**Data Science:   
R Basics**

**with Prof. Rafael Irizarry**

**James Strayer**Course Notes and Screenshots

HarvardX  
Data Science: R Basics

**Data Science: R Basics**

with Prof. Rafael Irizarry

**Table of Contents**

[1. R Basics, Functions and Data Type 2](#_Toc17044141)

[1.1. Motivation 2](#_Toc17044142)

[1.2. R basics Assessment 3](#_Toc17044143)

[1.2.1. R Basic 3](#_Toc17044144)

[1.2.2. Functions 3](#_Toc17044145)

[1.3. Data Types 5](#_Toc17044146)

[2. Vectors, Sorting 8](#_Toc17044147)

[2.1. Vectors 8](#_Toc17044148)

[2.1.1. Vectors 8](#_Toc17044149)

[2.1.2. Vectors coercion 9](#_Toc17044150)

[2.2. Vectors sorting 10](#_Toc17044151)

[2.3. Vector Arithmetic 13](#_Toc17044152)

[3. Indexing, Data Wrangling, Plots 14](#_Toc17044153)

[3.1. Indexing 14](#_Toc17044154)

[3.1.1. Indexing 14](#_Toc17044155)

[3.1.2. Indexing function 15](#_Toc17044156)

[3.2. Basic Data Wrangling 16](#_Toc17044157)

[3.2.1. Basic Data Wrangling 16](#_Toc17044158)

[3.2.2. Creating Data Frames 19](#_Toc17044159)

[3.3. Basic Plots 20](#_Toc17044160)

[4. Programming Basics 23](#_Toc17044161)

[4.1. Introduction to Programming in R 23](#_Toc17044162)

[4.2. Conditionals 23](#_Toc17044163)

[4.3. Functions 25](#_Toc17044164)

[4.4. For Loops 26](#_Toc17044165)

[4.4.1. For Loops 26](#_Toc17044166)

[4.4.2. Other Functions 27](#_Toc17044167)

**List of Figures**

**Aucune entrée de table d'illustration n'a été trouvée.**

1. R Basics, Functions and Data Type
   1. Motivation

R is not a programming language like others, it was not developed by software developer but by statisticians and data analyst as a interactive environment for data analysis. You can save your code as scripts that you can easily execute at any moment. They serve as record of your analysis, that is a key feature of reproducibility.

R is free and open source, that means that all the code of it is accessible, and his multiplatform (Windows, Mac, Linux) and data objects a share without any problem through those platforms.

As any popular open source program, R has a large community of developer that shares knowledge, thanks to that there is a high quantity of resources on how R works.

Interactive data analysis usual occurs in the R console, the R console execute commands as your type them, one of the best way to gain access to a R console is to download R on computer here https://cran.r-project.org/.

When you type a command into the console and hit enter, the expression is evaluated. The real advantage of R resides in his scripting capability, to access it you should use an IDE (Interactive Development Environment). R studios is the IDE for R, it comes with an R console, a window for your data, a window for your script and much more.

R has Pleiades of tools to help you in your data analysis, but the fresh install of R studios provided but a small portion of those tools. We call this fresh install Base R , the extra tools comes from addons from developers, there is a lot of those functionality that are available in shared repository like CRAN or others, like Github. However, because we don’t need all the functionality available for R (it would make the install of R studios realllllly long) we access those functionalities through Packages. Packages are easy to install through R, for example to install the package the package numpy we would type:

*Install.packages(“numpy”)*

IDE usually provides a graphical interface that facilitate the installation of packages. Once the package is installed, you can use it in your code by typing:

*Library(numpy)*

Once the packages are installed and loaded in your code you don’t need to reload it.

* 1. R basics Assessment
     1. R Basic

Before starting any data manipulation, we need to talk about object. Imagine a high school student ask you for help solving quadratic equations of the form:

We know that the solution to this equation is:

One of the advantages of programming is that we can defined variable to keep those expression general. So, if we are asked to solve the equation:

x^2+x-1=0

We can define a=1, b=1 and c=-1, in R we do that by:

a<-1

b<-1

c<- -1

when you type that in your R console, there is no output, which means the variable assignment worked (otherwise you would have an error message).

To see the content of the variable we have 2 ways:

Either we just type the name of the variable:

*a*

*[1] 1*

Or we can use the print function like this:

*Print(a)*

*[1] 1*

The term objects described stuff used to stored in R, it can be simple values called variables, as above, or can be much more complicated with function storing. Every time an object is changed, or created, we are changing what we call the workspace, if you type the function ls() you will see all the variable available in your workspace:

*ls()*

*[1] “a” “b” “c”*

Usually IDE have a tab showing all the variables in the workspace.

Now that those values are variables, we can use them in the quadratic solution to find the solution the equation:

*(-b+sqrt(b^2-4\*a\*c))/2\*a*

*[1]0.618*

*(-b-sqrt(b^2-4\*a\*c))/2\*a*

*[1]-1.618*

* + 1. Functions

Once variables are defined, data analysis can be described as a series of functions applied to the data. We’ve already seen a couple of them: install.packages, library, ls functions and sqrt that we used to solve the quadratic equation. There is many more built in functions and even more coming from the packages. Those functions do not appear in the workspace but are ready to use. Generally to evaluate a function you need the use of parenthesis. If you don’t put the parenthesis, for example on ls function, you will get the code of the function instead of the evaluation. In the case of ls, you just need an opened and closed parenthesis for ls to get evaluated and give you all the variables in the workspace. Most of the function requires at least 1 parameter to work, for example the log function, which is a function that calculate the natural log. If you type:

*Log(8)*

*[1] 2.079442*

8 is the argument of the log function. We can also passes defined variables in the arguments of functions, for example let’s take the variable a=1 defined earlier, we can type:

*log(a)*

*[1] 0*

Furthermore in R, functions can be nested, by that we mean that the result of a function can be given to another function as an argument, if we passes a to the function exp(), that stands for exponential, we’ll get:

*exp(a)*

[1] 2.718282

Exponential and log are inverse functions, so you can take the result of exp(a) to passes it as the argument of log() if you do:

*log(exp(a))*

*[1] 1*

In the case of nested functions, it is important to remember that the functions are evaluated from the inside out. If you have questions on how to use a particular functions and what are its parameters you can use the help, to see the help on a function you do:

*help(“log”)*

in the help file if a value is assigned by default to a parameter, we call that parameter optional. R assumes that you pass the parameter in the order the appear in the help file, for example if you type:

*log(8,2)*

*[1] 3*

R will assume that you want the log of 8 in base 2. If you specify the name of the parameter in your call to the function:

*log(base=2,x=8)*

*[1] 3*

You can put the parameter in whatever order you see fit.

Data sets are included also for users to practice and test functions, you can sell all the available data sets by typing

*data()*

* 1. Data Types

Variables in R can be of different types. We need to be able to distinguish numbers from character strings and tables from simple list of numbers. The function class() helps determine the type of an object. For example:

*a<-2*

*class(a)*

*[1] “numeric”*

So far we’ve seen variable that stores only 1 value, which is not very useful to store data. The most common way to store data in R is with data frames. Data frames are like table row represents observation and columns different variables. They are useful for data sets because we can combine different types of variables in the same data frame. For example

*library(dslabs)*

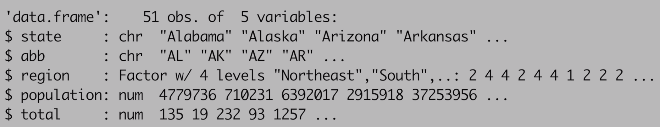
*data(“murders”)*

*class(murders)*

*[1] “data.frame”*

Now we know our data is stored in the object murders. But we need more information about the data itself, to do that we use the function str() that stands for structure. If we type:

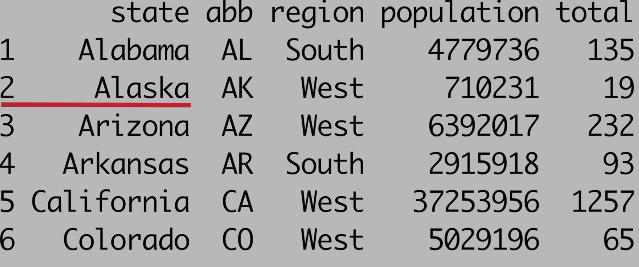
*str(murders)*



We can see that we have 51 observations, 5 variables, with the name of the 5 variables starting the $ in this case state, abb (abbreviation), region, population and total.

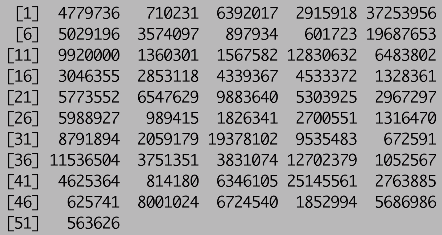
We can also show the first 6 lines of the data frame with the function head():

*head(murders)*



For our analysis we will need to access the different variables represented by columns, we achieve that with the $ it’s called the accessor. Here is an example:

murders$population



It shows us all the information stored in the population column. To know the name of the variable in the data frame we can use the str function as shown above or use the names() functions which will give us the name of all the columns in the data frame.

It is important to note that the orders of the row in murders$population conserved the order of the rows is the original data frame murders. Furthermore, the object murders$population is not one number but 51. We call these type of objects vectors. The function length tell you how many value there is in a vector. For example:

*pop<-murders$population*

*length(pop)*

*[1] 51*

This vector is a **numeric** vector since the population sizes are numbers.

We can store **characters** in R as well, because variables also use character strings, we are using the quotes “” to distinguish between variable names and character strings. For example:

*a<-1*

*a*

*[1] 1*

*“a”*

*[1] “a”*

In our example the state column stored characters so:

*class(murders$state)*

*[1] “character”*

As numeric vectors all value in character vector must be a character. There is another type of vector, not present in this data frame, the **logical** vector which stored TRUE or FALSE value. We can create a logical vector by:

*z<- 3==2*

*z*

*[1] FALSE*

It is FALSE because 3 is not equal to 2. It is because “==” is a relational operator asking if 3 is equal to 2.

The last type of data we need to talk about, it’s the **factor.** In our data set we have a variable called regions; these are the regions of the US. So which state is in which regions? Because there is only 4 regions, we call this variable a categorical variable, with only 4 categories, each state is in one of this 4. The factor type represents this categorical variable and the categories values are called levels. We can see those levels by using the functions level():

*levels(murders$region)*

*[1] “Northeast”,”South”,”North Central”, “West”*

1. Vectors, Sorting
   1. Vectors
      1. Vectors

Vectors are the most basic unit to stored data in R. Complex data sets can usually be broken down into components that are vectors. For example, in a data frame such as the murders data frame each column is a vector.

To create a vector, you have different methods. One way is to use the function “c”, which stands for concatenate. For example:

*codes<-c(380,124,818)*

We can also create characters vectors, for example:

*country<-c(“italy”, “canada”,”egypt”)*

If you don’t use the quotes here, R will look for variables Italy, Canada and Egypt and will return an error because they don’t exist. Sometime it is useful to name the entries of a vector, for example when defining a vector of country codes as we did, we can use the names to connect the two vectors:

*codes<- c(italy=380, canada=124,egypt=818)*

We can also use the names function to assign name to the value of a vector for example:

*codes<-c(380,124,818)*

*country<-c(“italy”, “canada”,”egypt”)*

*names(codes)<-country*

Another useful function to create vectors is the function seq, for sequence, for example:

*seq(1,10)*

*[1] 1 2 3 4 5 6 7 8 9 10*

In this function the first argument is the start and the second is the end, the default is to increment by 1, we can change the increment by having a 3rd like this:

*seq(1,10,2)*

*[1] 1 3 5 7 9*

Now let’s talk about subsetting, subsetting let us access specific parts of a vector, we use [] to access an element of a vector for example:

*codes[2]*

*[1] canada*

*124*

You can get more than one entry by using a multi-entry vector as index. For example we can type codes, and then create the vector c(1,3) and now get the first and 3rd element:

*codes[c(1,3)]*

*italy egypt*

*380 818*

the seq function above is particular useful if you want to access, say the first two elements:

*codes[1:2]*

*[1] italy canada*

*380 124*

*If the entries has names we can access them using those names for example:*

*codes[“canada”]*

*canada*

*124*

* + 1. Vectors coercion

R was developed with an idea of being flexible with data types, when an entry does not match the expected R tried to guess what we meant before throwing an error. That behavior is sometimes useful but might lead to a great deal of confusion for the developer. For example, we said earlier that vectors must be all of the same type so if we type:

*x<-c(1,”canada”,3)*

We expect an error, but we don’t, we don’t even get a warning. If you look at the type of x:

*x*

*[1] “1” “canada” “3”*

R has converted the 1 and 3 to character strings, and the class of x is character, we say that R coerced the data into a character string. It guessed that because of the character string in the middle of the data.

R offer function to force for a specific coercion, for example you can turn number into characters with the as.character() function:

*x<-1:5*

*y<-as.character(x)*

*y*

*[1] “1” “2” “3” “4” “5”*

We can turn them back to numeric with the function as.numeric().

Missing data is very common in practice, in R we have a special value for missing data it’s the NA. We can get NAs from coercion for example:

*x<-c(“1”,”b”, “3”)*

*as.numeric(x)*

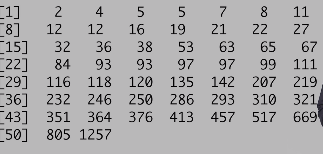
*[1] 1 NA 3*

NA was introduced because R couldn’t convert “b” to a numeric

* 1. Vectors sorting

Now that we saw some context on how variable are handled in R, let’s go back to our initial data sets and try to gain some insights into the safety of different states in the context of gun murders. We want to rank the states from the least to most dangerous. We can use the function sort() to sort a vector in increasing order. For example:

*sort(murders$total)*

*[1]* 

We can see that the smallest number of murders had 2 and the one with the most had 1257, however this does not give us any information about the states only the totals. The order() function is closer to what we want. This function takes a vector a returns the indices that sorts that vector, let’s see an example:

*x<- c(31, 4, 15, 92, 65)*

*x*

*[1] 31 4 15 92 65*

*sort(x)*

[1] *4 15 31 65 92*

*index<-order(x)*

*x[index]*

*[1] 4 15 31 65 92*

If we compare the object x and the index we can see why it works:

*x*

[1] 31 4 15 92 65

order(x)

[1] 2 3 1 5 4

This is the index that put x in order, 2 is the first entry because the second entry of x is 4 which is the smallest number, then the second smaller is 15 at index 3 of x and so on..

How this will help us in order the states by murder? Remember that the entries of vector you accessed with the dollar sign follow the same order as the rows in the table. So, we can order the state names by their total murders by first obtaining the index that orders according to murders total and then indexing the state names or abbreviations using that index:

*index<-order(murders$total)*

*murders$state[index]*

*[1] "Vermont" "North Dakota" "New Hampshire"*

*[4] "Wyoming" "Hawaii" "South Dakota"*

*[7] "Maine" "Idaho" "Montana"*

*[10] "Rhode Island" "Alaska" "Iowa"*

*[13] "Utah" "West Virginia" "Nebraska"*

*[16] "Oregon" "Delaware" "Minnesota"*

*[19] "Kansas" "Colorado" "New Mexico"*

*[22] "Nevada" "Arkansas" "Washington"*

*[25] "Connecticut" "Wisconsin" "District of Columbia"*

*[28] "Oklahoma" "Kentucky" "Massachusetts"*

*[31] "Mississippi" "Alabama" "Indiana"*

*[34] "South Carolina" "Tennessee" "Arizona"*

*[37] "New Jersey" "Virginia" "North Carolina"*

*[40] "Maryland" "Ohio" "Missouri"*

*[43] "Louisiana" "Illinois" "Georgia"*

*[46] "Michigan" "Pennsylvania" "New York"*

*[49] "Florida" "Texas" "California"*

Now if we are only interested in the largest value, we can use the function max():

*max(murders$total)*

*[1] 1257*

Which.max give us the index where this number resides so if we define i\_max as:

*i\_max<-which.max(murders$total)*

*i\_max*

*[1] 5*

It give me the index at which to find the value 1257, and now I can index the state with i\_max to get the name of the state we are looking for:

*murders$state[i\_max]*

*[1] "California"*

We can do the exact same thing with min and which.min to get the minimum. To finish there is one last functions that can help us achieve the sorting it’s the function rank(), for any given list it gives you a vector with the rank of the first entry, second entry,… for example:

*x<-c(31,4,15,92,65)*

*x*

*[1] 31 4 15 92 65*

*rank(x)*

*[1] 3 1 2 5 4*

31 is the 3rd smallest entry in x, 4 is the smallest and so on..

* 1. Vector Arithmetic

We just saw that California had the most murders of any state, but does this mean it is the most dangerous state? What if it just has more people than any other states? Using what we learned so far we can quickly see that indeed California has the largest population:

*murders$state[which.max(murders$population)]*

*[1] "California"*

We can know how much population it has just by typing:

*max(murders$population)*

*[1] 37253956*

With more than 37,000,000 inhabitants it might be unfair to compare California to other the states, what we should do is compute the murders per capita, and this where the arithmetic capabilities of R come in handy.

In R arithmetic operation on vectors occur element wise. For example

*heights<- c(69,62,66,70,70,73,67,73,67,70)*

*heights\*2.54*

*[1] 175.26 157.48 167.64 177.80 177.80*

*[6] 185.42 170.18 185.42 170.18 177.80*

In here all elements of the vector height has been multiplied by 2.54, in a similar way we can subtracts the average value to all the values of the vector for example if the average is 69 then :

*heights-69*

*[1] 0 -7 -3 1 1 4 -2 4 -2 1*

We now have for each entry how much they differ from 69. If we have 2 vectors of the same length these operations are even more powerful, adding 2 vectors of the same length is done by adding they entry by entry, and the same goes for all other operations as division, subtractions and multiplication. That implies that to compute the murder rate per 100000 people we simply do:

*Murder\_rate<-murders$total/murders$population\*1000000*

Once we do this we see that California is no longer at the top, we can use what we learned to order states by murder rate, in this case we ordering them in decreasing order:

murders$state[order(murder\_rate,decreasing = TRUE)]

*[1] "District of Columbia"*

*[2] "Louisiana"*

*[3] "Missouri"*

*[4] "Maryland"*

*[5] "South Carolina"*

*[6] "Delaware"*

*[7] "Michigan"*

We can that California is not even in the top 10 (14th place), the highest rate of murder is in the district of Columbia.

1. Indexing, Data Wrangling, Plots
   1. Indexing
      1. Indexing

R provide a powerful way of indexing vectors. We can, for example, subset a vector based on properties of another vector. Let’s continue our example or murders in the US, using the murders\_rate we calculated before. Imagine you coming from Italy, where according to ABC news the murder rate is only at 0.71 per 100000 people, to the US and you want to find the state with a murder rate similar or lower than 0.71. We are going to define index as the murder rate smaller than 0.71:

*index<-murder\_rate<=0.71*

*index*

*[1] FALSE FALSE FALSE FALSE FALSE*

*[6] FALSE FALSE FALSE FALSE FALSE*

*[11] FALSE TRUE FALSE FALSE FALSE*

*[16] TRUE FALSE FALSE FALSE FALSE*

*[21] FALSE FALSE FALSE FALSE FALSE*

*[26] FALSE FALSE FALSE FALSE TRUE*

*[31] FALSE FALSE FALSE FALSE TRUE*

*[36] FALSE FALSE FALSE FALSE FALSE*

*[41] FALSE FALSE FALSE FALSE FALSE*

*[46] TRUE FALSE FALSE FALSE FALSE*

*[51] FALSE*

The result is a vector with 51 entries either TRUE or FALSE, depending if the murder rate is lower than 0.71. To see which states it correspond, we can leverage the fact that vectors can be indexed with logicals:

*murders$state[index]*

*[1] "Hawaii" "Iowa"*

*[3] "New Hampshire" "North Dakota"*

*[5] "Vermont"*

Another nice feature is that to count how many entries are true, the function sum returns the sum of these entries, because R coerced the logical vectors to numeric by turning TRUE to 1 and FALSE to 0.

Now imagine we like the mountains and we want to move to a safe state in the west region of the country( where mountains are). We want the murder rate to be at most 1. Here is how we do it:

*west<-murders$region=="West"*

*safe<-murder\_rate<=1*

*index<- safe & west*

*murders$state[index]*

*[1] "Hawaii" "Idaho" "Oregon"*

*[4] "Utah" "Wyoming"*

* + 1. Indexing function

We going to talk about three very useful functions related to indexing that use logical operators:

* + - * which
      * match
      * %in%

**which()** gives us the entries of a logical vector that are true, for example suppose we want to look up Massachusetts’ murder rate:

*index<-which(murders$state=="Massachusetts")*

*> index*

*[1] 22*

*murder\_rate[index]*

*[1] 1.802179*

**match()** looks for entry in a vector and returns the index needed to access them, suppose instead of wanting the murder rate of just one state we want several like New York, Florida and Texas:

*index<-match(c("New York","Florida","Texas"),murders$state)*

*murder\_rate[index]*

*[1] 2.667960 3.398069 3.201360*

The **%in%** operator, as implied this operator returns a logical vector telling us if entry of one vector are in another one for example, let’s say you are not sure that Boston, Dakota and Washington are state’s and you want to find out here is how you do it:

*c("Boston","Dakota","Washington") %in% murders$state*

*[1] FALSE FALSE TRUE*

* 1. Basic Data Wrangling
     1. Basic Data Wrangling

Up to now we’ve been manipulating vectors, but once we start more advanced analyses we want to manipulate data tables. For this purpose we will talk about the dplyr package, it provides intuitive functionality for working with tables. To use it you need to install and load the package like this:

*install.packages("dplyr")*

*library(dplyr)*

This package introduces functions that perform the most common operations in data manipulations and uses names for these functions that are relatively easy to remember. For example, to change the data table by adding a new column or changing an existing one, we use **mutate(),** to filter the data by subsetting rows we use the function **filter()** and to subset the data by selecting specific columns, we use **select().** We can also perform series of operation, by sending the result of one functions to another, by using the pipe operator **%>%.** For example:

*murders<-mutate(murders,rate=total/population\*100000)*

*#looking at the new data*

*head(murders)*

*state abb region population*

*1 Alabama AL South 4779736*

*2 Alaska AK West 710231*

*3 Arizona AZ West 6392017*

*4 Arkansas AR South 2915918*

*5 California CA West 37253956*

*6 Colorado CO West 5029196*

*total rate*

*1 135 2.824424*

*2 19 2.675186*

*3 232 3.629527*

*4 93 3.189390*

*5 1257 3.374138*

*6 65 1.292453*

Note that we use total and population, these are not defined in the workspace, mutate knows to look for those value in the data frame instead of the workspace.

*#Entries for which the murder rate is lower that 0.71*

*filter(murders,rate<=0.71)*

*state abb region*

*1 Hawaii HI West*

*2 Iowa IA North Central*

*3 New Hampshire NH Northeast*

*4 North Dakota ND North Central*

*5 Vermont VT Northeast*

*population total rate*

*1 1360301 7 0.5145920*

*2 3046355 21 0.6893484*

*3 1316470 5 0.3798036*

*4 672591 4 0.5947151*

*5 625741 2 0.3196211*

*#Definition of new\_table as a selection of just 3 columns of murders*

*new\_table<-select(murders,state,region,rate)*

*#Filtering of this new\_table for states that have a murder rate less than 0.71*

*filter(new\_table,rate<=0.71)*

*state region*

*1 Hawaii West*

*2 Iowa North Central*

*3 New Hampshire Northeast*

*4 North Dakota North Central*

*5 Vermont Northeast*

*rate*

*1 0.5145920*

*2 0.6893484*

*3 0.3798036*

*4 0.5947151*

*5 0.3196211*

We’ve just seen how **select(), mutate()**  and **filter()** works. Now let’s see how to put them together using **%>%** operator:

*#Definition of a new table with 3 variables for states that have murder rates below 0.71*

*murders %>% select(state,region,rate) %>% filter(rate<=0.71)*

*state region*

*1 Hawaii West*

*2 Iowa North Central*

*3 New Hampshire Northeast*

*4 North Dakota North Central*

*5 Vermont Northeast*

*rate*

*1 0.5145920*

*2 0.6893484*

*3 0.3798036*

*4 0.5947151*

*5 0.3196211*

Note that using the pipe operator you don’t need to provide the argument for the data because dplyr assumes that whatever is being piped is what should be worked on.

* + 1. Creating Data Frames

A lot of the analyses done in dplyr find necessary to create data frames in R. We do this with the function **data.frame()**, here is an example:

*murders %>% select(state,region,rate) %>% filter(rate<=0.71)*

*#Creating an object grades containing name, grade at exam1, grade at exam2*

*grades<-data.frame(names=c("John","Juan","Jean","Yaa"),exam\_1=c(95, 80, 90, 85),exam\_2=c(90, 85, 85, 90))*

*#Seeing the result*

*grades*

*names exam\_1 exam\_2*

*1 John 95 90*

*2 Juan 80 85*

*3 Jean 90 85*

*4 Yaa 85 90*

Be careful by default data.frame() turns characters into factors, For example :

*#class of grades$names*

*class(grades$names)*

*[1] "factor"*

To avoid this we use the argument **StringAsFactor=True** in the creation of the data frame:

*#Creating an object grades containing name, grade at exam1, grade at exam2 with names as character instead of factor*

*grades<-data.frame(names=c("John","Juan","Jean","Yaa"),exam\_1=c(95, 80, 90, 85),exam\_2=c(90, 85, 85, 90),stringsAsFactors = FALSE)*

*#class of grades$names with the change*

*class(grades$names)*

*[1] "character"*

* 1. Basic Plots

One of the main strength of R might be the Exploratory data visualization, someone can quickly go from idea to data to plot with flexibility and ease. Excel may be easier than R, but it is nowhere as near as flexible. D3, an interactive data visualization programming language, may be more flexible and powerful than R, but it takes much longer to generate a plot let’s use an example:

*#We define 2 objects population\_in\_millions and total\_gun\_murders and plot them against each other*

*population\_in\_millions<-murders$population/10^6*

*total\_gun\_murders<-murders$total*

*plot(population\_in\_millions,total\_gun\_murders)*

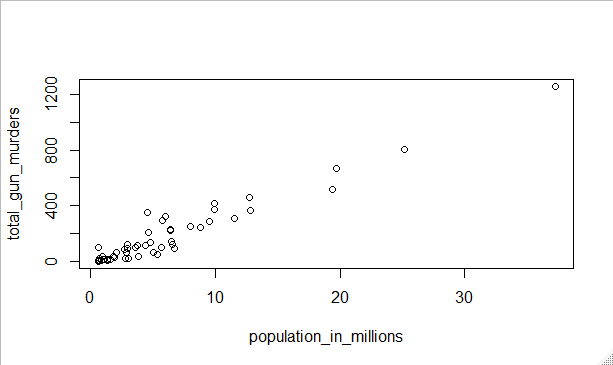


Figure 1 Population versus gun murders

We can see that there is a relationship between population and total gun murders.

Another example of quickly done plot are histograms. They are powerful graphical summaries of a list of numbers that gives you a general overview of the types of values you have. Here is how you make one:

*#Creation of an histogram of murder rate*

*hist(murders$rate)*

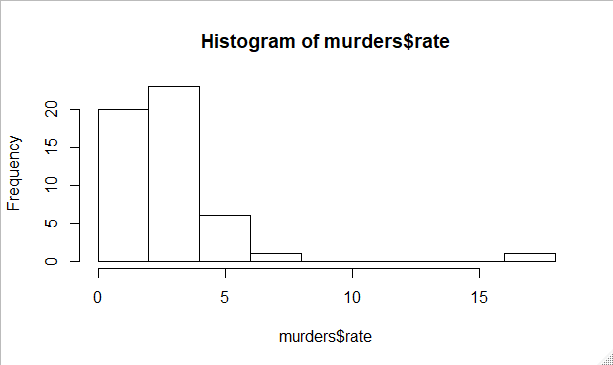


Figure 2 Histogram of murder rate

We can see that there’s a wide range of values of murder rate, most of them between 2 and 3 with one extreme value at over 15.

Finally there is boxplot, boxplot are more terse summary than the histogram, but they’re easier to stack against each other. So we can see many distributions in one plot, we going to use them to compare murder rates for different regions:

*#Comparing murder rate in different region*

*boxplot(rate ~ region,data=murders)*

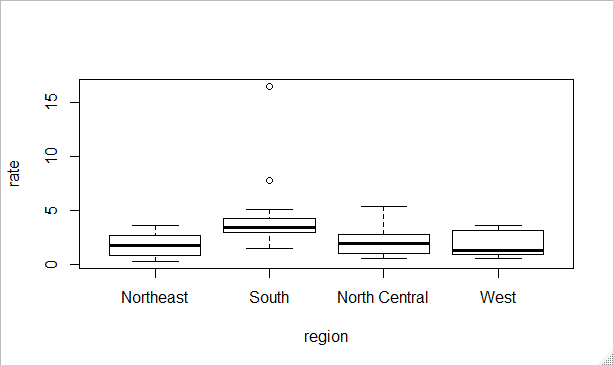


Figure 3 boxplot of murder rate by region

We can see that the South has the highest murder rate of all regions.

1. Programming Basics
   1. Introduction to Programming in R

R is not just a data analysis environment but a programming language as well. Advance R programmers can develop complex packages, and even suggest ways to improve R itself. In here we are going to cover 3 key concepts of programming: **conditional expressions, for loops and building functions.**

* 1. Conditionals

Conditions are the bases of programming, on of the most known conditional expression in the programming world is the **if..else** statement. Here is general example of how it works in R:

*a<-2*

*if(a!=0){*

*print(1/a)*

*}else{*

*print("No reciprocal of 0.")*

*}*

You will get 0.5 with a=2 and a different one (no reciprocal of 0) when a=0. Let’s look at the some example with the US murders:

*#Load data frame and define rate variable*

*data("murders")*

*murder\_rate<-murders$total/murders$population\*100000*

*#We ask the question which states, if any, have murder rates lower than 0.5*

*ind<-which.min(murder\_rate)*

*if(murder\_rate[ind]<0.5){*

*print(murders$state[ind])*

*}else{*

*print("No state has murder rate that low")*

*}*

*[1] "Vermont"*

Another function that is very useful is the ifelse function , it takes three arguments, a logical and two possible answers. If the logical is true the first is returned, otherwise the 2nd is returned. Here’s an example:

*a<-0*

*ifelse(a>0,1/a,NA)*

*[1] NA*

This function is practically useful because it works on vector, it examines each element of the logical vector and returns a corresponding answer accordingly. Here is an example:

*a<-c(0,1,2)*

*> result<-ifelse(a>0,1/a,NA)*

*> result*

*[1] NA 1.0 0.5*

We can use this function to replace missing values by 0 in a data set. For example:

*data("na\_example")*

*sum(is.na(na\_example))*

*[1] 145*

*no\_nas<-ifelse(is.na(na\_example),0,na\_example)*

*sum(is.na(no\_nas))*

*[1] 0*

Two more functions that are useful are **any()**  and **all()**, any takes a vector of logicals and it returns true if any of the entries is true here’s an example:

*z<-c(TRUE,TRUE,FALSE)*

*any(z)*

*[1] TRUE*

*z<-c(FALSE,FALSE,FALSE)*

*any(z)*

*[1] FALSE*

The all function takes a vector of logicals and returns TRUE if all the entries are true here’s an example:

*> z<-c(TRUE,TRUE,FALSE)*

*> all(z)*

*[1] FALSE*

*> z<-c(TRUE,TRUE,TRUE)*

*> all(z)*

*[1] TRUE*

* 1. Functions

As you go through the analysis, you will note that you’ll repeat the same operation over and over, for example the computation of the average. You compute the average of a vector by:

We can define function that does this automatically. Because we do that computation often it’s more efficient to write a function that performs that operation, which has been done, this is the role of the mean function in R.

However you will encounter situation in which the function that you need does not already exist, so you have to write your own.

You can define a simple version of a function that computes average by:

*avg<-function(x){*

*s<-sum(x)*

*n<-length(x)*

*s/n*

*}*

Note that variable created inside a function are not saved in the workspace, so while we use s and n in the function they are created and changed only during the call to the function.

You can have multiple arguments for a function, for example:

*avg<-function(x,arithmetic=TRUE){*

*n<-length(x)*

*ifelse(arithmetic,sum(x)/n,prod(x)^(1/n))*

*}*

Note that the arithmetic parameter has a default value of TRUE.

* 1. For Loops
     1. For Loops

We are going to talk about loops, to explain it we are going to take an example from math. There is a formula that tell you what the sum of 1 + 2 + 3…+ n is. The formula is:

But what if we were not sure that was the right formula? We can easily check with R. Using what we know we can easily write a function that compute the sum:

*compute\_s\_n<-function(n){*

*x<-1:n*

*sum(x)*

*}*

Now, let’s say we want to compute this sum for various values of n. We are computing 25 sums, so we can’t use 25 lines of codes, that’s what loops are for. For loops let us define the range that our variable takes, In our example, it would go from 1 to 25, then change the value as you loop and evaluate the expression every time inside your loop. For example:

*m<-25*

*#Create empty vector*

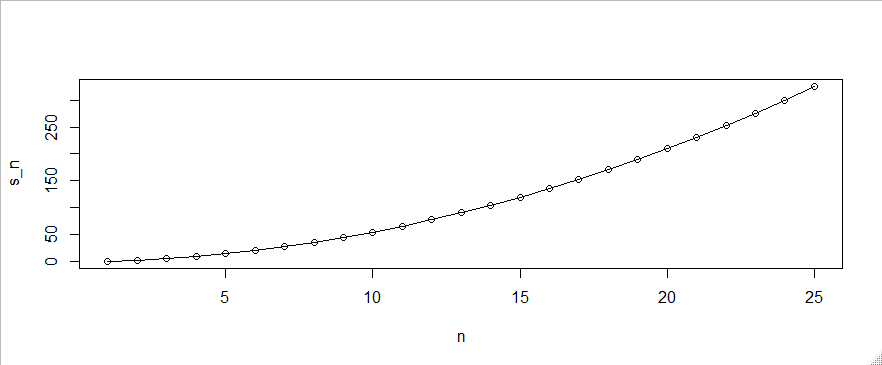
*s\_n<-vector(length = m)*

*for(n in 1:m){*

*s\_n[n]<compute\_s\_n(n)*

*}*

We can compare our function with the formula by plotting the two on the same plot with:



We can see that the form is quadratic that suggest that they are the same.

* + 1. Other Functions

You should check other functions that are widely used such as :

* apply
* sapply
* mapply
* mapply

They are part of the apply family and there is other too like:

* split
* cut
* quantile
* reduce
* identical
* unique
* etc…