Introducing R and Rstudio to beginners who wish to learn data analytics

Sun Ho Song

10/30/2017

Introduction

Are you interested in data analytic? Are you willing to learn a new coding language? Then this is for you!

I will be writing this post to advertise the use of the R language to students who are interested in the data analytic. I would like to show them how convenient R is (in terms of writing the code) and emphasize its various usability. Yet most of the contents are the recap from Stat 133 course, there will be some other new materials that are not covered in the course. Adding to that, I will be also sharing my learning experience of R. In order for some students who already have coding backgrounds, I also put some distinctions between other coding languages and R. Hope that everyone can have a trigger on learning more about R after reading my post!

As a result, I will be introducing and sharing my experience regarding different usage of R:

- *Markdown syntax and R syntax*
- *Object and variable*
- *Vectors*
- *Manipulation of data frame*
- *Packages (dplyr and ggplot)*

Motivation

The primary reason that I am writing the post with this topic is because I want to help other students to get more familiar with data analytic. Some of them get worried about the coding part of the data science even before they start learning data analytic. Although it is true that coding skill is needed in order to clean, visualize, and interpret the data we have, that is not the core of data analytic. What is really important is the analysis skill itself, and I want to tell everyone that manipulation of data is also not too hard to learn.
Also, according to the instruction, it says, “especially for those of you who this is the first time using command-based software, analyzing (wrangling, reshaping, exploring) data, and working with (toy) projects in which you are more aware of the file-structure, you could write a post summarizing your learning experience in the last weeks.” Thus, I want to share my learning experience in the last weeks and show that R is not overwhelming, but very entertaining.

Background

One of University of California, Berkeley’s upper division statistics courses, Stat 133 (Concepts in Computing with Data), introduced me into the world of data analytic. Before taking this course, just like anyone else, I did not have any experience with coding, and I had zero knowledge in data science. I always wanted to learn data science, but I did not know where to start and how to learn it.
I am assuming that many other students feel just like me. So, I will be introducing some of the basic R and data science materials, so that they can feel more confident when they enter into the world of data science, and that they can use some skills that they learn from this post.

Explanations, Examples, and Discussions

1. Markdown syntax and R syntax

Body of the Rmd document in Rstudio is consisted of two parts: Markdown and R. To simply define each syntax, Mardown is really for writing narratives whereas R syntax is for writing the codes. Everything from here to top (except for title, author, date, and output lines) is an example of Markdown syntax. Markdown is not only limited to writing down the narratives, but you can put the headings, images, links, and tables. I will show some of the features as I go along.

On the other hand, the example of R syntax would be:

```r
# Printing the word "Hi"
"Hi"
```

```
## [1] "Hi"
```

The first rectangular box above is called “code chunk.” This is where we can write our code. Generating code chunk is easy. All you need to do is just click “Insert” from the toolbar then choose “R” in RStudio.

My learning experience:
According to my learning experience, I had a hard time differentiating between Markdown and R. I could not understand how there could be two types of coding syntax in just one file. Also, the fact that the functionality of each syntax differs added more confusion. Since most of times, only one syntax is used while coding, this division between Markdown and R would be also confusing for many programmers who did not have any exposure to R. Regardless of past experience with coding, constant exposure to the RStudio environment and reading some learning materials help to distinguish between the two.

New material that we did not cover in the class or in the assignments:
Horizontal Rule
You can use three or more of hyphens, asterisks, or underscores to make horizontal lines!

This is how

---

you make

---

horizontal lines!

To finish up this section, here is some useful resource.

Useful resource: Markdown Cheat Sheet

2. Object and Variable

Not only Rstudio is amazing because it provides both Markdown syntax and R syntax, but also because the functionality of R is very cool!

Let’s see the example:

```r
# Calculating 3 + 5
3 + 5
```

```
## [1] 8
```

The code chunk above shows that we can use Rstudio and R as a calculator. Isn’t this beautiful?

What is cooler is this: R can actually memorize the value we got! This is what we call a process of assigning a value to the **variable** or an **object**. Let’s see the code below.

```r
# Saving the value into the variable
eight <- 3 + 5
```

Oops! Nothing happened! Well this is because we told R to memorize the value; we did not tell it to tell me what that value is. R basically memorized `3 + 5` as the word `eight`. This is the steps which you can use to assign the value to the variable.

1. Put the name of the variable first
2. Put either `<-` or ‘=’
3. Put the value(s) or computations that you want to solve

Caution: the name of the variable cannot begin with numbers and “-”, and it cannot have space or comma inside.

If I call the variable `eight`, it gives,

```r
# Calling the variable
eight
```

```
## [1] 8
```

8, which is the result of `3 + 5`. Another way of checking whether R correctly memorized the value is by looking at the “Environment” pane in RStudio. We see that under the Values, we have variable `eight` assigned as 8.
You might think that this process is redundant. You might think “well, why not just type 8 instead of typing 5 letters of”eight“?” If assigning value to the variable seems arduous, check this out!

```r
# Calculating such big numbers!
1238214803140 + 234834832040423
```

```
## [1] 2.36073e+14
```

Even if you got the correct result for the equation above, how would you memorize the result? Not only it is impossible to memorize the answer for that equation, but also it is very time-consuming to write down the digits. Instead we can assign the result to the variable.

```
# Assigning big number to the variable is convenient
long_result <- 1238214803140 + 234834832040423
```

And whenever we need the result of that equation, we can just simply call `long_result`. Do you now see the power of assigning a value to the variable?

Adding to this, not only you can assign numerical values to the variable, but also you can assign characters and logical. Logical simply means whether the value if TRUE or FALSE and it is binary.

```
# Assigning characters to the variable
character <- "You can assign character values too!"
```

```
# Assigning logical value to the variable
bool <- TRUE
```

To check the type of the variable, you can use the `typeof()` function.

```
# Checking the type of the variable
typeof(eight)
```

```
## [1] "double"
```

```
typeof(character)
```

```
## [1] "character"
```

```
typeof(bool)
```

```
## [1] "logical"
```

The results from above code chunk are “double”, “character”, and “logical”. There are two modes within **numeric**: **double** and **integer**. **Double** means the number has decimal places and **integer** means that the number is integer value. But wait, 8 is not decimal?! Well, this is the weird part of R. If you do not put “L” behind the number, R regards the number as having decimal places.
If you want `eight` to show its type as integer, you can simply use the function called `as.integer()` to change its type.

```
# Changing double to integer
int_eight <- as.integer(eight)
```

Now you can see that `int_eight` has the value of 8L, which means its type is integer. To double check, let’s run the code chunk below.

```
# Finding out the type of the variable
typeof(int_eight)
```

```
## [1] "integer"
```

My learning experience:
To the people who are new to the programming, assigning value to the variable may be confusing, which happened to me too. However, from my personal experience, what was more confusing was R’s notion of ‘integer’. However, often times, we just use the decimal format.

New material that we did not cover in the class or in the assignments:
According to my research, both Java and Python treat numbers without decimal points as integers. However, R only treats the number as integer if that number has the word “L” at the end. Thus, what is integer in either Python or Java is considered as double. But, there are not many times that we use integer in R. To make things more simple and be more convenient, we just put everything into decimal format.

3. Vectors and some visualizations

Vector is a sequence of data elements. So far, we only focused on saving and calling one element, but now we will learn how to put multiple values in one variable.

Here is an example of a vector:

```
# Saving six different numerical values to the variable
vector <- c(1, 2, 3, 4, 5, 6)
```

When we assigned a value to the variable, we did not need any brackets because it was a single value. However, because we now want to put multiple values to one variable, we need to group them by calling `c()`. Here, c stands for combine. One thing to keep in mind is that vectors must have atomic structure, meaning that only the same type of values can go into a vector. If you try to put values with different types, R will give an error.

What if you want to pull out a value from the vector? Here, you have to use a new type of brackets. Let’s say that you want to pull out 5 from vector variable.

```
# Pulling out fifth element of the vector
vector[5]
```

```
## [1] 5
```

This method is definitely one way. How this works is that, “vector” saves 1 in its first position, 2 in its second position, 3 in its third position, and so on. Thus, if we want to pull out 5, all we need to do is calling out its position. In this case, it was 5.

Variables are not just there to pull out the values. But, we can use it to perform computation using the variables.

```
# Adding 1 to all the elements of the vector
vector + 1
```

```
## [1] 2 3 4 5 6 7
```

What I did just now is that I added 1 to each component of the vector. This process is called recycling, but let’s not be stressed about the term.

```
# We can also perform computation between the variables as well.
vector[2] * eight
```

```
## [1] 16
```

My learning experience:
As a student who learned a little about Java in the high school, the notions of counting and vector were quite confusing for me. What is interesting is that counting the position is different for R from other programming languages. R starts its counting from 1 whereas many other programming languages counts starting from 0.
Also, many other programming languages do not have what is called “vector”. Vector is simply a set of values assigned to one variable. Other programming languages do not have vector, but have “list” and “array”. “Array” operates in the same way as the vector does.

New material that we did not cover in the class or in the assignments:
What would happen if we try to pull out a value at the position that is beyond the length of the vector? Would R generate error? Let’s see what happens!

```
# Calling a value at the position 8, which does not exist
vector[8]
```

```
## [1] NA
```

Wait, it gave us `NA`! Does this mean that we had a value called `NA` at position 8?
The answer is no. Unlike other programming languages that generate an error when we try to access the value at the position that is beyond the length of the array, R kindly instructs that we do not have any value at that position. `NA` simply means there is no value in that position; what is important in this part is that R does not generate an error.

4. Manipulation of data frame

In this part, we will learn the basic operations of the data frame. But before that, what is a data frame? If what is consisted of single values is called vector, data frame is where all the vectors are combined. Thus, data frame has many columns with each column having different or the same data types.
Since we are going to learn the manipulation of data frame, not the creation of data frame, you can ignore the following code.
Importing the data in R by `read.csv()` function:

```
# Calling the data set from outside resource
dat <- read.csv('nba2017-players.csv', stringsAsFactors = FALSE)
```

What I did here was that I called a data frame with statistics of different NBA players and I saved it in the variable called `dat`. Since there are too many rows and columns in dat, let’s reduce the size so that it is easier for us to manipulate.

```
# Reducing the size of the `dat` by pulling out certain rows and columns
reduced_dat <- dat[1:5, 1:5]
reduced_dat
```

```
##              player team position height weight
## 1        Al Horford  BOS        C     82    245
## 2      Amir Johnson  BOS       PF     81    240
## 3     Avery Bradley  BOS       SG     74    180
## 4 Demetrius Jackson  BOS       PG     73    201
## 5       Gerald Green  BOS       SF     79    205
```

Useful Resource : Here is the cheat sheet of the `readr` package that you can take a closer look.

Selecting elements

How do we select elements from the data frame? Well, the process is very similar to what we did for the vectors! Just like vectors, we put brackets after the name of the variable. However, inside the brackets, the space is divided into two. The first space designates row and the second space is for the column What divides two spaces is comma. Let’s see the example.

```
#If you want to get the value in 2nd column and 3rd row,
reduced_dat[3, 2]
```

```
## [1] "BOS"
```

What if you want to get the whole column or whole row?

```
#This gets the first row of the data frame
reduced_dat[1, ]
```

```
##       player team position height weight
## 1 Al Horford  BOS        C     82    245
```

```
#This gets the first column of the data frame
reduced_dat[ ,1]
```

```
## [1] "Al Horford"        "Amir Johnson"      "Avery Bradley"
## [4] "Demetrius Jackson" "Gerald Green"
```

Do you see how this works? If you want to print values just for specific columns, then you have to leave the row part blank. And vice versa for the rows as well.
What is interesting is that when we are getting the values for the column, the result is a vector; however, the result of pulling out a row is still the data frame. The reason why is that column itself is already a vector, which means it has atomic structure. However, the row consists of different data types, so it must remain as a data table.

My learning experience:
People with zero coding background would have no problem with this. However, for the people who already have one would get confused about the order of column and the row. For example, Python's functions deal with columns first, then the user will use another function to find the corresponding value in the specific row. However, for R, row comes first and the column comes second. Thus, the users must be extremely careful with this. Also, the fact that R starts its index with 1 is another confusing factor that I mentioned above.
As I mentioned before, since I had some Java experience, this was a confusing part for me. However, I could be ambidextrous now since I practiced many times. Remember: the only way to overcome confusion is through practice!
Other then that, selecting elements was very straightforward because this is very similar to what we did for the vectors.

Last but not least, you can use logical method to extract certain data sets.

```
# Return row(s) where the player's position is Center.
reduced_dat[reduced_dat$position == "C", ]
```

```
##       player team position height weight
## 1 Al Horford  BOS        C     82    245
```

New material that we did not cover in the class or in the assignments:
During the class or lab, we did not really cover whether the result of our selection of columns is vector or data frame. Let's have a time to distinguish whether the result is vector or data frame.

```
#This gets the first row of the data frame
reduced_dat[1, ]
```

```
##       player team position height weight
## 1 Al Horford  BOS        C     82    245
```

As mentioned before, this gives us data frame because each row consists of different type of values.

```
#This gets the first column of the data frame
reduced_dat[ ,1]
```

```
## [1] "Al Horford"        "Amir Johnson"      "Avery Bradley"
## [4] "Demetrius Jackson" "Gerald Green"
```

On the other hand, this gives us a vector because we are just pulling one column which basically is a vector.

Let's now consider this case:

```
# Getting first, second, and third column of the data frame
reduced_dat[ , 1:3]
```

```
##              player team position
## 1        Al Horford  BOS        C
## 2      Amir Johnson  BOS       PF
## 3     Avery Bradley  BOS       SG
## 4 Demetrius Jackson  BOS       PG
## 5      Gerald Green  BOS       SF
```

The result of this is data frame because it consists of three vectors each with different data types.
Do you now see how this works?

Adding a column

There are two ways of adding a column. One is by directly manipulating the data frame and the second method is by using the function. Let's create a vector that we want to add first.

```
blood_type <- c("O", "A", "A", "B", "AB")
```

The first method is by using the `$` sign.

1. Put the dollar sign after the name of the data frame
2. Put the name that you want the column to be after the dollar sign
3. Put either `<-` or `=`
4. Put the values or the variable that you want to add as a column

Caution: the length of the vector should match to the number of rows of the data frame.

```
# Using the dollar notation
reduced_dat$blood_type <- blood_type
```

The second method is by using the function called `cbind()`. This is an easier method because using function is more convenient!

```
# Using the cbind() function
reduced_dat <- cbind(reduced_dat, blood_type)
```

To interpret the code chunk above, we are putting the original reduced_dat data frame and blood_type together to create a new data frame. Once we made one by binding, we will name the new data frame as `reduced_dat`. Thus we are not really creating a new data frame variable, but we are updating the one we had before.

Deleting a column

If we know how to add a new column to the data frame, then we should also know how to delete the column in the data frame. This is how you can delete a column:

```
# Deleting the blood_type column
reduced_dat$blood_type <- NULL
```

Wait, but what is "NULL"? NULL basically means "nothing" in the general context. So what the code chunk above does is setting the column as nothing, which is same thing as getting rid of that column.

Moving a column

What if you do not like how the columns are arranged? That can possibly happen. For example, from our reduced_dat, we may want to display height and weight columns prior to team and position columns. The way you do this is by making a vector that consists of the names of the columns in the sequence that you want to put. Here is the example:

```
# First, we make a vector
reordered_names <- c("player", "height", "weight", "team", "position")

# Then, we update our original variable
reduced_dat <- reduced_dat[ ,reordered_names]
reduced_dat
```

```
##              player height weight team position
## 1        Al Horford     82    245  BOS        C
## 2      Amir Johnson     81    240  BOS       PF
## 3     Avery Bradley     74    180  BOS       SG
## 4 Demetrius Jackson     73    201  BOS       PG
## 5      Gerald Green     79    205  BOS       SF
```

Wow, the columns are now rearranged!

Transforming a column

This is the last part of the manipulating data frame. We are going to learn about how to transform the data in the column. This is desirable when we realized that we want to change the unit of the data we have in such column. For example, the weight's unit is now in pounds. However, what if we want to change the unit into kilograms? To make things simple, we will say that pounds is measured twice as kilogram is measured. Thus, 1 pound = 0.5 kg.

```
# This is the first way of doing it
reduced_dat$weight <- reduced_dat$weight * 0.5

# This is the second way of doing it
reduced_dat[ , "weight"] <- reduced_dat[ , "weight"] * 0.5
```

Two lines of codes above do the same thing. There is actually a third way of doing this, which is using the function called `transform()`. However, I will not introduce the use of this function because introducing too many functions at once would overwhelm you. I will try to keep everything as simple as possible for you to understand the core idea since you are new to programming.

My learning experience:
Although this section requires coding, this possibly is a new material for both beginners and experienced programmers because there are not many opportunities for people to manipulate data tables using other programming languages. Manipulation of data frame is a beauty of R and RStuido. This is another uniqueness that is particularly done by data scientists only. This part is what I mentioned as a cleaning part of the data science. With the given data sets, we have to clean the data so that it can be used more clearly and more accurately for later steps of the data analytic such as visualization and interpretation.
When I first learned about this, I felt that the concept was easy to absorb, but it was the coding part that needed practice. However, once I became familiar with the coding, it became my second nature.

5. Packages (dplyr and ggplot)

Anything related to data analytic can be done in RStudio because the built-in functions of the RStudio is pretty powerful. However, what is even more interesting in RStuido is that we can download and use what is called `packages`. Here is the easy way of understanding packages.
Let's say you are making music in Garage Band. Well, to make one, you may only use the basic sounds that are provided by the Garage Band

(the built-in features). However, what you also can do is that you can download music online and use that music to make your own one! That is what `packages` in R do. In this section, we will learn two different packages namely, `dplyr` and `ggplot`.

dplyr

`dplyr` is a package that makes the manipulation of data frame much easier. The basic dplyr verbs that we are going to cover in this section are:

- `slice()`, `filter()`, `select()`
- `mutate()`
- `arrange()`
- `summarise()`
- `group_by()`

The first few verbs in the list perform in the same way as the functions that we learned above in the data manipulation part. However, `summarise()` and `group_by()` are the beauty of the `dplyr` package. Before diving into these two verbs, let's start with `slice()`, `filter()`, and `select()` for now because they are the basics.
Before going further, lets call the package using the `library()` function.

```
# Calling "dplyr" package from the library
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
```

slice(), filter(), select()

`slice()` is just the same thing as selecting the elements; however, the only difference is that `slice()` is for extracting the elements by the rows. Let's see the example:

```
# Extract the elements by the rows
slice(reduced_dat, 2:3)
```

```
## # A tibble: 2 x 5
##          player height weight  team position
##           <chr>  <int>  <dbl> <chr>    <chr>
## 1  Amir Johnson     81     60   BOS        PF
## 2 Avery Bradley     74     45   BOS        SG
```

What I did above is that I extracted the second and third row of the whole data set. Similarly, in order to extract the data set by the columns, we use `select()`.

```
# Extract the elements by the columns
select(reduced_dat, 2:3)
```

```
##   height weight
## 1     82  61.25
## 2     81  60.00
## 3     74  45.00
## 4     73  50.25
## 5     79  51.25
```

`filter()` utilizes the logical method to extract certain data values.

```
# Use logical to extract certain data values
filter(reduced_dat, position == "C")
```

```
##       player height weight team position
## 1 Al Horford     82  61.25  BOS        C
```

However, the difference between the filter function and the method that we used manually above is that filter function does not need quotation marks when referring to the name of the column, which makes it more convenient to use.

mutate()

The mutate function acts just like the add column method that we used above.

```
# Adding blood_type vector to the reduced_dat data table
mutate(reduced_dat, "BT" = blood_type)
```

```
##              player height weight team position BT
## 1        Al Horford     82  61.25  BOS        C  O
## 2      Amir Johnson     81  60.00  BOS       PF  A
## 3     Avery Bradley     74  45.00  BOS       SG  A
## 4 Demetrius Jackson     73  50.25  BOS       PG  B
## 5      Gerald Green     79  51.25  BOS       SF AB
```

Not only that we can add a new vector into the data table, what we can do is that we can also interact with the values that are in the data table already. For example:

```
# Add another column to the existing data table that displays the weight of the players in kilograms
mutate(reduced_dat, weight_in_kilo = weight * 0.5)
```

```
##              player height weight team position weight_in_kilo
## 1        Al Horford     82  61.25  BOS        C         30.625
## 2      Amir Johnson     81  60.00  BOS       PF         30.000
## 3     Avery Bradley     74  45.00  BOS       SG         22.500
## 4 Demetrius Jackson     73  50.25  BOS       PG         25.125
## 5      Gerald Green     79  51.25  BOS       SF         25.625
```

arrange()

`arrange()` is the function that rearranges the rows. The rearrangements can be done by the columns. For example, let's try to rearrange the rows by the height of the players.

```
# Rearrangement by height in ascending order
arrange(reduced_dat, height)
```

```
##              player height weight team position
## 1 Demetrius Jackson     73  50.25  BOS       PG
## 2     Avery Bradley     74  45.00  BOS       SG
## 3      Gerald Green     79  51.25  BOS       SF
## 4      Amir Johnson     81  60.00  BOS       PF
## 5        Al Horford     82  61.25  BOS        C
```

However, the rearrangement seems to be in ascending order. What if we want it to be in descending order? Then we can put additional words.

```
# Rearrangement by height in descending order
arrange(reduced_dat, desc(height))
```

```
##              player height weight team position
## 1        Al Horford     82  61.25  BOS        C
## 2      Amir Johnson     81  60.00  BOS       PF
## 3      Gerald Green     79  51.25  BOS       SF
## 4     Avery Bradley     74  45.00  BOS       SG
## 5 Demetrius Jackson     73  50.25  BOS       PG
```

`desc()` simply changes the values into descending order.

One caution that the user has to take is that the methods that I used so far do not update the variable. So, if you want to have updated variables, make sure that you assign the results to the respective variable.

summarise()

What `summarise()` does is it gives the statistics for the given data. To give you an example, it can be used to find the mean height of the data set.

```
# Finding the mean height of the players
summarise(reduced_dat, mean(height))
```

```
##   mean(height)
## 1         77.8
```

Right now, it does not seem to be very powerful since the same result can be computed using `mean()` function. However, what makes it powerful is when it is used with `group_by()` function.

group_by()

What if you want to find the average height of each position? That does not seem to be possible using only `summarize()` function. This is where `group_by()` comes in.

```
# Finding the mean height of each position
summarise(group_by(reduced_dat, position), mean(height))
```

```
## # A tibble: 5 x 2
##   position `mean(height)`
##      <chr>          <dbl>
## 1        C             82
## 2       PF             81
## 3       PG             73
## 4       SF             79
## 5       SG             74
```

So what it does is, it groups the data frame into different positions. Among them, it will find the mean height of each position. As a result of the combination of two expressions, we can get more sophisticated results.

My learning experience:
When I first encountered `dplyr`, it was very interesting to see that R can use packages to do things in a simpler way. `dplyr` was not a very difficult package for me to learn because I felt it was very similar to the manipulation of data process that we did above. I had some difficulty understanding the functionality of `summarise()` and `group_by()`, however, again, practice made me perfect.

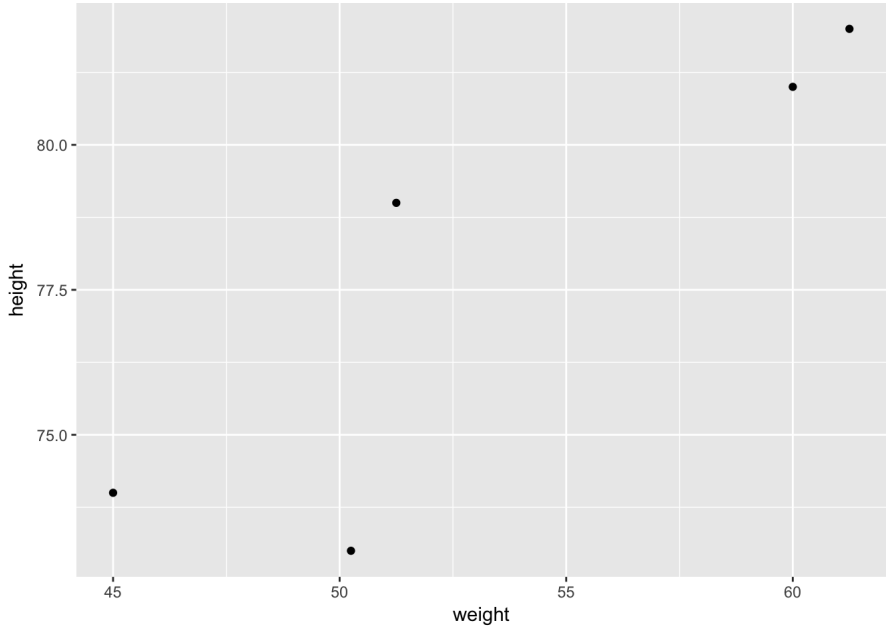
ggplot2

ggplot2 is a package that helps drawing different kinds of graphs. The way to draw a graph is very simple. Before doing that, let's call ggplot2 package from the library.

```
# Calling 'ggplot2' package from the library
library(ggplot2)
```

Here is the graph that is drawn with ggplot2.

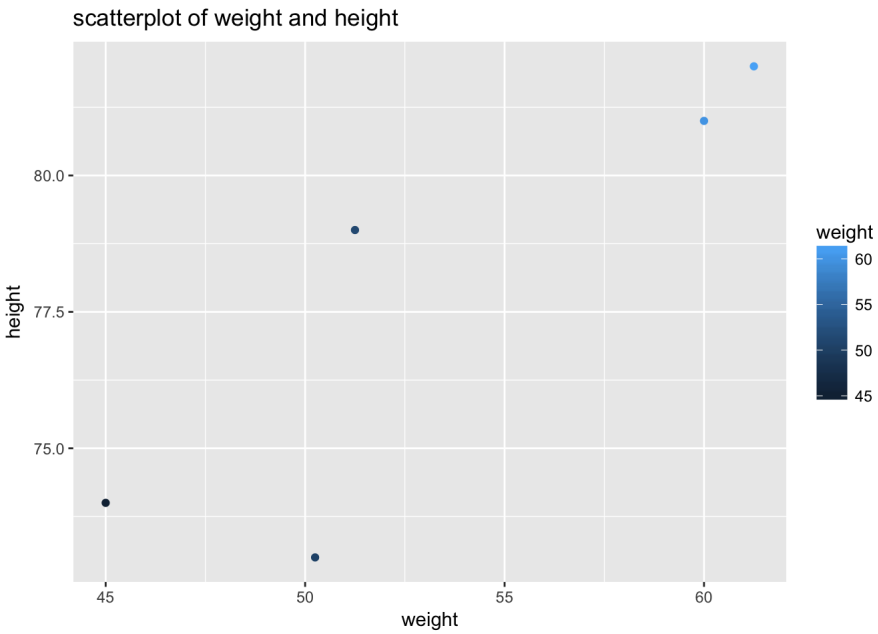
```
# Drawing scatterplot of weight and height
ggplot(data = reduced_dat, aes(x = weight, y = height)) + geom_point()
```



So, what goes into `ggplot()` is first, the data set we have, which is reduced_dat in this case. The second thing that goes in is `aes()`. This basically means the aesthetic of the graph. The aesthetic includes not only the color of the dots, but also the axis labels, and axis data. After `ggplot()` part is done, we add `geom_point()` in order to actually draw the scatterplot diagram. If we were to draw histogram, we would have used `geom_histogram()` and for other graphs, we would have used other functions.

Let's try to put more aesthetics.

```
# Drawing scatterplot of weight and height with more aesthetics
ggplot(data = reduced_dat, aes(x = weight, y = height)) + geom_point(aes(color = weight)) + ggtitle("scatterplot o
f weight and height")
```

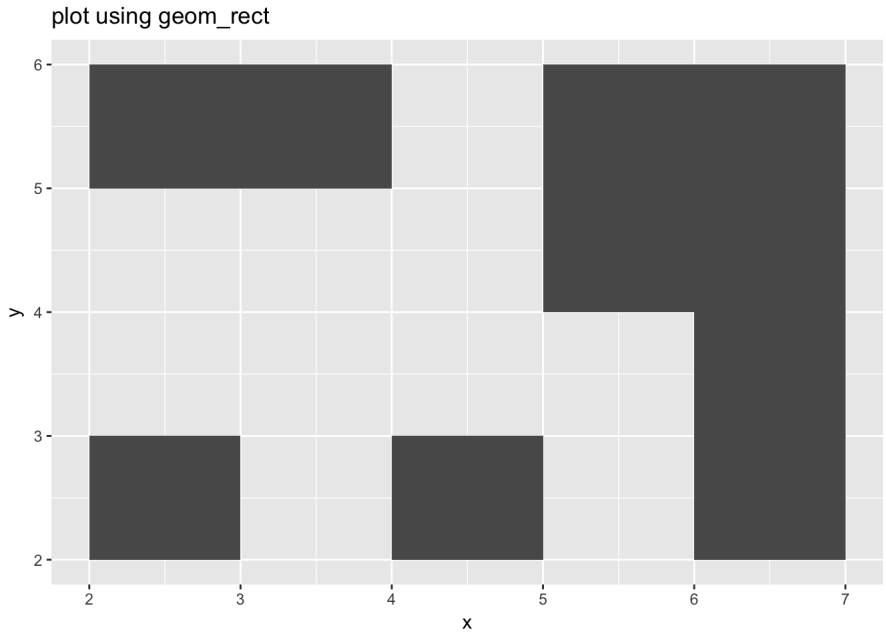


What I did here was that I put colors according to the value of the weights by writing `color = weight` inside `geom_point()`. Also, I put the title on the graph by adding `ggtitle()` which is a function that adds title to the graph.

My learning experience:
Unlike `dplyr`, learning `ggplot2` was more time-consuming and difficult. This is because although the package uses R, the way of writing code is totally different. `ggplot2` has its own way of writing the code. It seems like before getting familiar with such way of writing the code, manipulating graphs with ggplot is not very easy. However, though some difficulty follows, `ggplot2` has many functionalities that are not only limited to drawing basic graphs. You can draw rectangles, star diagrams, and many more. Thus, this package is very useful for the programmers.

New material that we did not cover in the class or in the assignments:
As I mentioned above, you can draw a rectangle using `ggplot2`. Let's try this!

```
# Drawing several rectangles using geom_rect()
d = data.frame(
  x_val = c(2, 4, 2, 6, 5),
  x_val2 = c(3, 5, 4, 7, 7),
  y_val = c(2, 2, 5, 2, 4),
  y_val2 = c(3, 3, 6, 4, 6)
  )
  ggplot() +
  xlab("x") +
  ylab("y") +
  geom_rect(
  data = d,
  mapping = aes(
  xmin = x_val,
  xmax = x_val2,
  ymin = y_val,
  ymax = y_val2
  )
  ) +
  ggtitle("plot using geom_rect")
```



All you need to know from this code is aesthetics that go inside mapping. What I did was I put the values for the xmin, xmax, ymin, and ymax (which basically work as coordinates) in order to draw rectangles. Other parts of the coding are very similar to what we learned so far.

Useful Resource : This is the cheat sheet of the `dplyr` package that you can take a closer look; and this is the cheat sheet of the `ggplot` package

Conclusions (with take home message)

I wrote this post to advertise the use of the R language to the students who are interested in the data analytic. I wanted to show them how convenient R is in terms of writing the code and emphasize its various usability using the materials I learned in Stat 133 course. However, I also added some new materials that were not covered in the class, which makes this post not just a mere review for the students who already took Stat 133 course, but rather an opportunity to get some new knowledge. For some students who already have coding backgrounds, but no R background, I also put some distinctions between other coding languages and R.

The primary reason that I wrote this post with this topic was because I wanted to help other students to get more familiar with data analytic and to be more interested in data analytic. Through sharing my personal experience with R, I wanted to tell them learning data analytic and R is not overwhelming, but rather it can be a fun experience.

Take home message :
As you have seen, using R is not hard. Everyone can learn it and use it. The difference between people who can use R to analyze data and people who cannot use R is that the latter group of people just have not have any exposure to R, thus they do not have any idea where to start. Once they get started, they will be surprised at how easy it is to use R, how convenient R is in terms of writing the code, and how variously it can be used. Try it now, and become a data analyst now!

References

- http://www.r-tutor.com/r-introduction/basic-data-types/integer
- https://github.com/ucb-stat133/stat133-fall-2017/blob/master/tutorials/02-intro-to-Rmd-files.md
- https://www.tutorialspoint.com/python/python_numbers.htm
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
- https://github.com/ucb-stat133/stat133-fall-2017/blob/master/tutorials/03-intro-to-vectors.md
- https://docs.scipy.org/doc/numpy-1.13.0/user/basics.creation.html
- https://stackoverflow.com/questions/11488820/why-use-c-to-define-vector
- https://docs.scipy.org/doc/numpy-1.13.0/user/basics.indexing.html
- https://github.com/ucb-stat133/stat133-fall-2017/blob/master/labs/lab04-data-frame-basics.md
- https://docs.google.com/presentation/d/e/2PACX-1vTGKt6asJIPfSXbuO5Jn1qtAOTQbOcWmE0TVXr67z7DOkOUHPj43Vi1Q7hqw-xYTy3pzD2985H9lt0T/pub?start=false&loop=false&delayms=3000&slide=id.g165e8b088b_0_0
- https://github.com/ucb-stat133/stat133-fall-2017/blob/master/tutorials/05-intro-to-dplyr.md
- https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
- http://sape.inf.usi.ch/quick-reference/ggplot2/geom_rect