I can write values to it that will remain in the workspace.

We could also do a similar thing for the mean in the rows:

```
> rowmeans <- rep(NA, 100)
> for(orange in 1:100){
+   rowmeans[orange] <- mean(mat2[orange, ])
+ }
```

What if I want to know the result of a repeated calculation that does not take the form of a scalar (or even a vector). For example, what if I wanted to run a linear model a number of different times and save the output each time? Specifically, what if I wanted to estimate a linear model where the first column of `mat2` is the dependent variable and (in turn) each of the remaining columns is the independent variable? The result will have to be saved in a list.

First, a few helpful hints about lists:

- The elements of a list can be indexed by `[[#]]` where `#` is the number of the element you want to see/extract.
- To initialize a list, simply make a new object in the following way: `newobj <- list()`
- Lists have to be filled in with sequential integers starting at 1. So, if your counter does not start at 1, you will have to also make another counter that does start at 1 to fill in the list and increment it in the function. (I'll show an example of this below)

Here is how we could make the loop for the problem above:

```

for(j in 2:ncol(mat2)){
  tmp <- lm(mat2[,1] ~ mat2[,j])
  print(summary(tmp))
}

```

This loop just prints the output to the console. The following way would save the output in a list.

```

mylmres <- list()
k <- 1
for(j in 2:ncol(mat2)){
  mylmres[[k]] <- lm(mat2[,1] ~ mat2[,j])
  k <- k+1
}

```

Now, there is a new object in the workspace called `mylmres`, which is a list with 9 elements. Each element is the output from a different bivariate linear model. We could see the first one by typing:

```
> mylmres[[1]]
```

```

Call:
lm(formula = mat2[, 1] ~ mat2[, j])

Coefficients:
(Intercept)  mat2[, j]
-0.14190     -0.05634

```

We could see the summary for this model as follows:

```
> summary(mylmres[[1]])
```

```

Call:
lm(formula = mat2[, 1] ~ mat2[, j])

Residuals:
    Min       1Q   Median       3Q      Max
-2.04453 -0.69792 -0.01411  0.74437  2.37827

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.14190     0.09490  -1.495   0.138
mat2[, j]   -0.05634     0.09791  -0.575   0.566

```

```

Residual standard error: 0.9444 on 98 degrees of freedom
Multiple R-squared: 0.003368, Adjusted R-squared: -0.006802
F-statistic: 0.3312 on 1 and 98 DF, p-value: 0.5663

```

So, each element of `mylmres` behaves exactly like the output from a linear model.

Now, let's imagine that we wanted to get the  $R^2$  from each model. There is an element of the summary object from a linear model called `r.squared`. For example, if you have a linear model object called `mod`, you can get the  $R^2$  by typing `summary(mod)$r.squared`. Let's try to get this and save the results in a vector called `r2s`.

```
> r2s <- rep(NA, length(mylmres))
> for(i in 1:length(mylmres)){
+   r2s[i] <- summary(mylmres[[i]])$r.squared
+ }
>
> r2s
[1] 3.368004e-03 5.609637e-05 1.519993e-03 3.107116e-04 3.813659e-03
[6] 1.867132e-02 1.331516e-02 8.759170e-04 4.105833e-02
```

We could have combined both of these functions in the same loop as follows:

```
mylmres <- list()
r2s <- rep(NA, length(2:ncol(mat2)))
k <- 1
for(j in 2:ncol(mat2)){
  mylmres[[k]] <- lm(mat2[,1] ~ mat2[,j])
  r2s[k] <- summary(mylmres[[k]])$r.squared
  k <- k+1
}
```

## 2.3 Apply and its Siblings

There are a number of commands that have the suffix `apply` (e.g., `lapply`, `tapply`, `sapply`, `mapply`, `apply`) that can execute some function repeatedly on various data structures. The benefit of these functions is that they are more efficient in two ways. Often times, they are more efficient to specify and they are more computationally efficient as well. Let's go back to the row and column means example.

### 2.3.1 apply

The `apply` function performs a function for some *margin* of a matrix (or array). The command is as follows:

```
apply(data, margin, function)
```

Where `data` is the matrix, `margin` is 1 for rows and 2 for columns and `function` is some function that is either defined within the command or some function that is defined elsewhere that is called in the function. I'll show a couple of examples below. Above we applied the mean function to each column of `mat2` and each row of `mat2` independently. To do this, we can use `apply`.

```
colmeans <- apply(mat2, 2, mean)
rowmeans <- apply(mat2, 1, mean)
```

Notice that this is much faster. You can read the first line above as

- We want to apply the function `mean` to the columns (`margin = 2`) to the matrix `mat2`.

The second line can be read similarly, though the margin changes to 1, so we are applying the function to the rows instead of the columns.

What if there are some missing values in the matrix? If this happens, the mean function will return `NA` for the row or column that contains the missing value unless `na.rm=T` is offered as an argument to the command `mean`. You can do this in `apply`. The additional arguments come after the function name. For example:

```
colmeans <- apply(mat2, 2, mean, na.rm=T)
rowmeans <- apply(mat2, 1, mean, na.rm=T)
```

Just for the sake of demonstration, this could be done equivalently as follows (just looking at the `colmeans` example):

```
colmeans <- apply(mat2, 2, function(pear)mean(pear, na.rm=T))
```

Notice here, that I am defining the function that will be applied to the columns. Here, `pear` is standing in for each column of `mat2`. Here, it is explicit that `na.rm=T` is an argument to the command `mean`.

We can also do more complicated things. For example, we could also use `apply` to get the same result as was in `mylmres` above:

```
mylmres <- apply(mat2[,2:ncol(mat2)], 2, function(banana)
  lm(mat2[,1] ~ banana))
```

Notice, here, that we are using only columns 2- $k$  of the matrix `mat2`. The function we're defining runs a linear model regressing the first column of `mat2` on the  $k^{th}$  column where  $k = \{2, 3, 4, 5, 6, 7, 8, 9\}$ . The result here is a list, just as it was when we used a `for` loop to fill in the list.

If we wanted to obtain the  $R^2$  from each model, we could do so as follows:

```
r2s <- apply(mat2[,2:ncol(mat2)], 2, function(banana)
  summary(lm(mat2[,1] ~ banana))$r.squared)
```

Notice, I've kept the structure basically the same, but have defined that function that is operating on each column of `mat2` differently. The output now is a vector of  $R^2$  values, rather than a list of linear model output.

### 2.3.2 `sapply`

The `sapply` command does for lists what `apply` does for matrices. It applies the command element-wise to the list and returns that simplest data structure. Thus, we could obtain the same result with the  $R^2$  values by applying a function to the `mylmres` object.

```
r2s <- sapply(mylmres, function(x)summary(x)$r.squared)
```

The possibilities with `apply` and `sapply` are nearly endless. They can be nested within each other. For example, the following would work in an equivalent fashion to the last command:

```
sapply(apply(mat2[,2:ncol(mat2)], 2, function(banana)lm(mat2[,1] ~ banana)),  
       function(x)summary(x)$r.squared)
```

As these get *very* complicated, they get much slower, but they are still potentially faster than a `for` loop accomplishing the same task.

## 2.4 By and Aggregate

There are two more commands that have some utility as repeated calculators. These commands are good at doing repeated calculations on different subsets of the data. For instance, I might want to know the mean of some variable, for each different value of a factor. To make this concrete, let's think about wanting to know the mean of education for each different type in the Duncan data. I could do this "by hand" as follows:

```
> mean(Duncan$education[Duncan$type == "bc"])  
[1] 25.33333  
> mean(Duncan$education[Duncan$type == "prof"])  
[1] 81.33333  
> mean(Duncan$education[Duncan$type == "wc"])  
[1] 61.5
```

There would also be a way to do this with a `for` loop. We talked about loops yesterday, but there are a couple of different pieces here. First, we want to loop over the *unique* values of the factor `type`. To find out what those are, we can do the following:

```
> untype <- as.character(unique(Duncan$type))  
> untype  
[1] "prof" "wc"   "bc"
```

Rather than us specifying our loop over the values `1:3`, we can specify the loop over the values `1:legnth(untype)`. This will do the same thing, but we don't have to count how many values are in `untype` (this is not problematic when there are three, but could get problematic when there are lots more unique values). We then need to make the following loop:

```

> means <- rep(NA, length(untypes))
> for(i in 1:length(untypes)){
+   means[i] <- mean(Duncan$education[Duncan$type == untypes[i]])
+   names(means)[i] <- untypes[i]
+ }
>
> means
      prof      wc      bc
81.33333 61.50000 25.33333

```

First, we initialized the vector of means, then we set up the loop over the values 1 to the length of the `untypes` vector. Next for each `i`, we replace the  $i^{th}$  value of the vector `means` with the mean of education for the observations whose `type` equal the  $i^{th}$  value of `untypes`. Finally, we change the  $i^{th}$  value of the names attribute of the `means` vector to the  $i^{th}$  value of the `untypes` vector.

### 2.4.1 Aggregate

Rather than do this, we could use either `by` or `aggregate` depending on the type of output we want. If the output of each repeated calculation is a scalar, then we can use `aggregate` which will return a data frame. We can do this as follows:

```

> ag <- aggregate(Duncan$education, list(Duncan$type), mean)
> ag
  Group.1      x
1      bc 25.33333
2     prof 81.33333
3      wc 61.50000

```

The object `ag` is a data frame whose first column indicates the value of `type` and whose second column is the mean of education for that value of `type`. There are three arguments to the `aggregate` command. First is the data we want to aggregate (here, the values of education). Next are the values we want to aggregate by (here, the values of type). Note, that the second argument *must* be a list, even if it is only one variable. Finally, the last argument is the function that will do the aggregation. Here it is the mean, but you could do any function that returns a scalar.

You can aggregate by more than one variable. Consider the following:

```

> inc.dum <- cut(Duncan$income, 2)
> ag2 <- aggregate(Duncan$education, list(Duncan$type, inc.dum), mean)
> ag2
  Group.1  Group.2      x
1      bc (6.93,44] 24.47368
2     prof (6.93,44] 71.50000
3      wc (6.93,44] 61.00000
4      bc (44,81.1] 33.50000
5     prof (44,81.1] 84.14286
6      wc (44,81.1] 61.75000

```

Notice the only thing that changed here is another element was added to the list in the second argument. The output now, gets the mean for each combination of the two variables in the list.

### 2.4.2 by

The input to the `by` command is the same as the input to the `aggregate` command, though it's a bit more flexible about the nature of the first argument. Where the `aggregate` command really wants a vector, you give `by` a matrix or data frame as its first argument. As suggested above, `by` returns a list rather than a data frame, which may be less convenient, but I'll show you below how we can change it back to a matrix.

```
> by1 <- by(Duncan$education, list(Duncan$type), mean)
>
> by1
: bc
[1] 25.33333
-----
: prof
[1] 81.33333
-----
: wc
[1] 61.5
```

We can make this into a vector simply by typing:

```
> by1.vec <- c(by1)
> by1.vec
      bc      prof      wc
25.33333 81.33333 61.50000
```

One of the benefits of the `by` command as opposed to `aggregate` is it allows you to provide a matrix as the first argument to the command and allows the function to perform matrix operations. One simple thing we might want to do is take the mean of three columns of a matrix for the values of another variable. Specifically, we might want to know the means of prestige, education and income for the values of type in the Duncan dataset.

```
> by2 <- by(Duncan[,c("prestige", "income", "education")],
+ list(Duncan$type),
+ function(x)apply(x, 2, mean))
>
>
> by2
: bc
prestige  income education
22.76190  23.76190  25.33333
```

```
-----
: prof
prestige    income education
80.44444    60.05556  81.33333
-----
```

```
: wc
prestige    income education
36.66667    50.66667  61.50000
```

Now, what if we want this to be a  $3 \times 3$  matrix? Let's see what happens when we use `c()` on `by2`.

```
> c(by2)
$bc
prestige    income education
22.76190    23.76190  25.33333
```

```
$prof
prestige    income education
80.44444    60.05556  81.33333
```

```
$wc
prestige    income education
36.66667    50.66667  61.50000
```

Here, we can see that this is now a list object with three elements: `bc`, `prof` and `wc`. As it turns out, we can use `as.data.frame` here to make this list into a data frame:

```
> as.data.frame(c(by2))
      bc      prof      wc
prestige 22.76190 80.44444 36.66667
income   23.76190 60.05556 50.66667
education 25.33333 81.33333 61.50000
```

As an aside, we would not be able to do the following: `as.data.frame(by2)`, using `c()` is a crucial step here. If we wanted the orientation of the matrix to be reversed (i.e., the types in the rows), we could just transpose it:

```
> t(as.data.frame(c(by2)))
      prestige  income education
bc  22.76190 23.76190  25.33333
prof 80.44444 60.05556  81.33333
wc   36.66667 50.66667  61.50000
```