Lesson 5 of 5

Loops, Writing Functions, Debugging

Intro to R workshop, LU Skills School

Instructor:

Christopher Swader
LU Sociology (Assoc. Prof) and LMU (Munich, Researcher)

Teaching Assistant: Maximilian Hornung (MS programme in Social Scientific Data Analysis, LU)

Today’s agenda

- Loops
- Writing your own functions
- Debugging

- Use the link to download the files we will be using today: https://github.com/ChristopherSwader/R_introduction

- Download the Day 5 folder.

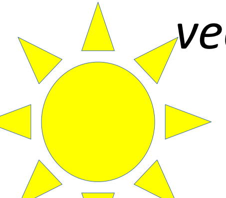
Introduction

Today we will cover more intermediate topics, to include

- if statements
- loops (for and while)
- functions
- debugging
- lapply functions

Loops

- In a more complex task, you want to avoid typing the same code twice!
- Needless repetition of code is harder to read, inefficient, and makes errors more likely. If you need to change something, you will have to do it multiple times.
- Often the length of a list or a vector might change, and looping through it is the most sensible way to iteratively perform some tasks.
- A sensible way to make your code efficient is through using loops.
- A more advanced topic, something to be aware of, is that if you are writing complex functions, packages, you should later try to update your loops by *vectorizing* them, because loops can be slow. But that is for another day…



Example: Multiverse analysis

- We will make a very simple version of a multiverse analysis to demonstrate today's content
- Multiverse analysis is a way to illustrate the web of possible research outcomes that derives from the different combinations of multiple research decisions
- It can be used to show how one's results are robust and not the result of a strange set of choices

Load Data

- We first load our flfp individual-level data

```
library(readr)
flfp <- readRDS("flfp-individual-
level.rds")
```

Run simple model

We run a basic model predicting patriarchical values.

```
library(broom)
simple_model <- lm(data=flfp, patr_values~ religious + age_gr +edu)
tidy(simple_model)
## # A tibble: 9 × 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   0.444    0.0268      16.6  1.98e- 61
## 2 religious     0.0753   0.00981      7.67 1.74e- 14
## 3 age_gr18-25  -0.373    0.0290     -12.9  9.37e- 38
## 4 age_gr26-35  -0.337    0.0280     -12.1  2.11e- 33
## 5 age_gr36-45  -0.322    0.0281     -11.5  2.28e- 30
## 6 age_gr46-55  -0.290    0.0285     -10.2  3.03e- 24
## 7 age_gr56-65  -0.206    0.0303      -6.79 1.10e- 11
## 8 eduMiddle    -0.211    0.0112     -19.0  8.34e- 80
## 9 eduHigh      -0.452    0.0144     -31.3  1.34e-212
```

Simple multiverse set up

We pretend that:

- We want to run the above analysis separately for each religious denomination
- We want to dichotomize age with different splits

```
#Make a vector of denominations
denominations <- levels(flfp$denom)
print(denominations)
## [1] "Christ" "Muslim" "Other"  "None"
#Put age categories in correct order
flfp$age_gr <-   factor(flfp$age_gr,
            ordered = TRUE,
               levels = c("18-25", "26-35","36-45", "46-55", "56-65", ">66"))
age_category_cutoff <- levels(flfp$age_gr )

#because we are going to use this to define who belongs to the lower age group, that means that
we don't need the upper one.
#so we cut it off
age_category_cutoff <- age_category_cutoff[-length(age_category_cutoff)]
print(age_category_cutoff)
## [1] "18-25" "26-35" "36-45" "46-55" "56-65"
```

First for loop

```r
#now we make a loop of the denominations
for (denom in denominations){
  #denom is a new variable created by the loop
    #denom changes for each iteration of denominations
  #let's print and see if it works
  print(denom)
}
## [1] "Christ"
## [1] "Muslim"
## [1] "Other"
## [1] "None"
```

Second for loop

```
#now we make a loop of the age group cutoffs within the other loop
#ALERT: nested loops (loops within loops) are slow... but since i only have a few items in each list
(and not thousands), it won't matter here.
for (denom in denominations){
  for (age_cutoff in age_category_cutoff){ #notice I indent here to keep track of the hierarchy of
loops

    #I again print something out to make sure i get the desired result
    #cat is a wonderful way to put together your own print messages
  cat("\n", denom, "and",age_cutoff, "combination")

  }
}
##
##  Christ and 18-25 combination
##  Christ and 26-35 combination
##  Christ and 36-45 combination
##  Christ and 46-55 combination
##  Christ and 56-65 combination
##  Muslim and 18-25 combination
```

Let’s instead loop by an index number

It is generally *FAR* more useful to loop by an index number than by the vector value. Index numbers can be more easily used to piece together different types of information.

```
#adapting code for my best practice
for (denom in 1:length(denominations)){
  for (age_cutoff in 1:length(age_category_cutoff)){
  cat("\n", denominations[denom], "and",age_category_cutoff[ age_cutoff], "combination")

    #now we have unique combinations of religious denomination and the age category
cutoff to work with.

  }
}
##
##  Christ and 18-25 combination
##  Christ and 26-35 combination
##  Christ and 36-45 combination
##  Christ and 46-55 combination
##  Christ and 56-65 combination
```

Choose the religious denomination subsample

```
#adapting code for my best practice
for (denom in 1:length(denominations)){
  for (age_cutoff in 1:length(age_category_cutoff)){

    this_denomination <- denominations[denom]
    temporary_flfp <- flfp[flfp$denom==this_denomination & !is.na(flfp$denom) ,]

     print(nrow(temporary_flfp)) #the number of rows should differ if the filtering worked!
    #notice how we again use print() or cat() to print out the output to make sure it looks correct!

  }
}
## [1] 18632
## [1] 18632
## [1] 18632
## [1] 18632
## [1] 18632
```

Your turn

Adapt the loop so that you use tidyverse instead to filter rows by religious denomination.

Dichotomize the age_group variable

```
library(dplyr)
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
for (denom in 1:length(denominations)){
  for (age_cutoff in 1:length(age_category_cutoff)){

    this_denomination <- denominations[denom]
    temporary_flfp <- flfp[flfp$denom==this_denomination,]
    this_age_cutoff <- age_category_cutoff[age_cutoff]
    age_gr_lower_group <- age_category_cutoff[1:age_cutoff]
    temporary_flfp$age_gr <-  ifelse(temporary_flfp$age_gr %in% age_gr_lower_group,"younger","older")
    temporary_flfp$age_gr <- as.factor(temporary_flfp$age_gr)
    print(sum( temporary_flfp$age_gr=="younger"))
  }
}
## [1] 2819
## [1] 7650
## [1] 12049
```

Run the regression and save the results

```
results_list <- vector("list", length =0)

for (denom in 1:length(denominations)){
  for (age_cutoff in 1:length(age_category_cutoff)){

    this_denomination <- denominations[denom]
    temporary_flfp <- flfp[flfp$denom==this_denomination,]
    this_age_cutoff <- age_category_cutoff[age_cutoff]
    age_gr_lower_group <- age_category_cutoff[1:age_cutoff]
    temporary_flfp$age_gr <-  ifelse(temporary_flfp$age_gr %in% age_gr_lower_group,"younger","older")
    temporary_flfp$age_gr <- as.factor(temporary_flfp$age_gr)

    #here is a way to enter in a regression formula so that the dv and ivs are changing, in case you would want
#different dvs and ivs to enter your multiverse analysis. here they are stable, but we merely use a differently defined age_gr variable and a different sample.

    dv <- "patr_values"
    ivs <- c("religious", "age_gr", "edu")

    f <- as.formula(
      paste(dv,
            paste(ivs, collapse = " + "),
            sep = " ~ "))

    this_regression <- eval(bquote(   lm(.(f), data = temporary_flfp)   ))

     this_model_iteration <- t(data.frame(this_regression$coefficients))


    results_list <- append(results_list, list(this_model_iteration))


  }
}
```

```
results_list[1:2]
## [[1]]
##                              (Intercept)  religious age_gryounger  eduMiddle
## this_regression.coefficients   0.2055364 0.07113498    -0.0862266 -0.2764633
##                                 eduHigh
## this_regression.coefficients -0.4978739
##
## [[2]]
##                              (Intercept)  religious age_gryounger  eduMiddle
## this_regression.coefficients   0.2235323 0.07321065   -0.08847895 -0.2712477
##                                eduHigh
## this_regression.coefficients -0.487907
```

While loop

- While loops are more dynamic than for loops, as they can keep running until a particular condition is complete.
- I can for example draw random subsets from the overall sample until the new sample's intercept is at least as large as that of one of my multiverse results.(Perhaps I want to afterwards use that new subset to compare with my multiverse results)

```
threshold_to_beat <- mean(bind_rows( lapply(results_list, data.frame))[,1]) # we take the mean intercept of the 20 models run

this_intercept <- 0 #we start the test intercept number at zero
intercepts <- vector("numeric",0)
while(this_intercept <=threshold_to_beat){

  this_sample <- sample(1:nrow(flfp), 5000, replace = F) #indices of this new subset

   temporary_flfp <- flfp[this_sample,]

   simple_model <- lm(data=temporary_flfp, patr_values~ religious + age_gr +edu)
this_intercept <-  tidy(simple_model)[1,2]
intercepts <- c(intercepts, this_intercept)
#print(unlist(unname(this_intercept)))

}

summary(temporary_flfp) #we can print a summary of this sample's characteristics to compare it with the the groups we have tested
```

```
##          cntry          year              wgt              lfp
##  India       : 159   Length:5000       Min.   :0.08035   Min.   :0.0000
##  South Africa: 151   Class :character   1st Qu.:0.88729   1st Qu.:0.0000
```

If statements

- If statements are the bread and butter of any programming language.
- Think of them as a door or a gate that is only passed if the condition is fulfilled.
- We will use one here to help us count the number of iterations of our while loop, since it is variable.

```
#we add an if statement and a counter inside the above while loop. We want to add a count so that every 100th iteration we get a message.

threshold_to_beat <- mean(bind_rows( lapply(results_list, data.frame))[,1])

this_intercept <- 0
intercepts <- vector("numeric",0)
counter <- 0 #we use this counter to count iterations within the while loop

while(this_intercept <=threshold_to_beat){
  counter <- counter+1#here the counter ticks forward

  this_sample <- sample(1:nrow(flfp), 5000, replace = F)

   temporary_flfp <- flfp[this_sample,]

   simple_model <- lm(data=temporary_flfp, patr_values~ religious + age_gr +edu)
this_intercept <-  tidy(simple_model)[1,2]
intercepts <- c(intercepts, this_intercept)

if ((counter %% 10) ==0){ # %% calculates the REMAINDER of division. So the remainder of x  divided by 10 is zero if x is some multiple of 10. In other words, this
if statement will trigger every 10th iteration
  cat("\nIteration number is", counter) #A message will be trigger by this statement
}

} #end while loop
##
## Iteration number is 10
## Iteration number is 20
## Iteration number is 30
## Iteration number is 40
```

Functions

Another important way to track what is happening in a complex routine and to avoid repetition is to use functional programming.

- Functions have an input and an output.
- As a result, it should be easy to see before and after a function.
- The opposite would be something like ‘spaghetti code,’ with lots of repetition and intransparency.

```
#this is how we define a function from our while loop above
random_subset <- function(threshold_to_beat=0){
  #we define the arguments that the function will take
  #we can set a default if we wish by setting equals to our desired value.

  #we comment this out, because the user will enter this in as an argument!
#  threshold_to_beat <- mean(bind_rows( lapply(results_list, data.frame))[,1])

this_intercept <- 0
intercepts <- vector("numeric",0)
counter <- 0 #we use this counter to count iterations within the while loop

while(this_intercept <=threshold_to_beat){
  counter <- counter+1#here the counter ticks forward

  this_sample <- sample(1:nrow(flfp), 5000, replace = F)

   temporary_flfp <- flfp[this_sample,]

   simple_model <- lm(data=temporary_flfp, patr_values~ religious + age_gr +edu)
this_intercept <-  tidy(simple_model)[1,2]
intercepts <- c(intercepts, this_intercept)

if ((counter %% 10) ==0){ # %% calculates the REMAINDER of division. So the remainder of x  divided by 10 is zero if x is some multiple of 10. In other words, this if statement will trigger every 10th iteration
  cat("\nIteration number is", counter) #A message will be trigger by this statement
}

} #end while loop


#functions should usually output something. we specify this using return()
return(temporary_flfp)


}
```

Running the new function

```
#we run the function code above, so that the function is known to R
and loaded in the memory (just like when we create any other new
object)

#then we call the function like any other
use_this_threshold <- mean(bind_rows( lapply(results_list,
data.frame))[,1])
my_results <- random_subset(threshold_to_beat = use_this_threshold)

#you can look in myresults as you wish. e.g. using View().
#you can also have your functions output a variety of different
information, e.g. in the form of a list
#e.g. results(list(subset=temporary_flfp,
iteration_number=counter))
```

Debugging

- Debugging your own code is a kind of dark art. There are many ways to do it.
- My method involves lots of calls to print() and cat() to isolate where the problem occurs.
- And a use of the magical function called browser()

```
# I expand this function, adding a bug as well

random_subset_new <- function(threshold_to_beat=0){
  #we define the arguments that the function will take
  #we can set a default if we wish by setting equals to our desired value.

  #we comment this out, because the user will enter this in as an argument!
#  threshold_to_beat <- mean(bind_rows( #lapply(results_list, data.frame))[,1])

this_intercept <- 0
intercepts <- vector("numeric",0)
counter <- 0 #we use this counter to count iterations within the while loop

while(this_intercept <=threshold_to_beat){
  counter <- counter+1#here the counter ticks forward

  this_sample <- sample(44671, 5000, replace = T)

   temporary_flfp <- flfp[this_sample,]

   simple_model <- lm(data=temporary_flfp, patr_values~ religious + age_gr +edu)
this_intercept <-  tidy(simple_model)[1,2]
intercepts <- c(intercepts, this_intercept)

if ((counter %% 10) ==0){ # %% calculates the REMAINDER of division. So the remainder of x  divided by 10 is zero if x is some multiple of 10. In other words, this if statement will trigger every 10th iteration
  cat("\nIteration number is", counter) #A message will be trigger by this statement
}

} #end while loop



return(summary(temporary_flfp), counter)


}
```

Running the bugged function

```
#we source/run the function code above, so that the
function is known to R

#then we call it like any other function
my_results <- random_subset_new(threshold_to_beat
=mean(bind_rows( lapply(results_list, data.frame))[,1])  )

#you can look in myresults as you wish.
#you can also have your functions output a variety of
different information, e.g. in the form of a list
#e.g. results(list(summary=summary(temporary_flfp),
iteration_number=counter))
```

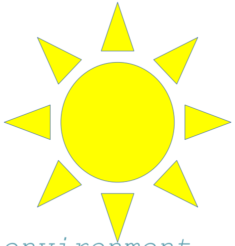
Help!

- We get different results every time. Why?
- Because we use a function called sample that draws a random sample.
- For debugging such cases, we need to first set a seed so that we get stable results and can debug the right instance.

```
set.seed(2) #we can set different seeds each time until we catch the
bug we want to fix
my_results <- random_subset_new(threshold_to_beat
=mean(bind_rows( lapply(results_list, data.frame))[,1])   )

#you can look in myresults as you wish.
#you can also have your functions output a variety of different
information, e.g. in the form of a list
#e.g. results(list(summary=summary(temporary_flfp),
iteration_number=counter))
```

Browser()



- browser() will stop the code within a loaded function, allowing you to go through line by line.
- enter browser() in the first line of the function after the ‘{’

```
# I expand this function, adding some bugs as well
random_subset_new <- function(threshold_to_beat=0){
browser() #this is like a stop point that will allow us to investigate within the function environment
this_intercept <- 0
intercepts <- vector("numeric",0)
counter <- 0

while(this_intercept <=threshold_to_beat){
  counter <- counter+1

  this_sample <- sample(44671:100000, 5000, replace = T)

   temporary_flfp <- flfp[this_sample,]

   simple_model <- lm(data=temporary_flfp, patr_values~ religious + age_gr +edu)
this_intercept <-  tidy(simple_model)[1,2]
intercepts <- c(intercepts, this_intercept)

if ((counter %% 10) ==0){
  cat("\nIteration number is", counter) #
}

}
return(summary(temporary_flfp), counter)
}
```

Run the function again

5 crucial debugging buttons appear above the console.

- ***Next*** takes you to the next line of code
- ***Step into*** takes you within the next lower function if one is called within the code
- ***Execute remainder*** finishes a current for or while loop, so you don't need to go through it line by line hundreds of times.
- ***Continue*** continues running the function again, exiting debug mode.
- ***Stop*** simply stops the function.
- Your turn: Try to find the two bugs I put in and fix them :-)

```
set.seed(2)

my_results <- random_subset_new(threshold_to_beat
=mean(bind_rows( lapply(results_list, data.frame))[,1])  )
```

lapply() functions

- The lapply() family of functions (lapply, sapply, mapply) is popular. You will find them online when searching for solutions.
- They actually run loops! But they are faster because the function is written in a faster underlying language (C)
- They can be quite handy. You can apply any existing function to the items in the loop or make your own

```
lapply(X=results_list, FUN = max) #it loops through items of a list in order OR for a dataframe, it loops through the columns.
## [[1]]
## [1] 0.2055364
##
## [[2]]
## [1] 0.2235323
##

#you can make your own function in the following way within lapply
lapply(X=results_list, FUN = function(x) abs(x[1] -simple_model$coefficients[1])) #x in this function will be each item in the list. or in this case, each row of coefficients. I take the first item, which is the intercepts... so I compare the new models' intercepts with the original simple model.
## [[1]]
## (Intercept)
##  0.07599292
##
## [[2]]
## (Intercept)
##  0.05799702
# mapply()
# * mapply() is like lapply, except that it excepts multiple lists (of the same size), which you can interact, combine in any way you like.
```