Statistic Quality & Engineering
Universitas Indonesia

Module 1 :

R Basics

A brief introduction

SQE - 2020

Content of this module :

1. R Basics, Function, and Data Types
2. Vectors
3. Indexing, Data Wrangling, and Basic Plots
4. Programming Basics
5. R Markdown

This is what you will learn in this section...

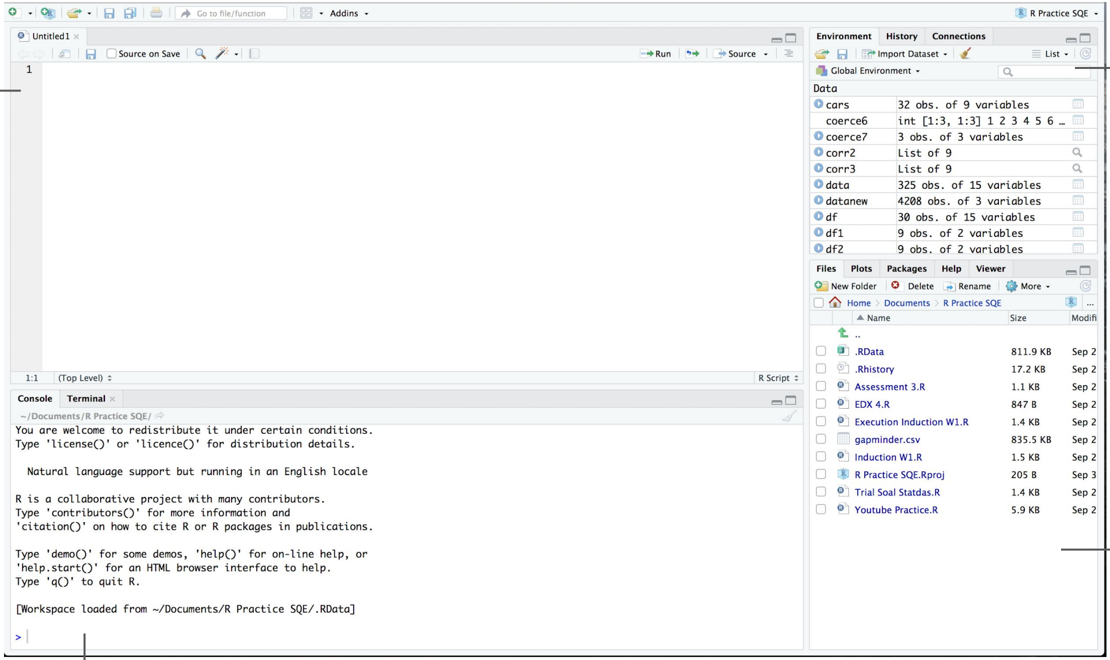
Section 1 - overview

Section	What you will learn...
1.1	• R Studio Interface • Installing packages • Load datasets • Running commands while editing scripts
1.2	• Assigning values to an object • Solving equations with functions in R
1.3	• Identifying data types • Learn the structure of a data frame and accessing columns • Learn about the differences of each data type

R Studio

The interface of R Studio

R Script: a field to write commands and can be saved later.



Environment: informs about datasets and objects loaded in your workspace

Files: The directory, used to access files/folders

R Console: a field to write commands and display results.

R Studio

The interface of R Studio

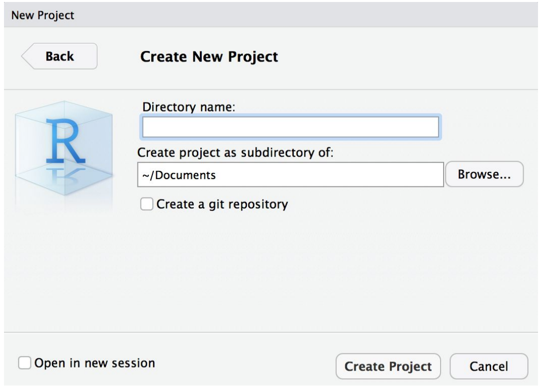
First step in R, create a Working Directory

- **Working Directory** is a folder where R reads and save files. Why is it important that we create a working directory?
    - All of the scripts will be saved in a single working directory
    - R will directly be able to read data and/or files that exist in the working directory
    - All data and variables saved in the workspace would always be accessible
    - If we were to move the folder to another location, we do not need to redefine the new path of a data/file

To create a new working directory, we simply create a new project.

File > New Project... > New Directory

A folder will be created and there would be a file with the same name in an .RProj extension.



R Studio

The interface of R Studio

There are two options to run commands in R Studio,

Using R Script

- Commands in R Script can be edited
- To run a single line of command, use *Ctrl + Enter* (Windows) or *Cmd + Enter* (Mac)
- To run the entire script, use *shift* key: *Shift + Ctrl + Enter* (Windows) or *Shift + Cmd + Enter* (Mac)

```
library(dslabs)
library(dplyr)

```

- The commands would be returned in the console
- R Script can be saved as a .R file
- To access: *File > New Script*
  Or keyboard shortcut to a new R Script is:
  > Shift + Ctrl + N (Windows)
  > Shift + Cmd + N (Mac)

Using R Console

- Commands in R Console cannot be edited, so a typo means start over (*watch out!*)
- To run commands in R Console, simply use the *enter* key, the result or value would be returned immediately
- Codes written in the console cannot be saved

```
> installed.packages()
           Package       LibPath
abind      "abind"       "/Library/Frameworks/R.framework/Versions/3.5/Resc
ape        "ape"         "/Library/Frameworks/R.framework/Versions/3.5/Resc
askpass    "askpass"     "/Library/Frameworks/R.framework/Versions/3.5/Resc
assertthat "assertthat"  "/Library/Frameworks/R.framework/Versions/3.5/Resc
backports  "backports"   "/Library/Frameworks/R.framework/Versions/3.5/Resc
base       "base"        "/Library/Frameworks/R.framework/Versions/3.5/Resc
base64enc  "base64enc"   "/Library/Frameworks/R.framework/Versions/3.5/Resc
BH         "BH"          "/Library/Frameworks/R.framework/Versions/3.5/Resc
bitops     "bitops"      "/Library/Frameworks/R.framework/Versions/3.5/Resc
blob       "blob"        "/Library/Frameworks/R.framework/Versions/3.5/Resc
boot       "boot"        "/Library/Frameworks/R.framework/Versions/3.5/Resc
```

Packages

Installing and loading packages

Packages are collections of functions and datasets which add extra-functionality to our base R.

How to install packages

```
#For example we would like to install a package named ‘dslabs’
install.packages(“dslabs”)
```

How to load packages

```
#Load dslabs into the R session
library(dslabs)
```

Check list of packages installed

```
installed.packages() #To see list of installed packages
```

Datasets

Load datasets into workspace

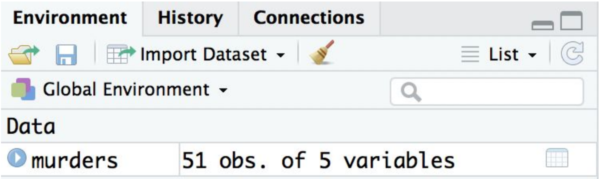
Datasets are what we will work on mainly when using R, there are several methods to access datasets

1. Using datasets installed in R

Datasets are included in packages, or available by default from base R.

```
#For example we would like to load a data
set named 'murders'
data(murders)
```

Our environment should be updated and display 'murders' as one of the data in the workspace



If we click 'murders' on the environment, a table of the data set would be displayed.

Untitled1* murders

Filter

	state	abb	region	population	total
1	Alabama	AL	South	4779736	135
2	Alaska	AK	West	710231	19
3	Arizona	AZ	West	6392017	232
4	Arkansas	AR	South	2915918	93
5	California	CA	West	37253956	1257
6	Colorado	CO	West	5029196	65
7	Connecticut	CT	Northeast	3574097	97
8	Delaware	DE	South	897934	38
9	District of Columbia	DC	South	601723	99
10	Florida	FL	South	19687653	669
11	Georgia	GA	South	9920000	376
12	Hawaii	HI	West	1360301	7
13	Idaho	ID	West	1567582	12
14	Illinois	IL	North Central	12830632	364
15	Indiana	IN	North Central	6483802	142
16	Iowa	IA	North Central	3046355	21

Datasets

Load datasets into workspace

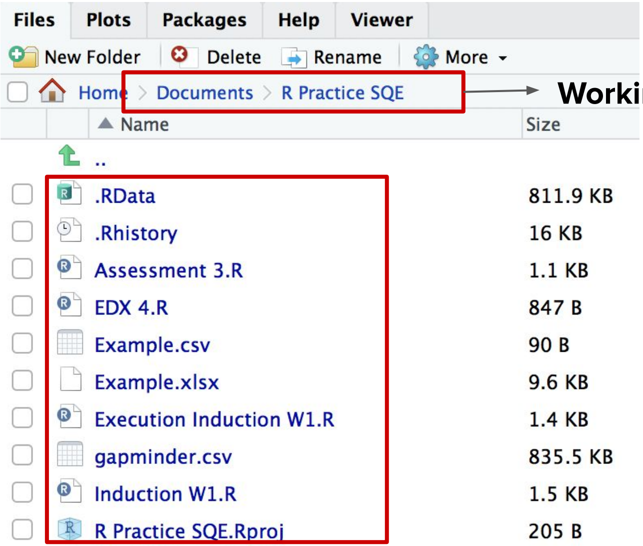
Datasets are what we will work on mainly when using R, there are several methods to access datasets

2. Using our own data by importing datasets

Make sure your file is in your Working Directory
First, we need to make sure that the file or dataset that we'd like to import is in our current working directory.

You can check this by seeing the 'Files' corner. Do you see your dataset on the list?

For example, we would like to access the dataset named Example.csv. If it isn't on the list, what you should do first is move the data to the folder of our working directory. In this case, our working directory is named 'R Practice SQE'



Working Directory

List of files inside our working directory

Datasets

Load datasets into workspace

2a. Importing our data in .xlx/.xlxs format

We can do this manually.
For example we would like to import an .xls/.xlsx file.

Files > Import Datasets > From Excel > Browse file

Or, we could use a function.
To import a dataset in an excel format, we must install a package called "readxl" and load the package.

```
install.packages("readxl")
library(readxl)
```

Let R auto-detect excel format

If we would like R to auto-detect the format of our excel file (either .xls or .xlsx), we use read_excel("x")

```
saved_name <- read_excel("filename.xls")
```

R to read original format of excel (.xls)

If we would like to import an old format, we use read_xls("x")

```
saved_name <- read_xls("filename.xls")
```

R to read new format of excel (.xlsx)

If we would like to import a new format, we use read_xlsx("x")

```
saved_name <- read_xlsx("filename.xls")
```

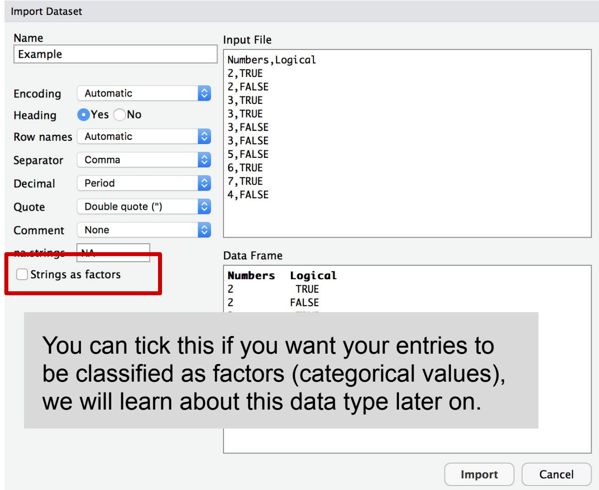
Datasets

Load datasets into workspace

2b. Importing our data in .csv format

We can load them manually.
For example we would like to import a .csv file.

Files > Import Datasets > Text(base) > Select file



You can tick this if you want your entries to be classified as factors (categorical values), we will learn about this data type later on.

Or, we could use a function.
Remember that the file must exist in the working directory. The basic function to import a .csv (for comma separated values) dataset is:

```
saved_name <- read.csv(“dataname.csv”)
```

If our .csv file is separated by semicolon, then we use:

```
saved_name <- read.csv2(“dataname.csv”)
```

Suppose we would like to import a file named *Example.csv* separated by comma, and save it as *df*

```
df <- read.csv(“Example.csv”)
```

Datasets

Load datasets into workspace

2c. Importing our data in .txt format

We can load them manually.
For example we would like to import a .txt file.

Files > Import Datasets > Text(base) > Select file

For .txt data, the values are separated by white space and if we select the file, R would automatically identify white space as separator.

Or, we could use a function.
To import a file in a .txt format, we use the function read.table(“x”)

```
Saved_name <- read.table(“file name.txt”)
```

By default, read.table would claim that our **.txt data does not have a header.** If we do have one, we should add another argument for the function as:

```
Saved_name <- read.table(“file name.txt”,
     header = TRUE)
```

This argument applies to the importing .csv data as well,, however, the default of other functions is *header = TRUE.* So, we need to add an argument if our .csv file does not have a header by adding: , header = FALSE.

To play it safe, you could always add the argument of header = TRUE or header = FALSE depending on your data for the functions: read.csv, read.csv2 or read.table

Datasets

Load datasets into workspace

To sum it up if we would like to import our own data

Function	Format	Typical Suffix
read.csv	Comma separated values	.csv
read.csv2	Semicolon separated values	.csv
read_excel	Auto-detect excel format	.xls or .xlsx
read_xls	Original excel format	.xls
read_xlsx	New excel format	.xlsx
read.table	White space separated values	.txt

Objects

Assigning values

Objects are things that are stored in named containers in R. They can be variables, functions, etc.

How to assign value to an object

```
#We would like to assign the value 1 as ‘a’, 2 as ‘b’, 3 as ‘c’
a <- 1
b <- 2
c <- 3
```

How to know the names of objects saved in your workspace

```
#Show the names of objects saved
ls()
```

```
> ls()
[1] "a" "b" "c"
```

Objects

Assigning values

See the value stored in an object

```
#Either type the object name or use the function print(variable_name)
print(a)
a
```

```
> print(a)
[1] 1
> a
[1] 1
```

Functions

Getting to know functions

Function is a set of statements organized together to perform a specific task.

Before, we already encountered several functions, i.e. install.packages(x), print(x), etc.
What we need to evaluate a function is parentheses ( ), and most functions require arguments inside the parenthesis.

Some functions are installed by default in R, though we can explore more functions by installing packages.
For example, mathematical functions such as *log* and *mean* are useable by default.

```
> log(8,2)
[1] 3
> mean(c(2,4,5,5))
[1] 4
```

This evaluates log 8 with 2 as the base

This evaluates the average of 2, 4, 5, and 5.

How do we know the required arguments for a specific function?
R provides more than enough information for one to explore. To know more about a specific function, we could use *help(function_name)* or *?function_name*

```
#For example we want to know about the function log( )
help(log)        #or
?log
```

Data Types

Knowing and identifying data types and structures in R

There is a wide variety of data types in R

Each data type has some specific functions or actions that could only be performed on them. The basic data types are:

1. **Character**
   Character strings, ex: “src”, “a”, “statdas”
2. **Numeric**
   Real or decimal, ex: 2, 1.15
3. **Integer**
   Integers indicated with L, ex: 1L, 45L
4. **Logical**
   True or False, ex: TRUE, FALSE, T, F
5. **Complex**
   Complex numbers (i.e. imaginary), ex: 1+4i

We can identify them by utilizing the function *class(x)*

```
#Identify data type of an object
class(a)
```

```
> class(a)
[1] "numeric"
```

```
#Identify data type of ‘murders’
class(murders)
```

```
> class(murders)
[1] "data.frame"
```

Note that the function returns ‘data.frame’. It is not a basic data type, but a **data structure.**
In general, data structures are used to handle multiple values.

Data Types

Knowing and identifying data types and structures in R

R has various data structures…

1. **Vector**
   Collection of elements composed of a specific data type.
2. **List**
   A container in which the composition is not restricted to a single data type.
3. **Matrix**
   A vector with dimensions (rows and columns)
4. **Data frames**
   Consists of several list/vector in which every elements have the same length.
5. **Factors**
   A list of data with 'levels' or in other words it is categorical.

We need to get to know the data before processing them

str(x) : to know more about the structure of data
names(x) : to know the data header names
head(x) : display first six lines of the dataset

```
#Suppose we use the 'murders' dataset as an example
str(murders)
names(murders)
head(murders)
```

```
> str(murders)
'data.frame':	51 obs. of  5 variables:
 $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
 $ abb       : chr  "AL" "AK" "AZ" "AR" ...
 $ region    : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
 $ population: num  4779736 710231 6392017 2915918 37253956 ...
 $ total     : num  135 19 232 93 1257 ...
> names(murders)
[1] "state"      "abb"        "region"     "population" "total"
> head(murders)
       state abb region population total
1    Alabama  AL  South    4779736   135
2     Alaska  AK   West     710231    19
3    Arizona  AZ   West    6392017   232
4   Arkansas  AR  South    2915918    93
5 California  CA   West   37253956  1257
6   Colorado  CO   West    5029196    65
```

Data Types

Accessing columns in data frames

Collecting information from a data frame

In order to understand more, sometimes we need to assess only a single column of a data frame. For that, we use the **$** symbol called an *accessor*.

```
#Try to access the population column of 'murders'
dataset
murders$population
#save the column as a separate object named 'pop'
pop <- murders$population
```

If we call 'pop', only the values of the population column would be returned

```
> pop
 [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097   897934   60172
3 19687653  9920000  1360301
[13]  1567582 12830632  6483802  3046355  2853118  4339367  4533372  1328361  577355
2  6547629  9883640  5303925
[25]  2967297  5988927   989415  1826341  2700551  1316470  8791894  2059179 1937810
2  9535483   672591 11536504
[37]  3751351  3831074 12702379  1052567  4625364   814180  6346105 25145561  276388
5   625741  8001024  6724540
[49]  1852994  5686986   563626
```

1 **Knowing number of entries**

To know how many entries are there in a vector, we use length(x)

```
length(pop)
```

```
> length(pop)
[1] 51
```

If we use this function on a data frame, it would return the number of variables (columns) instead.

2 **Obtaining levels of a factor data type**

If we check the class of murders$region, it would return 'factor', thus it contains levels.

```
levels(murders$region)
```

```
> levels(murders$region)
[1] "Northeast"     "South"         "North Central" "West"
```

Content of this module :

1. R Basics, Function, and Data Types
2. **Vectors**
3. Indexing, Data Wrangling, and Basic Plots
4. Programming Basics
5. R Markdown

This is what you will learn in this section...

Section 2 - overview

Section	What you will learn...
2.1	• Create numeric and character vectors. • Name the columns of a vector. • Generate numeric sequences. • Access specific elements or parts of a vector. • Coerce data into different data types as needed.
2.2	• Find the maximum and minimum elements, as well as their indices, in a vector.
2.3	• Perform arithmetic between a vector and a single number. • Perform arithmetic between two vectors of the same length.

Vectors

Creating vectors in R

Vectors is the most basic unit in R to store data.

How to create vectors in R

```
x <- c(1,3,5)          #The function c(), which stands for concatenate
y <- c(“Blue”, “Red”, “Yellow”)
```

How to generate sequence

```
seq(1,5)           #to write numbers from 1 to 5 with increment of 1
[1] 1 2 3 4 5

seq (1,5,2)        #the third argument is how much to jump by
[1] 1 3 5
```

Vectors

Name a column of a vector

Suppose we have 2 vectors *measurement* and *label*. We want to assign the vector label as the name of column

```
measurement <- c(160,55,42)
label <- c("Height", "Weight", "Shoes")

names (measurement) <- label
measurement
```

```
> measurement
Height Weight  Shoes
   160     55     42
```

Vectors

Access specific elements or parts of a vector

Subsetting lets us access specific parts of a vector by using **square brackets**.

```
measurement[2]        #only access second element of data


#using multi entry vector as an index

measurement [c(1,3)]  #access first and third data
measurement [1:2)]    #get the first two element of vector


#using names for the entries

measurement["Height"]
```

Vectors

Vector Coercion

In general, coercion is an attempt by R to be flexible with data types. When an entry does not match the expected, R tries to guess what we meant before throwing in an error.

```
x <- c(1, “Apple”, 3)
```

If we run this into R, we don’t get an error. Why?

```
> x <- c(1, "Apple", 3)
> x
[1] "1"     "Apple" "3"
```

Because R coerced the data into a character string. It means that R has converted the number 1 and 3 into string, even though 1 and 3 were originally numbers.

```
> class(x)
[1] "character"
```

Vectors

Vector Coercion

We can turn numbers into character and otherwise.

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

y <- as.character(x)      #turn a vector to a string vector
[1] "1" "2" "3" "4" "5"

z <- as.numeric(y)        #turn a vector to a numeric vector
[1] 1 2 3 4 5
```

Missing Data

We can get NAs from coercion. For example, when R fails to coerce something, it tries to coerce but it can't, we will get NA. Here's an example:

```
x <- c(1, "Apple" , 3)
y <- as.numeric(x)        #turn a vector to a numeric vector
[1] 1 NA  3
```

Sorting

Return minimum and maximum data

The function **max()** returns the largest value, while **which.max()** returns the index of the largest value. The functions **min()** and **which.min()** work similarly for minimum values.

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

max(x)
[1] 5                # 5 is maximum value in vector x

min (x)
[1] 1                # 1 is minimum value in vector x

which.max(x)
[1] 3                # the third data is maximum value in vector x

which.min(x)
[1] 4                # the fourth data is minimum value in vector x
```

Vector Arithmetic

Rescaling a vector

In R, arithmetic operations on vectors occur element-wise.
For example: suppose we have height in inches (`height_inc`) and want to convert to centimeters.

```
height_inc <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)

height_cm <- height_inc * 2.54     #we multiplied each element by 2.54
```

```
> height_cm
 [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18 177.80
```

Content of this module :

1. R Basics, Function, and Data Types
2. Vectors
3. **Indexing, Data Wrangling, and Basic Plots**
4. Programming Basics
5. R Markdown

This is what you will learn in this section...

Section 3 - overview

Section	What you will learn...
3.1	• Indexing • Logical Operators
3.2	• Basic Data Wrangling • Creating Data Frames
3.3	• Basic Plots • Histogram • Box Plot

Indexing

Section 3.1 : Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector

Use a dataset provided by R named "murders", then load the data:

```
library(dslabs)
data("murders")
```

We can calculate murder rate, then saved it to variable called *murder_rate*:

```
murder_rate <- murders$total/murders$population*100000
```

Indexing

Section 3.1 : Indexing

Another feature of R is that we can use logical to index vectors. If we compare a vector to a single number, it actually performs the test for each entry

```
ind <- murder_rate < 0.71
```

```
> ind
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[49] FALSE FALSE FALSE
```

Note that we get TRUE for each entry smaller than 0.71

To see which state these are:

```
> murders$state[ind]
[1] "Hawaii"        "Iowa"          "New Hampshire" "North Dakota"
[5] "Vermont"
```

If we are only interested in counting the numbers:

```
> sum(ind)
[1] 5
```

Indexing

Section 3.1 : Logical Operators

R has many logical operators which can be used in various situations, some of them are:

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x | y	x OR y
x & y	x AND y

For example, we can form two logical:

```
west <- murders$region == “West”
safe <- murder_rate < 1
ind <- safe & west
```

```
> murders$state[ind]
[1] "Hawaii"   "Idaho"    "Oregon"
  "Utah"     "Wyoming"
```

We use “&” to get a vector that satisfy both conditions then we use “[ ]” to see which state these are

Indexing

Section 3.1 : Logical Operators

Basically, there are three functions that we can use in logical operators

1 which

which tells us which entries of a logical vectors are TRUE

```
ind <-
+ which(murders$state ==
+        “California”)

ind
murder_rate[ind]
```

```
> ind
[1] 5
> murder_rate[ind]
[1] 3.374138
```

2 match

match tells us which indexes of second vectors match each entries of a first vector

```
ind <-
+ match(c(“New York”,
+         “Texas”),
+       murders$state)

ind
murder_rate[ind]
```

```
> ind
[1] 33 44
> murder_rate[ind]
[1] 2.66796 3.20136
```

3 %in%

which tells us which entries of a logical vectors are TRUE

```
c(“Boston”, “Dakota”,
+ “Washington”) %in%
+ murder$state
```

```
> c("Boston", "Dakota",
+    "Washington")%in%
+    murders$state
[1] FALSE FALSE  TRUE
```

Data Wrangling

Section 3.2 : Basic Data Wrangling

Once we start more advanced analyses, we will want to manipulate data tables

For that purpose, we should use “dplyr” package

```
install.package(“dplyr”)
library(dplyr)
```

We continue working with the dataset

```
data(“murders”)
murder_rate <- murders$total/murders$population*100000
```

Data Wrangling

Section 3.2 : Basic Data Wrangling

Basically, we will learn about four function if we discuss about wrangling the data

1 **mutate**

"mutate" is used to change the data table by adding new column or changing an existing one

Suppose we want to add the murder rate to the data frame:

```
murders_new <- mutate(murders, rate = murder_rate)
head(murders_new)
```

```
> head(murders_new)
       state abb region population total     rate
1    Alabama  AL  South    4779736   135 2.824424
2     Alaska  AK   West     710231    19 2.675186
3    Arizona  AZ   West    6392017   232 3.629527
4   Arkansas  AR  South    2915918    93 3.189390
5 California  CA   West   37253956  1257 3.374138
6   Colorado  CO   West    5029196    65 1.292453
```

Note that there is a new column called **"rate"** which contains the value from **murder_rate** or **murders$total/murders$population*100000**

Data Wrangling

Section 3.2 : Basic Data Wrangling

Basically, we will learn about four function if we discuss about wrangling the data

2 **filter**

“filter” is used to filter the data by subsetting rows

Suppose we want to filter the data table to show only the entries which murder rate is lower than 0.71:

```
murders_new <- filter(murders_new, rate < 0.71)
murders_new
```

```
> murders_new
          state abb        region population total      rate
1        Hawaii  HI          West    1360301     7 0.5145920
2          Iowa  IA North Central    3046355    21 0.6893484
3 New Hampshire  NH     Northeast    1316470     5 0.3798036
4  North Dakota  ND North Central     672591     4 0.5947151
5       Vermont  VT     Northeast     625741     2 0.3196211
```

Note that variable **“murders_new”** only contain rows which **rate** are lower than 0.71

Data Wrangling

Section 3.2 : Basic Data Wrangling

Basically, we will learn about four function if we discuss about wrangling the data

3 select

"select" is used to subset the data by selecting specific columns

Suppose we only want to work with two columns, **state** & **rate**, we use **select** to make the table cleaner:

```
murders_new <- select(murders_new, state, rate)
murders_new
```

```
> murders_new
          state      rate
1        Hawaii 0.5145920
2          Iowa 0.6893484
3 New Hampshire 0.3798036
4  North Dakota 0.5947151
5       Vermont 0.3196211
```

Note that variable **"murders_new"** only contain two columns, **state** & **rate**

Data Wrangling

Section 3.2 : Basic Data Wrangling

Basically, we will learn about four function if we discuss about wrangling the data

4 **pipe** or >%>

“%>%” is used to perform series of operations

Actually, we can put the **mutate**, **filter**, and **select** function together using **%>%**

```
murders_new <- murders %>% mutate(rate = murder_rate) %>%
+  select(state, rate) %>% filter(rate < 0.71)
murders_new
```

```
> murders_new
          state      rate
1        Hawaii 0.5145920
2          Iowa 0.6893484
3 New Hampshire 0.3798036
4  North Dakota 0.5947151
5       Vermont 0.3196211
```

Note that we can make the variable **“murders_new”** same as the previous slides although we run the shorter function

Data Wrangling

Section 3.2 : Creating Data Frames

When we perform data analysis with R we will find it necessary to create data frames.

Fortunately, we can simply do this with **data.frame** function

```
pl_table <- data.frame(club=c(“Man Utd”, “Arsenal”, “Chelsea”, “Liverpool”),
+                        points=c(93, 85, 85, 82),
+                        gd=c(68, 70, 51, 29),
+                        stringAsFactors = FALSE)
```

```
> pl_table
       club points gd
1   Man Utd     93 68
2   Arsenal     85 70
3   Chelsea     85 51
4 Liverpool     82 29
```

Note that we should type “stringAsFactors = FALSE” to make sure the entities we put into the data frames are classified as a “character”, not a “factor”

To validate this, we can use:

```
class(pl_table$club)
```

```
> class(pl_table$club)
[1] "character"
```

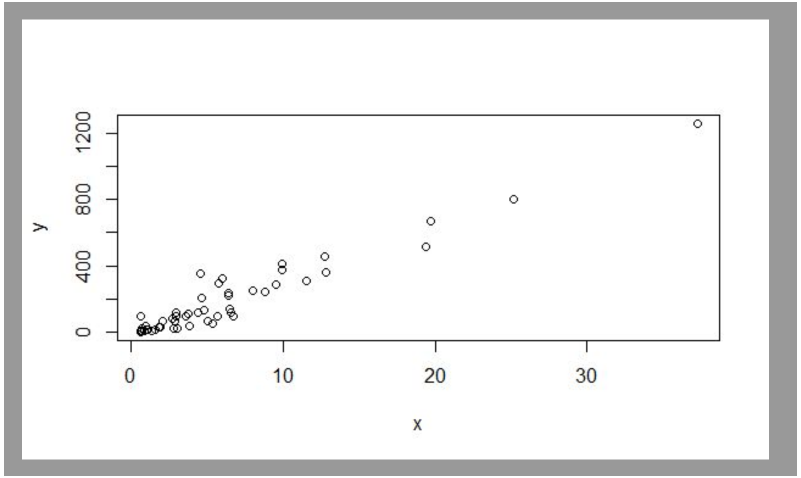
Here is a plot of total murders versus population

1 **Load the data**

```
> library(dslabs)
> data("murders")
```

2 **Create the index and scatterplot function**

```
> x <- murders$population/10^6
> y <- murders$total
> plot(x,y)
```



Histogram

Section 3.3 : Histogram

Here is the histogram plot on the murders data

1 **Load the data**

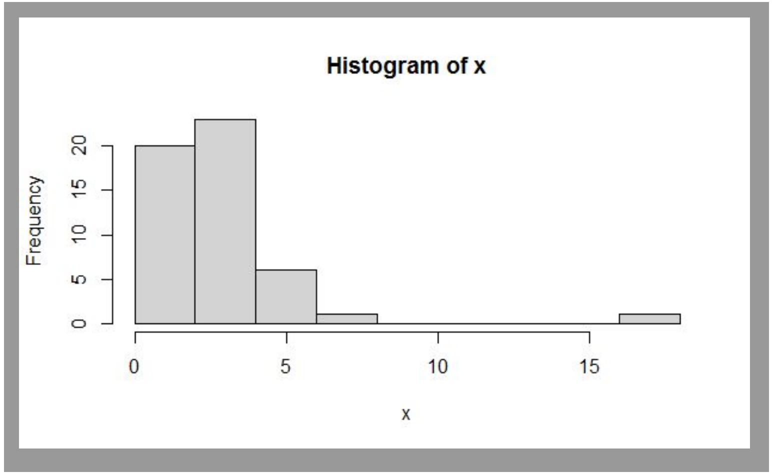
```
> library(dslabs)
> data("murders")
```

2 **Create the index and Histogram function**

```
> rate <- murders$total/murders$population * 100000
> hist(rate)
```

Histogram function in another way

```
> rate <- murders$total/murders$population * 100000
> x <- with(murders,rate)
> hist(x)
```



Boxplot

Section 3.3 : Boxplot

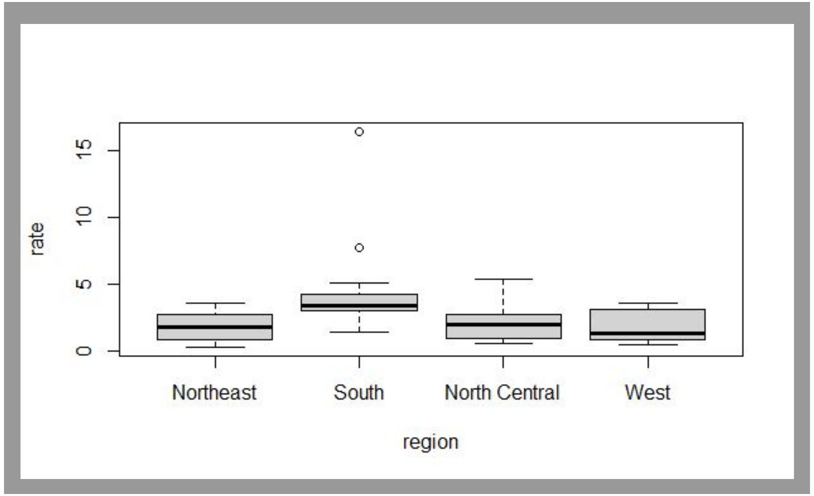
Boxplot which we will use by comparing different regions of the murders data

1 **Load the data**

```
> library(dslabs)
> data("murders")
```

2 **Create the index and Boxplot function**

```
> murders$rate <- with(murders,total/population*100000)
> boxplot(rate~region,data=murders)
```



Content of this module :

1. R Basics, Function, and Data Types
2. Vectors
3. Indexing, Data Wrangling, and Basic Plots
4. Programming Basics
5. R Markdown

This is what you will learn in this section...

Section 4 - overview

Section	What you will learn...
4.1	• Basic Conditional expression • Logical Vector
4.2	• Call Function • Pass Arguments to function • Return Variables /Objects from Function
4.3	• For-Loops • Sapply

Introduction to basic programming language

Section 4 - Introduction

R is a programming language...

- **R is not only a data exploration environment, but also a programming language**
  - R greatly encompasses the use of programming language to do a complex mathematical computation
  - For advance R programmer, the complex mathematical computation can be simplified through the packages that we frequently used now
  - For now, our focus is to implement the basic logics of Programming language, not to develop your skills as a software engineer.

...Basically, there are three key programming concepts

1. **Conditionals**
2. **Call Function**
3. **For-Loops**

Basic Conditional Expressions

Section 4.1 : If-else statement definition

Basic Form

```
if(boolean condition){
  expression
}else{
  alternative expression)
}
```

Meaning :

: The conditions that is evaluated

: The expression when the condition is satisfied

: The alternative expression when the condition is not satisfied

Simple example

```
> a <- 0
>
> if(a!=0){
+   print(1/a)
+ }else{
+   print("No Reciprocal for 0")
+ }
[1] "No Reciprocal for 0"
```

The code says that, if a is “not equal to zero”, then we shall print 1/a, if the value is outside the condition (in this context, if the value equal to 0), then we shall print “No Reciprocal for 0”

Basic Conditional Expressions

Section 4.1 : If-else statement example using US murders dataframe

Suppose we want to see whether there is any state that has murder rate lower than 0,5 or not

1 **Load the data, and define the variable**

```
library(dslabs)
data("murders")
murder_rate <- murders$total/murders$population * 10^5
```

2 **Create the index and the if-else function**

```
ind <- which.min(murder_rate)
```

Define a variable containing the index of the lowest rate

```
if(murder_rate[ind] < 0.5){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
```

Checking whether the lowest murder rate has rate below 0,5 or not

```
[1] "Vermont"
```

The output said that the state with the lowest murder rate has rate below 0,5

3 **Check if the rate is changed to 0,25**

```
if(murder_rate[ind] < 0.25){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
[1] "No state has murder rate that low"
```

There is not any state has rate below 0,25

Basic Conditional Expressions

Section 4.1 : ifelse function

There are other form of traditional if-else statement : **ifelse function**

This function takes three argument : a logical and two possible answer. If the value is TRUE, then the value in the second argument is returned and if FALSE, the value in the third argument is returned.

Simple example

```
> a <- 0
> ifelse(a > 0, 1/a, NA)
[1] NA
```

The same as previous example, the code says that, if a is "not equal to zero", then we shall print 1/a, if the value is outside the condition (in this context, if the value equal to 0), then we shall print "NA"

Example in using vector

One great advantage of this function is the compatibility to assess a vector in conditional forms

```
> a <- c(0, 1, 2, -4, 5)
> result <- ifelse(a > 0, 1/a, NA)
> result
[1]  NA 1.0 0.5  NA 0.2
```

the code says that, for every value in a that is "greater than zero", then we shall print 1/a, if the value is outside the condition (in this context, if the value less and/or equal 0), then we shall print "NA"

Basic Conditional Expressions

Section 4.1 : any and all function

"Any" function	"All" function
The any function takes a vector of logicals and returns TRUE if any of the entries is TRUE	The all function takes a vector of logicals and returns TRUE if all of the entries is TRUE

Here is an example

```
z <- c(TRUE, TRUE, FALSE)
any(z)
#> [1] TRUE
all(z)
#> [1] FALSE
```

Basic Functions

Section 4.2 : call functions definition

Function is a way to simplify a line of task through a definitive operator. It is used mainly to reduce the time and effort of writing a line of code, especially in a repeatable task

Basic form

```
my_function <- function(VARIABLE_NAME){
  perform operations on VARIABLE_NAME and calculate VALUE
  VALUE
}
```

Basic Functions

Section 4.2 : call functions example

As you become more experienced, you will find yourself needing to perform the same certain operation over and over again. The simple example of this is computing the "**Average Function**"

Normal way to compute average

```
> x <- c(1,2,3,4,5)
> avg <- sum(x)/length(x)
> avg
[1] 3
```

This is the normal way : you need to find sum(x) and length(x) and divide those two scores to find the average of those vector x. However, in the long term, this will cost us much time because you need to repeat it over and over.

How you make a basic function

```
> avg <- function(x){
+     s <- sum(x)
+     n <- length(x)
+     s/n
+ }
> avg(x)
[1] 3
```

This is the variable included

This is the operation

This is how you read it :
"Avg" is a function of x (so you can specify any variable within avg) in which involves computing the sum and length, therefore divide the sum and the length of it.

The "avg" will be a function used to compute the average.

For Loops

Section 4.3 : for loops definition

For loops is a way to do iterative processing using a function with some condition that you set

Simplest example

```
for(i in 1:5){
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

This is the condition

This is the operation

For Loops

Section 4.3 : for loops example

Suppose that you want to prove that the sum of series 1+2+3+…+n is n(n+1)/2. You can check the sanity of that function by proving it using the help of call function and for loops.

```r
#1 Define a call function
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
```

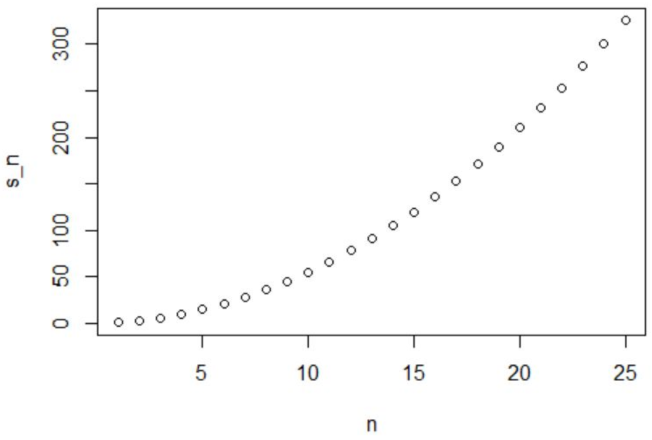
This is the call function. However, to compute the various value of n, we can't do the function for say like 25 times, we need for loops function to do that

```r
#2 Define a for loop
m <- 25
s_n <- vector(length = m) # create an empty vector
for(n in 1:m){
  s_n[n] <- compute_s_n(n)
}
```

This is the for loop function that is used to perform the operation 25 times with the m ranges from 1 until 25

```r
#3 Plot the n with sn
n <- 1:m
plot(n, s_n)
```

Scatter plot



For Loops

Section 4.3 : Sapply function

There is a more simple way to perform a “for loop” operation, especially for a vector-wise object : The function is called “sapply”. This function work best for an arithmetics operation that involves a vector

Suppose you want to do a sqrt function to all object within a vector ranging from 1 until 10

```
x <- 1:10
sapply(x, sqrt)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

Content of this module :

1. R Basics, Function, and Data Types
2. Vectors
3. Indexing, Data Wrangling, and Basic Plots
4. Programming Basics
5. R Markdown

The final product of data analysis is often an **organized report** from a built in R. Hereby we introduce you to **R Markdown** to make your report can be easily examined by other people. Please noted : **to standardize the output of your task, we will use R Markdown**

This is an example of R Markdown output :

Tugas 1 : R Basics

Statdas 03 - Group 07

09/10/2020

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

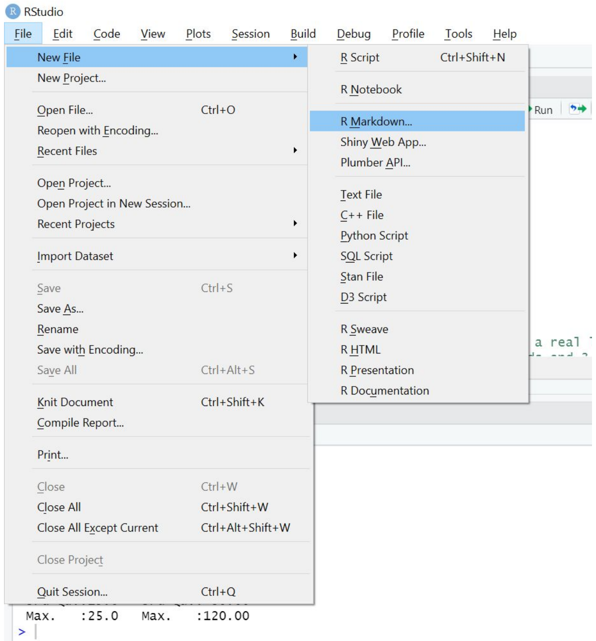
```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

R Markdown : the reproducible output for R

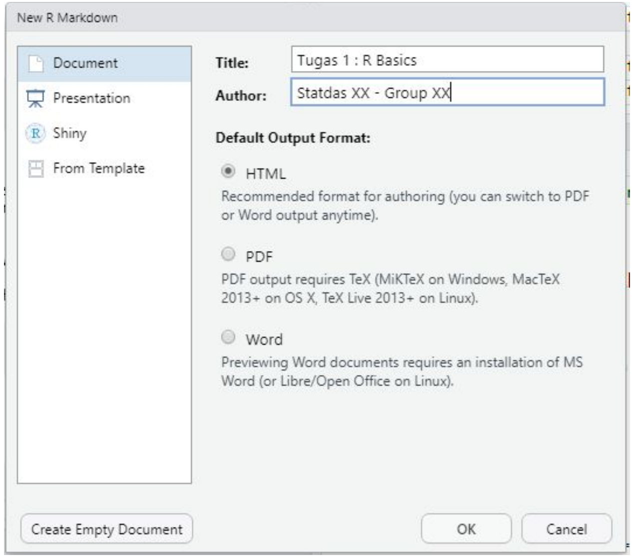
Section 5 - R Markdown how to...

To produce a R Markdown output, here are the important steps :

1. Click File → New FIle → R Markdown...



2. You will get this kind of pop up... please noted the details as follows :



2a) Title Format :
Tugas 1 : R Basics

2b) Author format :
Statdas (01/02/03/KKI) - Group XX
(i.e Statdas 03 - Group 07)

2c) Default Output Format :
Choose HTML

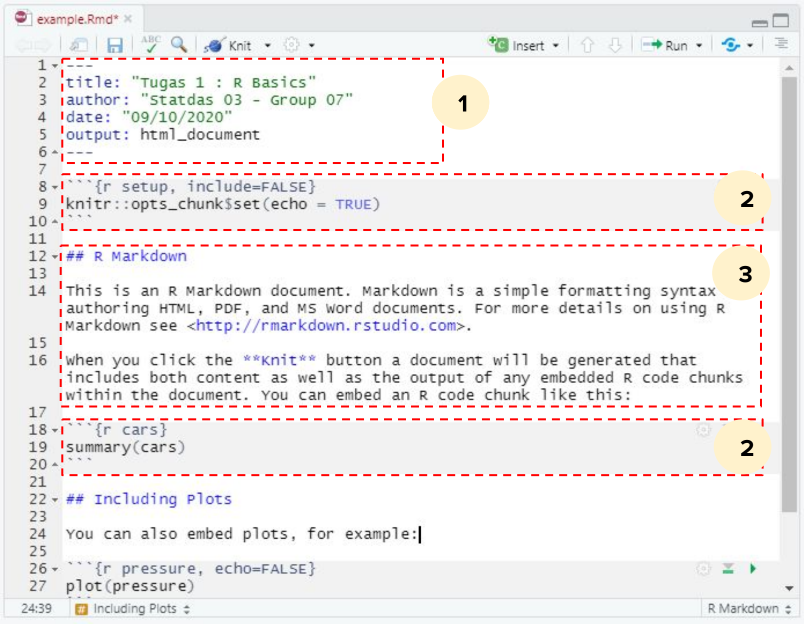
Then, click OK

R Markdown : the reproducible output for R

Section 5 - R Markdown how to...

To produce a R Markdown output, here are the important steps :

3. You will open a R Markdown script



4. Fill the script by taking a note of these details :

1 **The header**
The thing between "---" is the header. In your task later, it will much important for you to keep this style in format. Do not change that

2 **R Code Chunk**
This is where you will put your code. Your code will be evaluated and the result will be included in that position in your report. To add your own chunk, you can press Command+Option+I on mac and Ctrl+alt+I on windows

3 **Normal Text**
This is where you will explain your findings, you can write up your analysis anywhere outside the R code chunk and the header

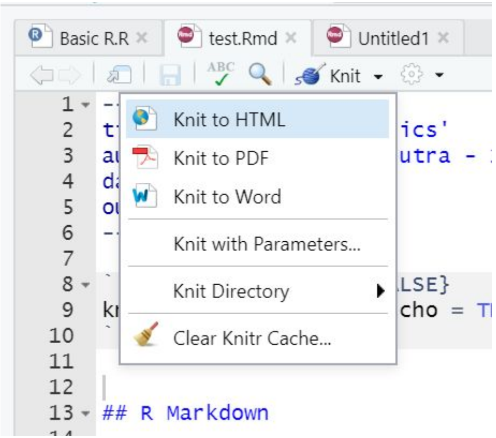
Important Note
You can learn how to fill the details in https://www.markdowntutorial.com/
Those link provides how to write/edit your R Markdown easily.

R Markdown : the reproducible output for R

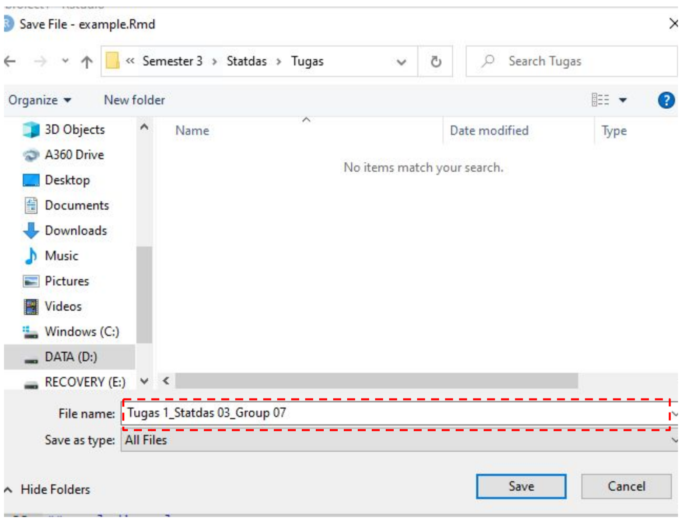
Section 5 - R Markdown how to...

To produce a R Markdown output, here are the important steps :

5. If you have already finished on writing your code and analysis, then click “knit” → knit to HTML



6. Type the file name and then click ok



File Name
Tugas x_Statdas 01/02/03/KKI_Group xx
(i.e Tugas 1_Statdas 03_Group 07)

Assignment

Due date on Oct 11th, 2020 at 23.59 WIB

1

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

What would happen if we are subsetting which.max() function into the x vector? and why does that happen?

2

```
1,½,⅓,¼,⅕,...1/100
```

Define an object **x** that contains the number 1 through 100. Compute the sum `1,½,⅓,¼,⅕,...1/100.`

3

Install the package “dslabs” and load the dataset “gapminder”. Define the variable ‘region’ as *gm_reg.*

a) What is the data type of *gm_reg* and b) how many entries are there in the variable *fert?*

4

Install the packages “dplyr” and load the dataset “gapminder”.

a) Filter the data so that it only displays rows where the country is ‘Indonesia’

b) Define the result as *data_indo,* then print it!

Assignment

Due date on Oct 11th, 2020 at 23.59 WIB

5 Load a dataset called “iris”. Which species has the highest median value of Petal Length? Visualize your argument. (clue: Create a boxplot)

6 From the dataset “iris”, is the sepal width with the highest frequency larger or smaller than 3.0? Visualize your argument.

Assignment

Due date on Oct 11th, 2020 at 23.59 WIB

Answer the questions and explain your answers in a **R Markdown**, please include the question number.

Your R Markdown should be like the following:

1 # Question 1

2 # Explain your answer

3 Function(x)

Submit your assignment with the format: **Tugas x_Statdas 01/02/03/KKI_Group x.zip** (including both the **Html file** and **RMD file**, grouped in a zipped folder) to **https://bit.ly/3n3sl8h**

Please submit your own work without **plagiarizing others**', If we detect any plagiarism, we will cut up your score

“In god we trust, all others must bring data” - W. Edward Demings

Thank You

SQE - 2020