







R Workshop 5: Automation

Teal Potter

11/12/2021

Table of Contents

Setup		1
Functions.....		2
Example function 1  <i>EASIEST</i>		2
Example function 2  <i>EASIEST</i>		3
Example function 3  <i>INTERMEDIATE</i>		3
Tips for building functions		4
For loops		4
Example For loop 1  <i>EASIEST</i>		5
Example For loop 2  <i>INTERMEDIATE</i>		5
Tips for building for loops		6
Example For loop 3  <i>INTERMEDIATE</i>		9

By 'automation', I mean using a method that allows you perform a repetitive task more efficiently. Functions and for loops are two advanced strategies that can help you save time and script space by avoiding copy/pasting code over and over and changing the variables each time. Functions and for loops can be used for the same uses so I recommend practicing the option that feels more intuitive to you for a while. If you happen to be operating on very large datasets or have complex maneuvers, note that functions take much less time to execute. You won't notice much difference for datasets with only a couple hundred samples.

Setup

```
library(car)  
library(ggplot2)
```

Functions

To Build a function you'll use the function() function and curly brackets to enclose the code you want use to develop what the function does with the arguments you provide it. Here is some pseudo code to show you what is included in a function.

```
function_name <- function(argument){ code that manipulates arguments in some way }
```

The name you assign in the first row of code will become the name of your function. One or more arguments can be created as the arguments of your function.

Using your function after you make it is like using any other type of function:

```
function_name(dataset)
```

Example function 1 EASIEST

Here is a function with one argument, x. This function takes a vector (x) and uses base R functions to calculate standard error.

```
se <- function(x){          # function with 1 argument, x. Function is named se.
  sd(x)/sqrt(length(x))    # standard deviation of x divided by n
}
```

Now you can test/use the se function:

```
vector1 <- c(2,6,5,6) # here is a vector we can test the function with
se(vector1)          # to test the function, place the vector in the parantheses like any
other function

## [1] 0.9464847

se(c(2,6,5,6) ) # altertnatively, we can specify the vector in the function
directly

## [1] 0.9464847
```

But what happens if our vector contains an NA?

```
se(c(2,6,5,6,NA))

## [1] NA
```

It returns NA. Let's make a new and improved function that can handle data with NAs. The sd() function, along with mean(), median(), and other functions can take an optional argument to state what to do with NAs. We can add na.rm = TRUE here and na.omit(x) which operates directly on the x vector to remove the NA from both parts of the calculation.

```
SE <- function(x) {
  sd(x, na.rm = TRUE)/sqrt(length(na.omit(x)))
}
```

Now when we test the SE() function it returns the same standard error calculation having ignored the NA in the data.

```
SE(c(2,6,5,6))
## [1] 0.9464847
SE(c(2,6,5,6,NA))
## [1] 0.9464847
```

Example function 2 EASIEST

Here is another function with a few more calculations made. This function calculates the number of samples, the mean, and standard error (using the function made above). Again a numerical vector must be provided as the function's argument.

```
sumry <- function(num_col) {
  Length <- length(num_col) # calculates sample size
  Mean <- mean(num_col) # calculates mean
  SE <- SE(num_col) # calculates standard error
  data.frame(Length, Mean, SE) # returns all 3 calculations
}

sumry(Soils$N)

##   Length      Mean      SE
## 1     48 0.1019375 0.009693504
```

Example function 3 INTERMEDIATE

In this final example, there are two arguments. *x* and *y* both need to be numerical vectors in order for this function to assess the linear relationship between these two variables using linear regression. The main reason I consider this more complex than the previous is because I've added several clean up steps to make the output clean to look at. This function returns the main test statistics from the model in a cleaner version than you get as a default when you use `summary(lm(...))`.

```
lm_coef <- function(x, y) {
  mod <- summary(lm(y ~ x)) # run linear model with x & y
  mod.df <- as.data.frame(t(data.frame(coefficients(mod)[2,]))) # keep coefficients
  row 2
  mod.df$R2 <- mod$r.squared # adding R2 since it's not in the coef table
  names(mod.df)[4] <- "P" # change 4th column name to P
}
```

```

clean <- round(mod.df, 3) # rounding all values to 3 places
clean$P <- ifelse(clean$P == 0, yes = "<0.01", no = clean$P) # if P rounds to 0
print(clean)
}

lm_coef(Soils$N, Soils$pH)

##              Estimate Std. Error t value      P      R2
## coefficients.mod..2...    6.369      1.137    5.599 <0.01 0.405

```

Tips for building functions

You will probably find it difficult to write your code inside the function because your argument doesn't represent real data at this point. One way to check that your lines of code work as you go is to assign data to your arguments. For example:

```

x <- c(1,2,3,4,5)
y <- c(1,2,3,4,6)

```

now you should be able to run the line of code that contains the model and the follow lines.

```

summary(lm(y ~ x))

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      1      2      3      4      5
## 2.000e-01  1.249e-16 -2.000e-01 -4.000e-01  4.000e-01
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.4000     0.3830  -1.044   0.3730
## x              1.2000     0.1155  10.392   0.0019 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3651 on 3 degrees of freedom
## Multiple R-squared:  0.973, Adjusted R-squared:  0.964
## F-statistic: 108 on 1 and 3 DF, p-value: 0.001901

```

For loops

For loops iterate (or cycle or loop) through a data object in a way that you specify so that you can perform a process and/or calculation on every group that meets your criteria. A common application is to append new information to a new table every time the for loop completes one pass through and makes the calculations.

The syntax set up a for loop can be auto-populated if you type the word for and then hit Tab on your keyboard.

Example For loop 1 EASIEST

Here is a super simple example to start. This for loop adds 1 to each element in the vector. *i* is a placeholder variable that represents different data each loop through the dataset. You can use any variable you like but it is common practice to use *i* and sometimes *j* and *k*.

```
for (i in 1:10) { # for each variable i from the first loop through the nth loop
  through data
  counter = i + 1 # code that manipulates i
  print(counter) # the output
}

## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
```

Example For loop 2 INTERMEDIATE

What if we want to know how many groups (aka levels) there are in each column containing factor data in the Soils dataset.

First we can make a new data frame that only contains columns with factor data.

```
factor_cols = dplyr::select_if(Soils, is.factor)
```

This is the way you might be tempted to perform this task, which I'm just showing to contrast with the for loop option below.

```
names(factor_cols)

## [1] "Group" "Contour" "Depth" "Gp" "Block"

length(levels(factor_cols$Group))

## [1] 12

length(levels(factor_cols$Contour))
```

```
## [1] 3
length(levels(factor_cols$Depth))
## [1] 4
length(levels(factor_cols$Gp))
## [1] 12
length(levels(factor_cols$Block))
## [1] 4
```

You can see how this could get annoying quickly if you had a lot of groups.

Alternatively, we can write a for loop to perform this code on all the columns of factor_cols.

```
for (j in 1:length(factor_cols)) { # specify # of columns to cycle through
  factor_cols.t = factor_cols[,j] # for each loop, operate on the jth column
  col = colnames(factor_cols)[j] # save the jth column name
  len = length(levels(factor_cols.t)) # count the # of levels in jth column & save
  print(paste(col, '=', len)) # output: column names & # of levels in column
}
## [1] "Group = 12"
## [1] "Contour = 3"
## [1] "Depth = 4"
## [1] "Gp = 12"
## [1] "Block = 4"
```

Tips for building for loops

Let's break that example down a bit more to show how you can test parts of the whole for loop. First you can check that you've included the correct number of iterations to cycle/loop through in the first line by selecting and running this code independently from the for loop

```
1:length(factor_cols) # 5 columns
```

```
## [1] 1 2 3 4 5
```

Next, to test whether you've specified a column to operate on you can set the variable j to a number and see if your subsetting code works with that single instance of j

```
j = 1
```

```
factor_cols[,j]
```

```
## [1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7
## [26] 7 7 7 8 8 8 8 9 9 9 9 10 10 10 10 11 11 11 11 12 12 12 12
## Levels: 1 2 3 4 5 6 7 8 9 10 11 12
```

We know this should be the first column in factor_cols, so let's compare

```
factor_cols[1]
```

```
##      Group
## 1      1
## 2      1
## 3      1
## 4      1
## 5      2
## 6      2
## 7      2
## 8      2
## 9      3
## 10     3
## 11     3
## 12     3
## 13     4
## 14     4
## 15     4
## 16     4
## 17     5
## 18     5
## 19     5
## 20     5
## 21     6
## 22     6
## 23     6
## 24     6
## 25     7
## 26     7
## 27     7
## 28     7
## 29     8
## 30     8
## 31     8
## 32     8
## 33     9
## 34     9
## 35     9
## 36     9
## 37    10
## 38    10
## 39    10
## 40    10
## 41    11
## 42    11
## 43    11
```

```
## 44    11
## 45    12
## 46    12
## 47    12
## 48    12
```

Looks good. Let's try 1 more column by changing `j` to represent 2nd column. We'll just check the top of the column this time using `head()`.

```
j = 2
head(factor_cols[,j])

## [1] Top Top Top Top Top Top
## Levels: Depression Slope Top

head(factor_cols[2])

##   Contour
## 1      Top
## 2      Top
## 3      Top
## 4      Top
## 5      Top
## 6      Top
```

That worked too. Looks like we've correctly showed the for loop how to recognize each column one at time. Now that we have `j` set equal to 2 we should be able to check the code in the remaining lines before running the whole for loop.

```
colnames(factor_cols)[j]      # save the jth column name

## [1] "Contour"

length(levels(factor_cols.t)) # count the number of levels (groups) in the jth
column & save

## [1] 4
```

Now those lines worked so let's save them to the temporary variable names so we can test the output code where we paste them together.

```
col = colnames(factor_cols)[j]      # save the jth column name
len = length(levels(factor_cols.t)) # count the number of levels (groups) in the
jth column & save

print(paste(col, '=', len)) #paste column name and # of levels in that column
together

## [1] "Contour = 4"
```

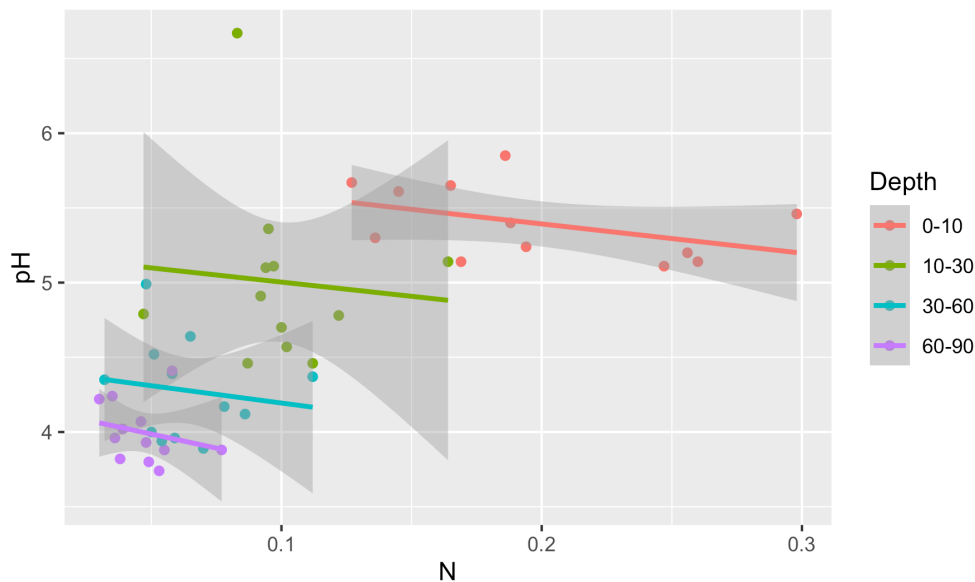
Once you've checked and gotten all the parts to work like this, you can be pretty sure that your for loop will run correctly.

Example For loop 3 ■ INTERMEDIATE

A common reason to build a for loop is if you need to run a statistical test on combinations of variables. Instead of typing out the functions and specifying the set of variables each time you run a test (literally copy/pasting code and just changing a few things), you can loop through your combinations to run a bunch of test and report the statistics of your choice in a new table.

Here is a visual for the example for loop below. Let's say you are interested in testing the relationship between N and pH but you would like to see if the relationship changes with depth.

```
ggplot(Soils, aes(x = N, y= pH, color = Depth))+
  geom_point()+
  geom_smooth(method = 'lm')
```



Here it looks like the slope is similar within each depth but let's test it.

```
data_frame_to_fill = data.frame() #reset each time you run for loop
```

```
depths = unique(Soils$Depth) # creating a new variable to specify the # of cycles through the data
```

```
for (i in 1:length(depths)) {
  Soils.temp = Soils[Soils$Depth == depths[i], ] # subset data to 1 depth per cycle
  mod <- summary(lm(pH ~ N, data = Soils.temp)) # run model using subsetted data
  mod.df = data.frame(coefficients(mod)) # save stats in a temporary data
  mod.df$depth <- depths[i] # save depth category name to the temporary data frame
  data_frame_to_fill = rbind(data_frame_to_fill, mod.df) # append new rows (data frame) to full data frame that gets added to with each loop through the data
}
```

Finally, run the name of the data frame to see what the for loop created.

```
data_frame_to_fill
```

```
##           Estimate Std..Error   t.value   Pr...t.. depth
## (Intercept)  5.783998  0.2626384 22.0226674 8.351646e-10  0-10
## N           -1.956127  1.2841444 -1.5232924 1.586628e-01  0-10
## (Intercept)1 5.193473  0.7120423  7.2937708 2.620330e-05 10-30
## N1          -1.900987  6.9178972 -0.2747927 7.890692e-01 10-30
## (Intercept)2 4.425775  0.3292346 13.4426169 9.977327e-08 30-60
## N2          -2.318866  4.9382566 -0.4695717 6.487327e-01 30-60
## (Intercept)3 4.174329  0.2354665 17.7279119 6.952866e-09 60-90
## N3          -3.762322  4.8458033 -0.7764083 4.554796e-01 60-90
```

It's not a pretty data frame but it's simple this way. We can fairly easily add code to round numbers, change column names, and omit the unneeded intercept rows as I've shown in the Example function 3 section. This detail can be added to the for loop or performed on the final data frame after it's created.