Lecture 3: Functions, Control Structures, and Other Useful Things

Intro to Data Science for Public Policy, Spring 2016

by Jeff Chen & Dan Hammer, Georgetown University McCourt School of Public Policy

Contents

Much of data science requires developing specialized code to handle the eccentricities of a dataset. Re-running blocks of code is required, often times on multiple data samples and subpopulations. It's simply not scalable to manually change variables and assumptions of the code everytime. In this lecture, we will extend EDA by writing custom functions and using control structures.

Control Structures

Variables are typically treated differently based on their quality and characteristics. In order to accomplish analytical and programming tasks, control structures are used to determine how a program will treat a given variable given conditions and parameters. In this section, we will cover two commonly used control structures: if...else statements and for loops.

If and If...Else Statement

If statements evaluate a logical statement, then execute a script based on whether the evaluated statement is true or false. If the statement is `TRUE`, then the code block is executed.

```
budget <- 450
if(budget > 400){
  #If statement true, run script goes here
  print("You're over budget. Cut back.")
}
```

```
## [1] "You're over budget. Cut back."
```

In cases where there are two or more choices, if...else statements would be appropriate. In addition to the `if()` statement, an `else` statement is included to handle cases where the logical statement is `FALSE`.

```
budget <- 399
if(budget >= 400){
  #If statement true, run script goes here
  print("You're over budget. Cut back.")
} else {
  #else, run script goes here
  print("You're under budget, but watch it.")
}
```

The complexity of these statements can be as simple as `if(x > 10){ print("Hello")}` more complex trees:

```
  age <- 23

  if(age <= 12){
      print("kid")
    } else if(age >12 && age <20) {
      print("teenager")
    } else if(age >=20 && age <65) {
      print("adult")
    } else{
      print("senior")
    }
```

```
## [1] "adult"
```

For-loops

Loops can be used to run the same statement of code multiple times for a list or index of values. Each iteration uses the same code, but may change certain parameters.

Let's take the following example. The code block essentially says "print values 1 through 5", where `i` is an *index value*. When executing the statement, R will push the first value in the sequence of 1:5 into the index (in this case, it's the number 1), then the code block in between the `{}` (curly brackets) will be executed, treating `i` as if it's the number 1. Upon executing the code without error, R will advance to the next value in the sequence and repeat the process until all values in the sequence have been completed.

```
  for(i in 1:5){
    print(i)
  }
```

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

We can do the same for a vector or list of values. In the example below, the vector `news` contains six terms. Using a for-loop, we can print out each word in the vector.

```
  news <- c("The","Dow","Is","Up","By","400pts")
  for(i in news){
    print(i)
  }
```

```
## [1] "The"
## [1] "Dow"
## [1] "Is"
## [1] "Up"
## [1] "By"
## [1] "400pts"
```

While

Whereas for loops require a range or list of values through which to iterate, `while()` statements keep iterating until some condition is met. The `while()` statement is formulated as follows:

```
while([condition is true]){

  [execute this statement]

}
```

A simple case may involve drawing a random value x from a normal distribution ($\mu = 1.0$, $\sigma = 0.5$) while x is greater than 0.01.

```
x <- 1
while(x > 0.01){
  x <- rnorm(1, 1, 0.5)
  print(x)
}
```

```
## [1] 0.9357534
## [1] 1.051594
## [1] 1.652722
## [1] 0.9763475
## [1] 0.1967486
## [1] 1.77621
## [1] 1.748971
## [1] 0.3345308
## [1] 0.4208325
## [1] 1.911306
## [1] 0.1797425
## [1] 1.252336
## [1] 1.29451
## [1] 0.4997665
## [1] 0.6255791
## [1] 0.2782497
## [1] 1.468796
## [1] 0.6083153
## [1] 1.677822
## [1] 0.3509417
## [1] 0.3477501
## [1] 1.102835
## [1] 0.6607486
## [1] 1.03413
## [1] 1.098683
## [1] 1.531102
## [1] 0.9294672
## [1] 1.44888
## [1] 0.9606352
## [1] 1.873981
## [1] 2.078476
## [1] 1.740123
## [1] 1.017969
## [1] 0.2455931
## [1] -0.5442862
```

```
print("done!")
```

```
## [1] "done!"
```

Exercises 3.1

1. [Easy] Write an if-else statement that classifies a number as positive number as "up" and a negative number as "down". Then, nest that if-else statement in a forloop to classify each record of a vector x from x_2 to x_{100} as "up"" or "down" relative to the preceding record. Use `table()` to tabulate the number of up days versus down days.

```
#define series
  n <- 500
  series <- sin((1:n)/100) + cos((1:n)/80)

#write if-else for i = 2
  temp <- c()
  if(series[2] >= series[2-1]){
      temp <- c(temp, "up")
    } else{
      temp <- c(temp, "down")
    }

#set empty vector
  temp <- c()

#loop through if-statement
  for(i in 2:length(series)){
    if(series[i] >= series[i-1]){
      temp <- c(temp, "up")
    } else{
      temp <- c(temp, "down")
    }
  }
  table(temp)
```

```
## temp
## down   up
##  267  232
```

2. [Medium] Fibonacci numbers are defined as $F_n = F_{n-1} + F_{n-2}$, or numbers that are defined as the sum of the preceding two numbers. For example, given an initial sequence of `0, 1`, the next five numbers are `1, 2, 3, 5, 8`. Using a `while()` loop, find the Fibonacci number that precedes 1,000,000.

```
#define variables
  n <- 0
  n0 <- 0
  n1 <- 1
  f <- 0
  s <- c()

#enter into loop
  while(f < 1000000){
    f <- n0 + n1
    n0 <- n1
    n1 <- f
    n <- n + 1
    s <- c(s, f)
  }
```

```
#get result in the (n-1)th position
  print(s[n-1])
```

```
## [1] 832040
```

3. [Hard] Often times, data files are stored in smaller chunks to save space and enhance searchability. In some cases, data is stored in daily chunks. The National Oceanic and Atmospheric Administration (NOAA) releases data every day on environmental and atmospheric conditions, including storms. Download the data for Lecture 3, get the list of the hail signature detection files in using `list.files()`, then write a loop to record the following measures in a data frame:

- month and year
- number of rows
- maximum hail size from the `maxsize` field

```
#retrieve files in working directory that start with the word "hail"
#be sure to open the file to examine its structure before opening it
  hail <- list.files(pattern = "^hail")

#create empty dataframe
  temp <- data.frame()
  for(rec in hail){
    df <- read.csv(rec, skip=2) #skip = 2 because the header of the data is located 2 rows down
    maxhail <- max(df$MAXSIZE)
    date <- regmatches(rec,regexpr("\\d{6}", rec))
    rows <- nrow(df)
    temp <- rbind(temp, data.frame(max.hail = maxhail, date = date, rows = rows))
  }

  print(temp)
```

```
##   max.hail   date    rows
## 1        4 201601   67782
## 2        4 201602  116668
## 3        4 201603  396600
## 4        4 201604  559897
## 5        4 201605 1119461
```

Functions

Functions are generalizable sets of code that can be used to calculate a single value, process an entire dataset, print graphs, among other things. A strong software engineering habit involves building narrowly defined functions and low-level functions that can be put together to do high-level tasks.

A typical function is constructed as follows. The function name is assigned to an object, followed by a list of parameters that will be used as inputs into the function, followed by the script that will be executed using the input parameters.

```
  function1 <- function(parameter1, ...){
    #Script goes here
    return([output goes here])
  }
```

To execute the function, we will simply need to pass call the function and pass inputs.

```
  function1(input1)
```

We can contextualize it by reconstructing a standard function such as the `mean()` method. `mean()` accepts accepts a vector `vec`, sums all values in `vec`, divides by the length of `vec`, then returns the result that is passed through `return()`.

```
#Create dataset
  n <- 1000
  df <- data.frame(id = 1:n,
                   x1 = rpois(n,3), x2 = rpois(n,10),
                   x3 = rpois(n,5), x4 = rpois(n,30),
                   x5 = rpois(n,1), x6 = rpois(n,1),
                   x7 = rpois(n,1), x8 = rpois(n,100))

#Set up Function
  mean.too <- function(vec){
      res <- sum(vec)/length(vec)
      return(res)
  }

#Execute
  mean.too(df$x1)
```

```
## [1] 2.972
```

But what if we wanted to obtain the mean for each row as opposed to each column? That can be achieved using the `rowMeans()` method, but we can also write a function to replicate the functionality. The function should:

- Accept the following parameters: `data` = the data frame, `start` = index value for the first column in range, `end` = index for the last column.
- steps:
  - Create an empty vector `output`
  - Loop through each row
  - Use the `mean.too()` function from above, calculate the row mean, append to `output`
  - Return `output` as result

```
#Write function
  row.means <- function(data, start, end){
    output <- c()
    for(i in 1:nrow(data)){
      output <- c(output, mean.too(data[i, start:end]))
    }
    return(output)
  }

#Run function
  df$means <- row.means(df, 2,9)
  head(df$means, 10)
```

```
##  [1] 16.750 19.375 18.125 18.375 19.625 18.250 18.625 17.125 20.125 21.125
```

Good habits for function writing

Notice how we rely on the `mean.too()` function that was previously built? There are some guiding principles that'll ensure that your code is clean, readable, and reusable:

1. Plan your code. Write or draw all the steps that are required to achieve your data processing requirements. Go through each line and cluster the steps into small, discrete modfules that can be relied upon independent of the initial context. For example, an entire data cleansing workflow should be broken into smaller functions rather than be converted into one long function.
2. Make your actions clear. Write your code in a manner that can be re-usable and interpreted by other humans. The code should be self-explanatory. Annotate every so often, but keeping in mind that clean production-level code, or code that will be used in servers for operational purposes, usually does not have annotation.
3. Pretty code is readable code. To make readable code, do:

- indent lines: add a tab to dependencies such as if-statements, loops, etc.
- add spaces before and after operators
- use `<-` instead of `=` except for when there are function calls

4. Name objects consistently. Name new data objects and functions should follow a naming convention, such as the Google R Guide.

Exercises 3.2

1. [Easy] Write a function that replicates the `unique()` method.

```
uniq <- function(vec){
  #create empty vector
    uniq <- c()
  #loop through
    for(i in vec){
      if(!(i %in% uniq)){
        uniq <- c(uniq, i)
      }
    }
  #return
    return(uniq)
}

#Try it out
#create a short series, repeat 10 times
  a <- rep(c(1,2,3), 10)
  uniq(a)
```

```
## [1] 1 2 3
```

2. [Medium] N-grams are a sequence of n-number of words in a sentence that are commonly relied upon for natural language processing and text analysis. Take the following sentence from the great statistician John Tukey: "Seek simplicity and distrust it." 2-grams from this sentence would include: `"seek simplicity", "simplicity and", "and distrust", "distrust it"`. A 1-gram would be a vector of all words parsed by spaces. Write a function that can return n-grams for any sentence.

```
  ngrams <- function(vec, delim, n){
    #vec = any string vector
    #delim = is what is used to parse
    #n = number of n-grams

    #split into list
      vec2 <- unlist(strsplit(vec, delim))
```

```
    #create placeholder then loop through each word
      grams <- c()
      for(k in n:length(vec2)){
        grams <- c(grams, paste(vec2[k-1], vec2[k]))
      }
      return(grams)
  }

  #Test it
    ngrams("Seek simplicity and distrust it.", " ", 2)
```

```
## [1] "Seek simplicity" "simplicity and"  "and distrust"    "distrust it."
```

What can you do with control structures?

Loops are a critical part of all parts of data science, enabling data cleaning, optimization, and automation. Loops are helpful when an function cannot be applied globally, meaning that each element, column, observation or iteration needs to be done on its own. For example, taking the sum of a random variable `x` can be done without looping as R is designed to operate with column-wise functionality. However, a moving average of 10 records would require a forloop.

Example: EIA Gasoline Spot Price Data

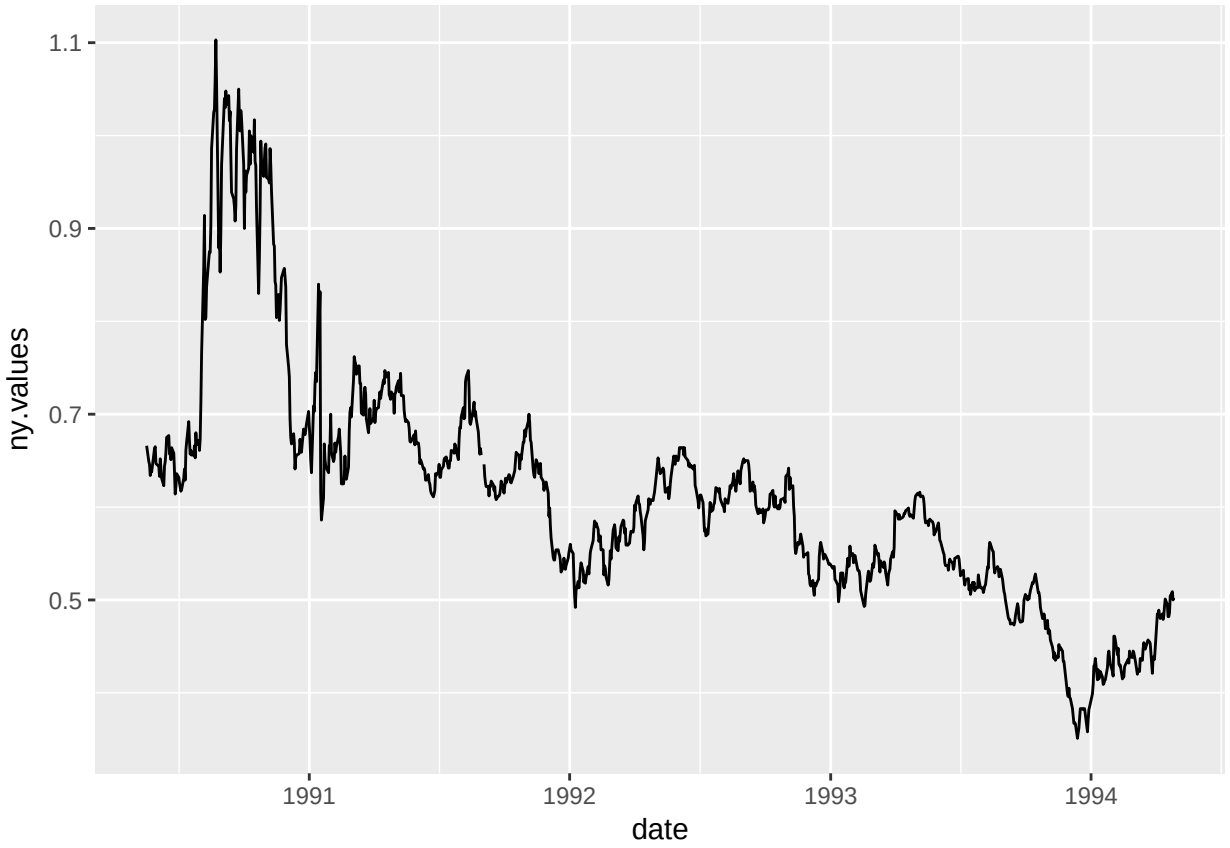
What if we had a time series dataset with a fair amount of random variability and swings in volume? This sounds very much like financial and economic data – it's often filled with noise. Let's take the US Energy Information Administration's spot price data, specifically the retail gasoline data.

```
#Call library to open XLS
  library(gdata)
  df <- read.xls("data/PET_PRI_SPT_S1_D.xls", sheet = 3, skip=2)

#Inspect and clean up date
  head(df,3)
```

```
##           Date
## 1 Jun 02, 1986
## 2 Jun 03, 1986
## 3 Jun 04, 1986
##   New.York.Harbor.Conventional.Gasoline.Regular.Spot.Price.FOB..Dollars.per.Gallon.
## 1                                                                             0.468
## 2                                                                             0.436
## 3                                                                             0.418
##   U.S..Gulf.Coast.Conventional.Gasoline.Regular.Spot.Price.FOB..Dollars.per.Gallon.
## 1                                                                             0.445
## 2                                                                             0.418
## 3                                                                             0.398
##    X
## 1 NA
## 2 NA
## 3 NA
```

```
  df <- df[1000:2000,1:3]
  colnames(df) <- c("date","ny.values","us.gulf.values")
  df$date <- as.Date(as.character(df$date), "%b %d, %Y")
```

```
#Plot the data
  library(ggplot2)
  ggplot(df, aes(date,ny.values)) + geom_line()
```



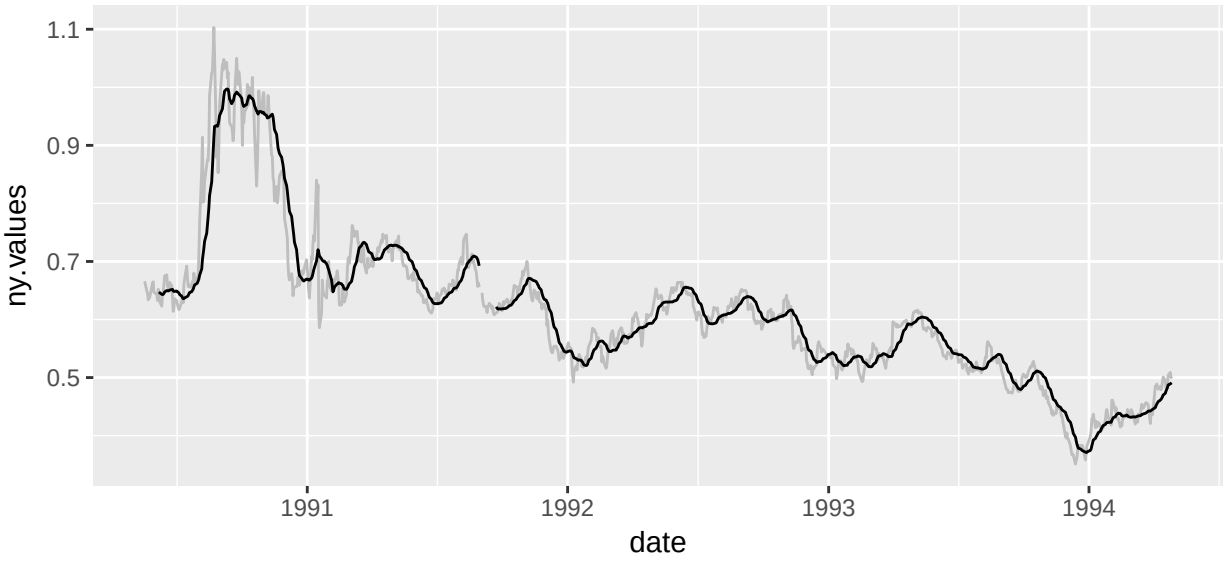
To smooth the series `ny.values` using a simple moving average, we can write a function to do the smoothing. Note, however, there is a `smooth()` is a pre-built smooth method available.

```
  moving <- function(vec, size){
    new.vec <- rep(NA, size-1)

    #Loop range from *size* to number of rows in vec minus *size*
    for(i in size:length(vec)){

        #Extract values of *x* from positions i-size to i
        extract <- vec[(i-size):i]

        #Calculate mean of *extract*, store to the ith value of *new*
        new.vec <- c(new.vec, mean(extract))
    }
    return(new.vec)
  }
```

Now we can test `moving()` using a 14-day window and plot the `ny.values` versus the 14-day moving average.

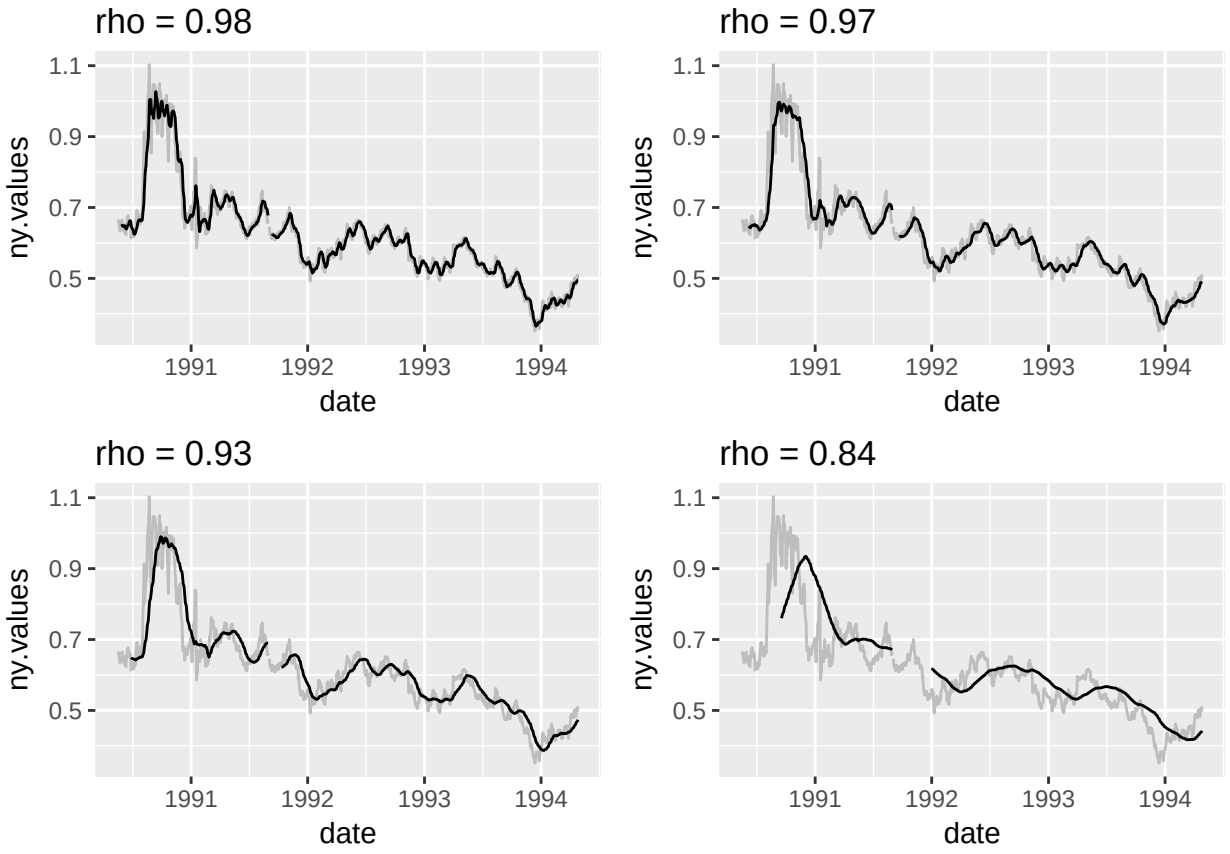
```
#Calculate
  df$new <- moving(df$ny.values, 14)

#Plot result
```

```
ggplot(df, aes(x = date, y = ny.values)) + geom_line(colour="grey") +
   geom_line(data = df, aes(x = date, y = new))
```



It's also possible to use loops within loops. What if we wanted to compare multiple window sizes, we can *nest* one loop inside another. In this case, looping through different potential window sizes helps with identifying the optimal window size.

```
#Vector of windows to be tested
    windows <- c(7, 14, 28, 84)

  #Outer loop (index value = *size*)
  for(size in windows){

    df$new <- moving(df$ny.values, size)

    #Calculate correlation
      cor_val <- round(cor(df$new, df$ny.values, use="complete.obs"), 2)

  #Plot graph
  g <- ggplot(df, aes(x = date, y = ny.values)) +
        geom_line(colour="grey") +
        geom_line(data = df, aes(x = date, y = new)) +
        ggtitle( paste("rho =", cor_val))
  assign(paste0("g",size), g)
}

#Compare
  library(gridExtra)
  grid.arrange(g7, g14, g28, g84, ncol=2)
```



Note that there are some pre-canned functions that can assist with smoothing; However, coding the function from scratch will provide you with greater flexibility to tackle the task at hand.