Business Analytics

Introduction to R

Proschool
An IMS Initiative

Introduction – R and RStudio

R:

- R is a computer language for statistical computing and is becoming the leading language in data science and statistics.
- Today, R is the tool of choice for data scientists in every industry and field.
- It open source freely available software and very widely used by professional statisticians.
- R has a large collection of intermediate tools and excellent graphical tools for data analysis.

Introduction – R and RStudio

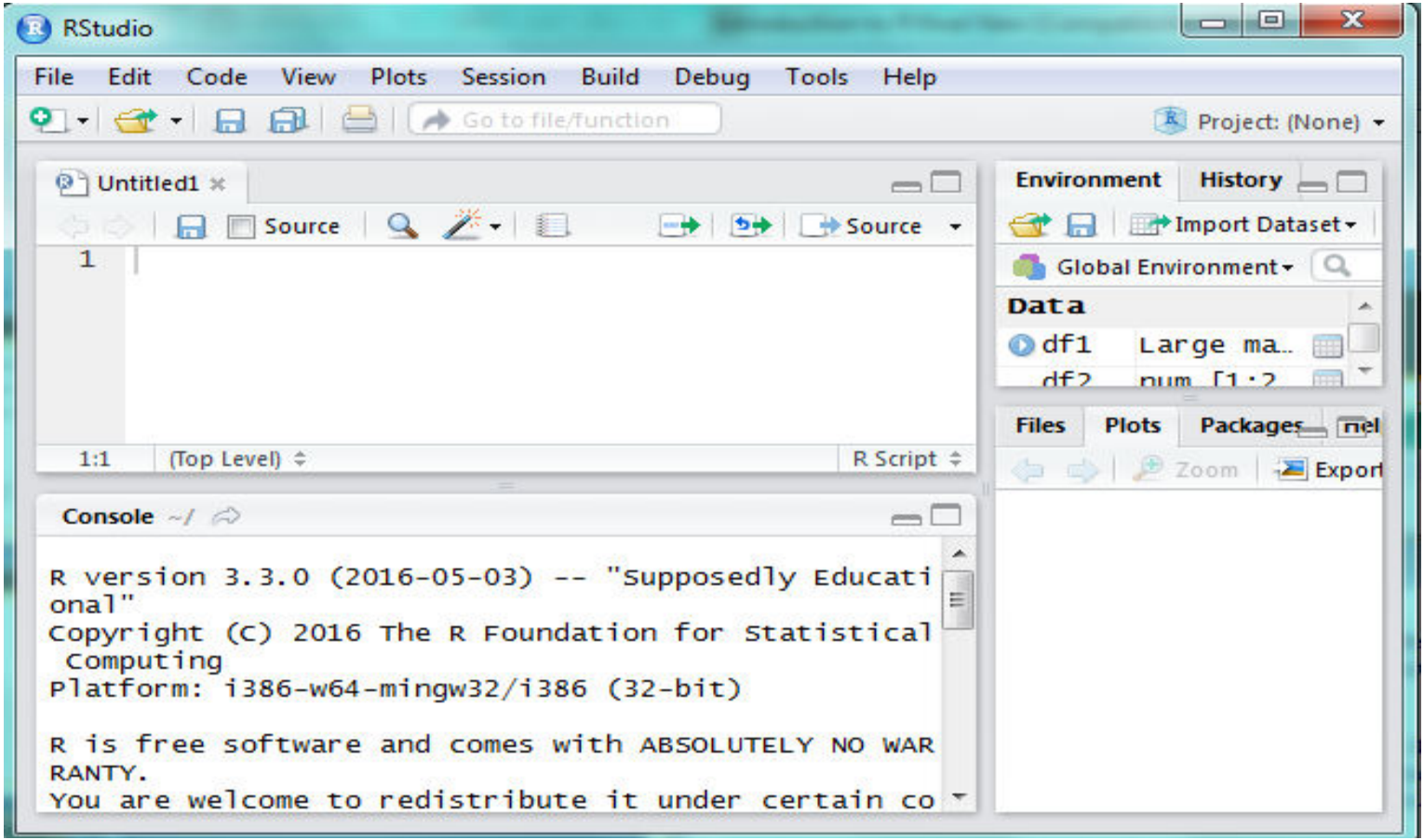
RStudio:

- RStudio is more widely used than R
- RStudio is an Integrated Development Environment (IDE) for R.
- It includes a console, syntax- highlighting editor that supports direct code execution as well as tools for plotting, history, debugging & workspace management.

Installation

- RStudio requires R version 2.11.1 or higher.
- RStudio version 0.99.902
- Latest version can be downloaded from the link below,
- https://www.rstudio.com/products/rstudio/download/

R Command Screen



Basic RStudio commands

- '>' is called the prompt
- If a command is too long to fit on a line, a + is used for the continuation prompt.

Arithmetic Operators

Operator	Function
+	Addition
-	Subtraction
/	Division
*	Multiplication
^ or **	Exponentiation

Basic RStudio commands as Calculator

Examples of arithmetic operations in RStudio

```
> 15+23    # Addition
[1] 38
> 45-36    # Substraction
[1] 9
> 25/7     # Division
[1] 3.571429
> 45*8     # Multiplication
[1] 360
> 5^3      # Exponentiation
[1] 125
```

Variable Assignment

Values are assigned to variables with the assignment operator '<-' or '='.

```
> x = 5    # Assigning value 5 to x
> x
[1] 5
```

Functions

R functions are invoked by its name, then followed by the parenthesis, and zero or more arguments. The following command combines five numeric values into a vector.

```
> z=c(1,2,3,4,5)
> z
[1] 1 2 3 4 5
```

Extension Package

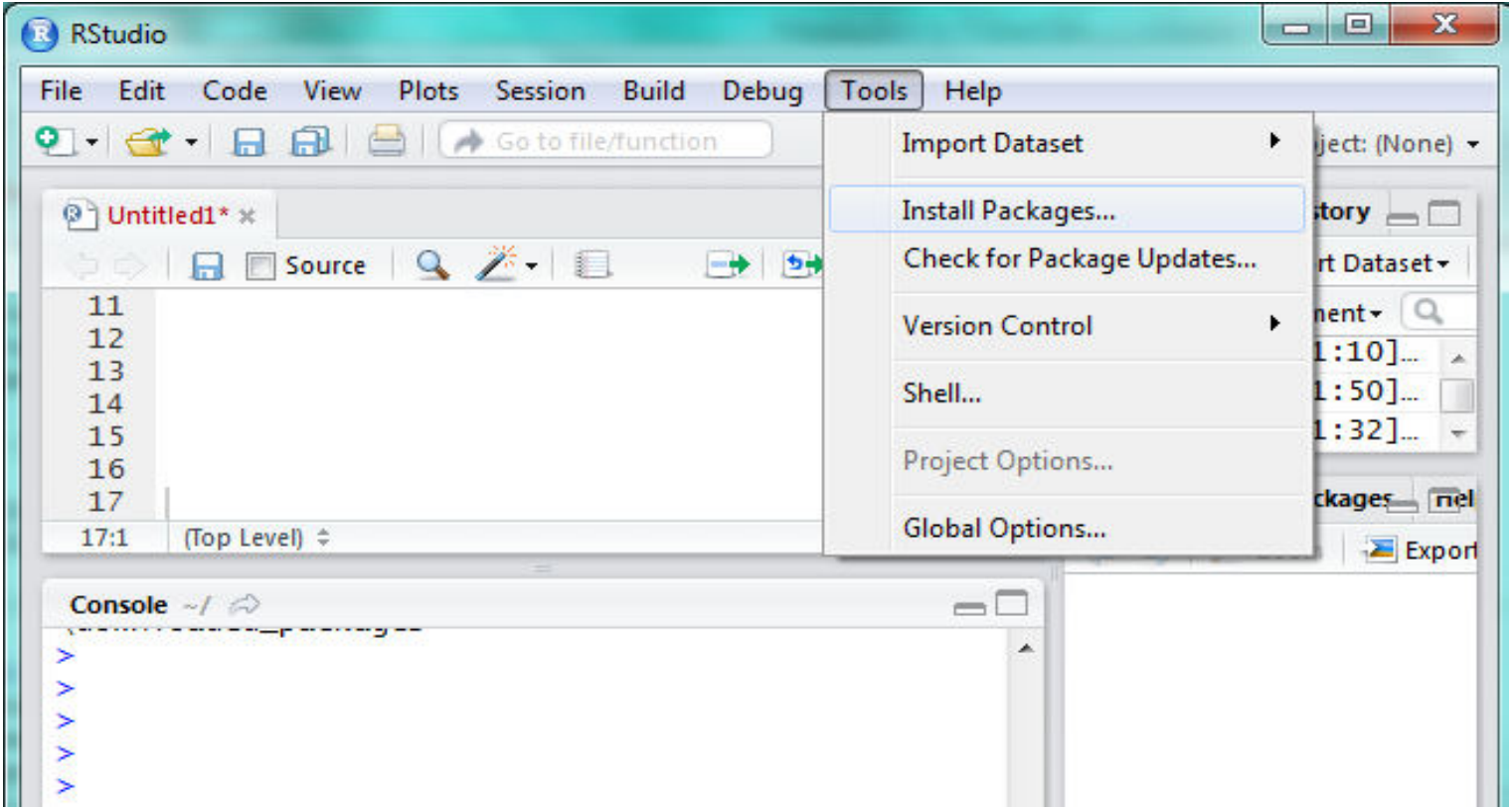
- Sometimes we need additional functionality beyond those offered by the core R library.
- In order to install an extension package, you should invoke the **install.packages function** at the prompt and follow the instruction.

```
> install.packages("packageName")
```

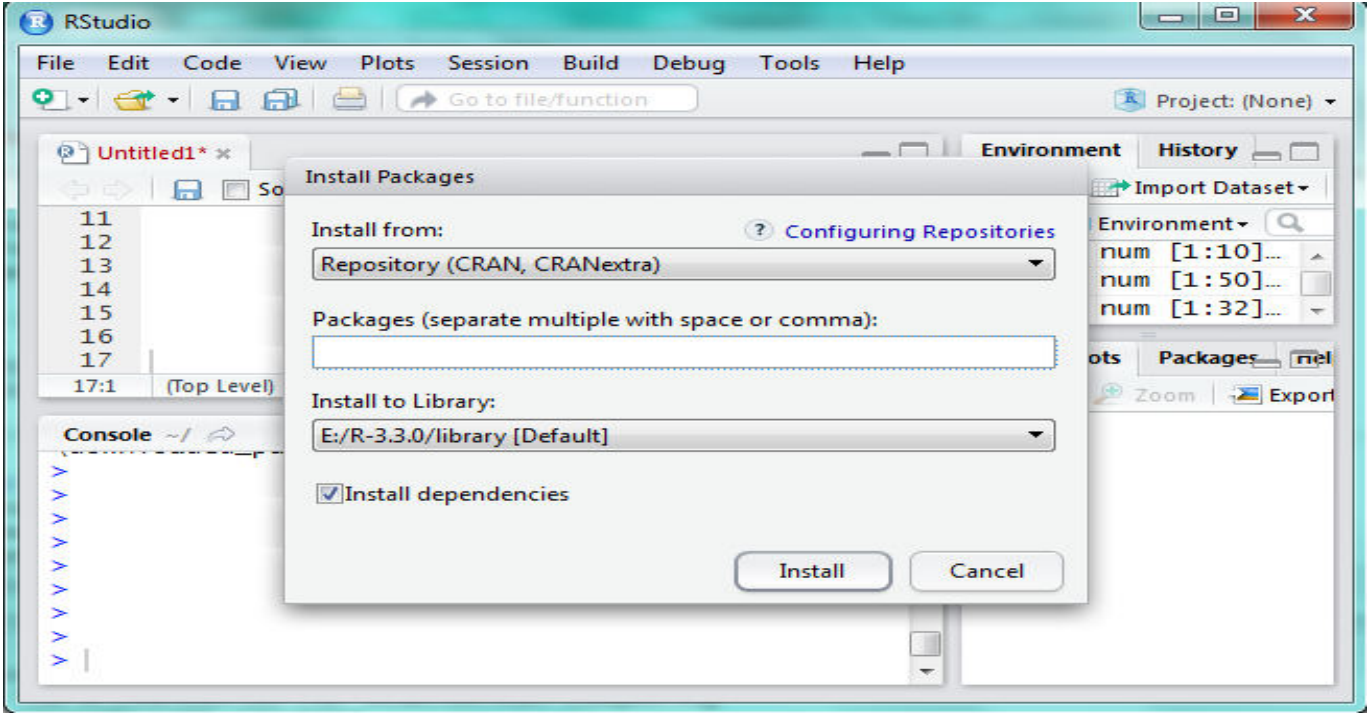
Or

- Select install packages from Tools drop down menu

Install Packages



Install Packages



Enter the name of the package you want to install.

Getting Help

- R provides an inbuilt facility for getting more information on any specific feature or named function.
- For example, entering ?c or help(c) at the prompt gives documentation of the function c in R.

```
> help(c)
```

- If there is no help available on any function or feature it displays an error message.
- By default the function help searches in the packages which are loaded in memory.
- The function try.all.packages allows searching all packages.

```
> help(ts,try.all.packages=TRUE)
```

Data Types

- In analysis of statistical data we use different types of data. For manipulating such data, R supports following data types

 - logical : It is a Boolean type data whose value can be TRUE or FALSE.

 - numeric : It can be real or integer.

 - complex : It consists of real and imaginary numbers.

Logical

A **logical** value is often created via comparison between variables.

```
> x=1; y=2   # sample values
> z=x > y    # is x larger than y?
> z          # print the logical value
[1] FALSE
> class(z)   # print the class name of z
[1] "logical"
```

Numeric

- Decimal values are called **numeric** in R.
- It is the default computational data type.
- If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
> x=10.65    # assign a decimal value to x
> x
[1] 10.65
> class(x)  # print the class name of x
[1] "numeric"
```

Integer

- In order to create an **integer** variable in R, we need invoke the as.integer function. We can be assured that y is indeed an integer by applying the is.integer function.

```
> y=as.integer(4)
> y
[1] 4
> class(y)        # print the class name of y
[1] "integer"
> is.integer(y)   # is y an integer?
[1] TRUE
```

- We can coerce a numeric value into an integer with the same as.integer function

```
> as.integer(4.76)   # coerce a numeric value
[1] 4
```

Complex Numbers

A **complex** value in R is defined via the pure imaginary value *i.*

```
> z=1+5i        # create a complex number
> z
[1] 1+5i
> class(z)      # print the class name of z
[1] "complex"
```

Character

- A **character** object is used to represent string values in R.
- We convert objects into character values with the as.character() function:

```
> x=as.character(3.14)
> x                          # print the character string
[1] "3.14"
> class(x)                   # print the class name of x
[1] "character"
```

- Two character values can be concatenated with the paste function.

```
> fname="Joe";lname="Smith"
> paste(fname,lname)
[1] "Joe Smith"
```

Vectors

- R operates on named data structures. The simplest such structure is numeric vector.
- Numeric vector is a single entity consisting of an ordered collection of numbers.
- To create a vector named x containing 5 numbers 2,5,8,1,35 use following command

```
> x = c(2,5,8,1,35)
> x
[1]  2  5  8  1 35
```

This is assignment using function c(). Assignment can also be made by using assign function.

```
> assign("x",c(2,5,8,1,35))
> x
[1]  2  5  8  1 35
```

Vector

- A vector can contain character strings.

```
> g=c("John","Albert","Gracy")
> g
[1] "John"   "Albert" "Gracy"
```

- Incidentally, the number of members in a vector is given by the length function.

```
> length(c("John","Albert","Gracy"))
[1] 3
```

Combining Vectors

- Vectors can be combined via the function c.
- For example, the following two vectors n and s are combined into a new vector containing elements from both vectors.

```
> n=c(2,3,5)
> s=c("aa","bb","cc","dd","ee")
> c(n,s)
[1] "2"  "3"  "5"  "aa" "bb" "cc" "dd" "ee"
```

Vector Arithmetic

- Arithmetic operations of vectors are performed member-by-member, *i.e.*, member wise.
- For example, suppose we have two vectors a and b.

```
> a=c(1,3,5,7)
> b=c(1,2,4,8)
```

- Then, if we multiply a by 5, we would get a vector with each of its members multiplied by 5.

```
> 5*a
[1]  5 15 25 35
```

- And if we add a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b.

```
> a+b
[1]  2  5  9 15
```

Vector Arithmetic

Recycling Rule

If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors u and v have different lengths, and their sum is computed by recycling values of the shorter vector u.

```
> u=c(10,20,30)
> v=seq(1,9,1)      # creates a sequence of numbers from 1 to 9
> u+v
[1] 11 22 33 14 25 36 17 28 39
```

Vector Index

- We retrieve values in a vector by declaring an index inside a *single square bracket* "[]" operator.
- For example, the following shows how to retrieve a vector member.

```
> s=c("aa","bb","cc","dd","ee")
> s[3]
[1] "cc"
```

Vector Index

Negative Index

If the index is negative, it would strip the member whose position has the same absolute value as the negative index. For example, the following creates a vector slice with the third member removed.

```
> s[-3]
[1] "aa" "bb" "dd" "ee"
```

Out-of-Range Index

If an index is out-of-range, a missing value will be reported via the symbol NA.

```
> s[10]
[1] NA
```

Numeric Index Vector

- A new vector can be sliced from a given vector with a **numeric index vector**, which consists of member positions of the original vector to be retrieved.
- Here it shows how to retrieve a vector slice containing the second and third members of a given vector's.

```
> s=c("aa","bb","cc","dd","ee")
> s[c(2,3)]
[1] "bb" "cc"
```

Numeric Index Vector

Duplicate Indexes

```
> s[c(2,3,3)]
[1] "bb" "cc" "cc"
```

Out-of-Order Indexes

```
> s[c(2,1,3)]
[1] "bb" "aa" "cc"
```

Range Index

```
> s[2:4]
[1] "bb" "cc" "dd"
```

Named Vector Members

- We can assign names to vector members.
- For example, the following variable v is a character string vector with two members.

```
> v=c("Mary","Sue")
> v
[1] "Mary" "Sue"
```

- We now name the first member as First, and the second as Last.

```
> names(v)=c("First","Last")
> v
 First   Last
"Mary"  "Sue"
```

Named Vector Members

- Then we can retrieve the first member by its name.

```
> v["First"]
 First
"Mary"
```

- Furthermore, we can reverse the order with a character string index vector.

```
> v[c("Last","First")]
  Last  First
 "Sue" "Mary"
```

Matrix

Matrix

- A **matrix** is a collection of data elements arranged in a two-dimensional rectangular layout. The following is an example of a matrix with 2 rows and 3 columns.

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

- We reproduce a memory representation of the matrix in R with the matrix function.

Matrix

The data elements must be of the same basic type.

```
> A = matrix(
+   c(2,4,3,1,5,7),      # the data elements
+   nrow=2,              # number of rows
+   ncol=3,              # number of columns
+   byrow=TRUE)          # fill the matrix by rows
> A                      # print the matrix
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

- An element at the m^{th} row, n^{th} column of A can be accessed by the expression A[m, n].

```
> A[2,3]        # the element at 2nd row, 3rd column
[1] 7
```

- The entire m^{th} row A can be extracted as A[m,].

```
> A[2,]         # the 2nd row
[1] 1 5 7
```

- Similarly, the entire n^{th} column A can be extracted as A[,n].

```
> A[,3]         # the 3rd column
[1] 3 7
```

We can also extract more than one rows or columns at a time.

```
> A[,c(1,3)]   # the first and third columns
     [,1] [,2]
[1,]    2    3
[2,]    1    7
```

- If we assign names to the rows and columns of the matrix, than we can access the elements by names.

```
> dimnames(A)=list(
+   c("row1","row2"),           # row names
+   c("col1","col2","col3"))    # column names
> A                             # print A
     col1 col2 col3
row1    2    4    3
row2    1    5    7
>
> A["row2","col3"]              # element at 2nd row, 3rd column
[1] 7
```

Matrix Construction

- There are various ways to construct a matrix. When we construct a matrix directly with data elements, the matrix content is filled along the column orientation by default.
- For example, in the following code snippet, the content of B is filled along the columns consecutively.

```
> B=matrix(
+   c(2,4,3,1,5,7),
+   nrow=3,
+   ncol=2)
> B                    # matrix B  has 3 rows and 2 columns
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
```

Matrix Transpose

- We construct the **transpose** of a matrix by interchanging its columns and rows with the function t .

```
> t(B)          # transpose of B
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
>
```

Combining Matrices

- The columns of two matrices having the same number of rows can be combined into a larger matrix. For example, suppose we have another matrix C also with 3 rows.

```
> C=matrix(
+   c(7,4,2),
+   nrow=3,
+   ncol=1)
> C            # C has 3 rows
     [,1]
[1,]    7
[2,]    4
[3,]    2
```

- Then we can combine the columns of B and C with cbind.

```
> cbind(B,C)
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
```

- Similarly, we can combine the rows of two matrices if they have the same number of columns with the rbind function.

```
> D=matrix(
+   c(6,2),
+   nrow=1,
+   ncol=2)
> D             # D has 2 columns
     [,1] [,2]
[1,]    6    2
>
> rbind(B,D)
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
[4,]    6    2
```

Deconstruction

- We can deconstruct a matrix by applying the c function, which combines all column vectors into one

```
> c(B)
[1] 2 4 3 1 5 7
```

The Workspace

- The workspace is your current **R** working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions).

- At the end of an **R** session, the user can save an image of the current workspace that is automatically reloaded the next time **R** is started.

- Commands are entered interactively at the **R** user prompt.

- **Up** and **down arrow keys** scroll through your command history.

The Workspace

- You will probably want to keep different projects in different physical directories. Here are some standard commands for managing your workspace.

IMPORTANT NOTE FOR WINDOWS USERS:

- **R** gets confused if you use a path in your code like
  c:\mydocuments\myfile.txt

- This is because R sees "\" as an escape character. Instead, use
  c:/mydocuments/myfile.txt

R Command for Working Directory


```
> getwd()    # print the current working directory
[1] "C:/Users/USER/Documents"
>
> ls()       # list the objects in current workspace
```

```
> setwd(mydirectory)     # change to mydirectory
```

work with your previous commands

```
> history()                 # displays last 25 commands
> history(max.show=Inf)     # displays all previous commands
```

save your command history

```
> savehistory(file="myfile")     # default is ".Rhistory"
```

R Command for Working Directory

recall your command history

```
> loadhistory(file="myfile")      # default is ".Rhistory"
```

Data Frame

- A **data frame** is used for storing data tables. It is a list of vectors of equal length.

  Example :

  Following are the height (in cms) and weight (in kgs) of 10 boys. Prepare a data frame of height and weight.

```
> height = c(137, 140, 165, 156, 172)
> weight = c(45, 51, 59, 54, 63)
> data.frame(height, weight)
  height weight
1    137     45
2    140     51
3    165     59
4    156     54
5    172     63
```

Importing Data

- Importing data into **R** is fairly simple.

  Comma Delimited File (.CSV extension)

```
> mydata<-read.csv("D:/Analytics/Anaytics PPT/Linear Regression/Housing.csv",
+                  header=TRUE)
```

  Text File(.txt extension)

```
> mydata<-read.table("D:/Analytics/Anaytics PPT/Linear Regression/Housing.txt",
+                  header=TRUE)
```

From Excel (.xlsx Extension)

- One of the best ways to read an Excel file is to export it to a comma delimited file and import it using the method above.

- Alternatively you can use the **xlsx** package to access Excel files. The first row should contain variable/column names.

- read in the first worksheet from the workbook *myexcel.xlsx*

- first row contains variable names

```
> library(xlsx)
> mydata1=read.xlsx("C:/myexcel.xlsx",1)
```

- read in the worksheet named *mysheet*

```
> mydata1=read.xlsx("C:/myexcel.xlsx",sheetName="mysheet")
```

From SAS

- Save SAS dataset in trasport format

```
libname out xport 'c:/mydata.xpt';
data out.mydata;
set sasuser.mydata;
run;
```

- In R

```
> library(Hmisc)
> mydata=sasxport.get("C:/mydata.xpt")
```

character variables are converted to R factors

Sorting Data

- To sort a data frame in R, use the **order( )** function.

- By default, sorting is **ASCENDING**. Prepend the sorting variable by a minus sign to indicate DESCENDING order.

Here are some examples.

Sorting examples using the mtcars dataset

```
> attach(mtcars)
```

sort by mpg

```
> newdata=mtcars[order(mpg),]
```

Sorting Data

- # sort by mpg and cyl

```
> newdata=mtcars[order(mpg,cyl),]
```

- #sort by mpg (ascending) and cyl (descending)

```
> newdata=mtcars[order(mpg,-cyl),]
```

```
> detach(mtcars)
```

Renaming Variable (Header)

To rename a variable :

```
> library(reshape)
> mydata=rename(mydata,c(price="Cost"))
```

To Exit R :

```
> q()      # Exit R
```

Thank You