Welcome to

Data Analytics with R

Day 1

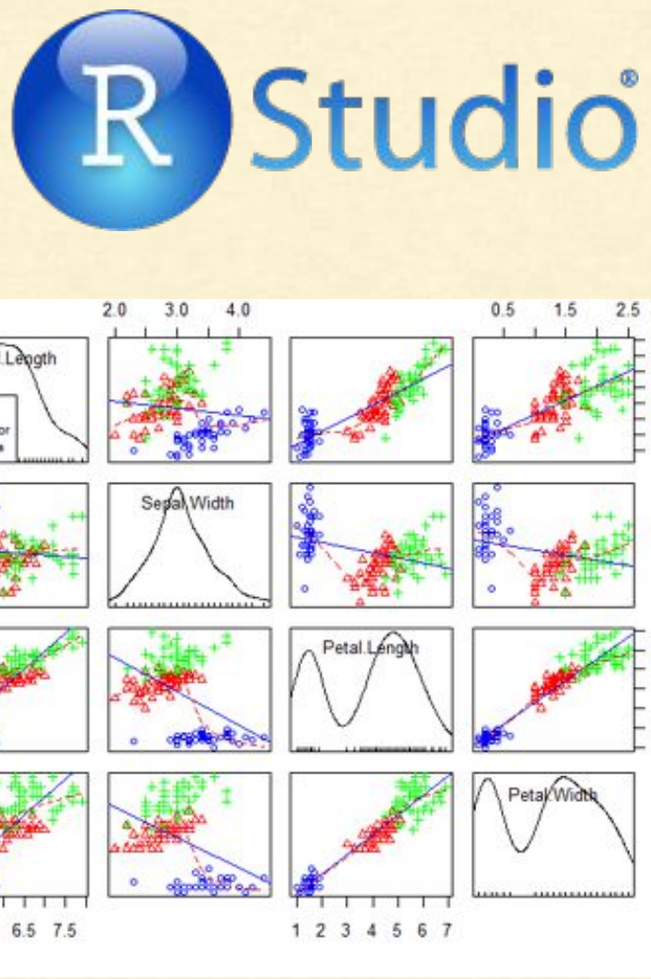
ABOUT INSTRUCTOR - Abhinav Singh

2016	**CloudxLab**	Building platform for practicing Big Data Technologies
2015	**Byjus**	#1 Edtech application in India on PlayStore
2015		
2013	**Specadel**	Developed platform for disrupting indian education
2012		
2009	**HashCube**	Developed #1 Sudoku Game on Facebook

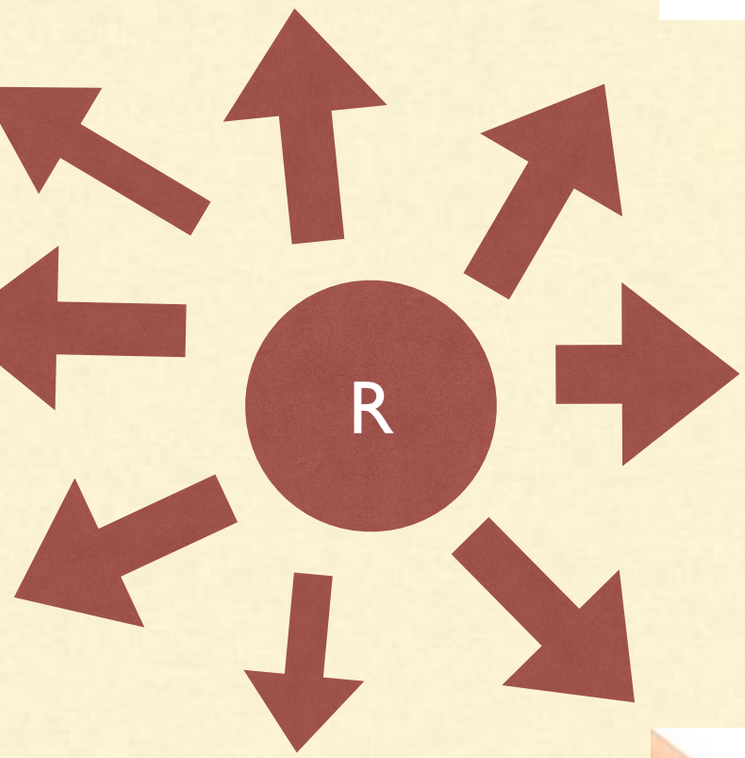
WHAT IS R?

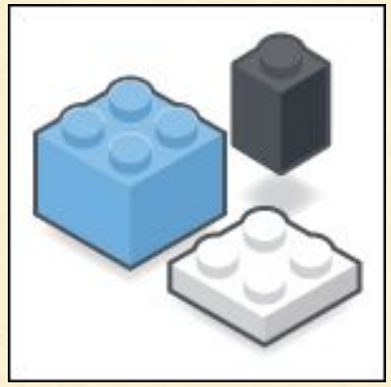
Based on S,
Bell Labs

Modeling
Analysis
Clustering
Classification
Statistics



Visualization & Graphs





Extensible
C, C++, Java
Python
.Net

open source

Powered by GNU

Developer
Community

CRAN

POWER OF R

Statistics Modeling
- Various Regression Models
- Time series and forecasting
- Design of Experiments
- Survival Analysis
- Prob. Distributions

Linear & Non-Linear Optimization
- Linear Programming
- Stochastic Optimization
- Genetic Algorithms

Machine Learning
- Baysian Methods
- SVM/SVR
- Decision Trees
- Ensemble Models
- Neural Networks

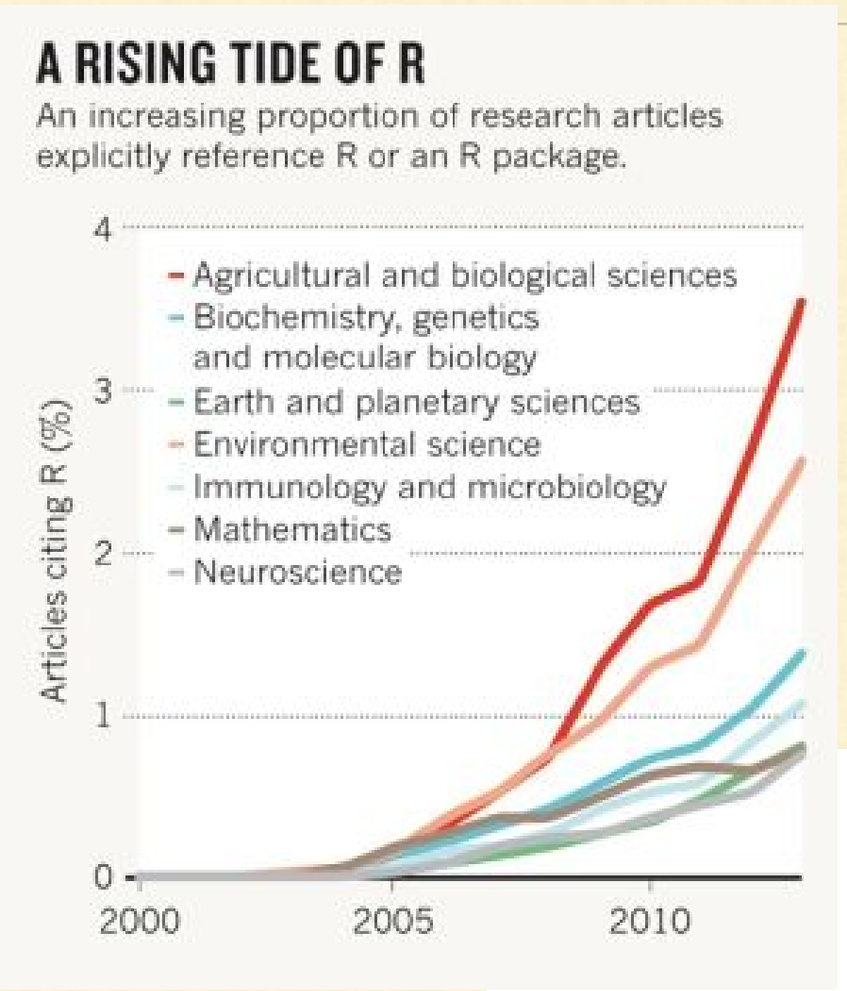
Heuristics Programming
- Easy connectivity to Python / JAVA / C / C++ / Fortran

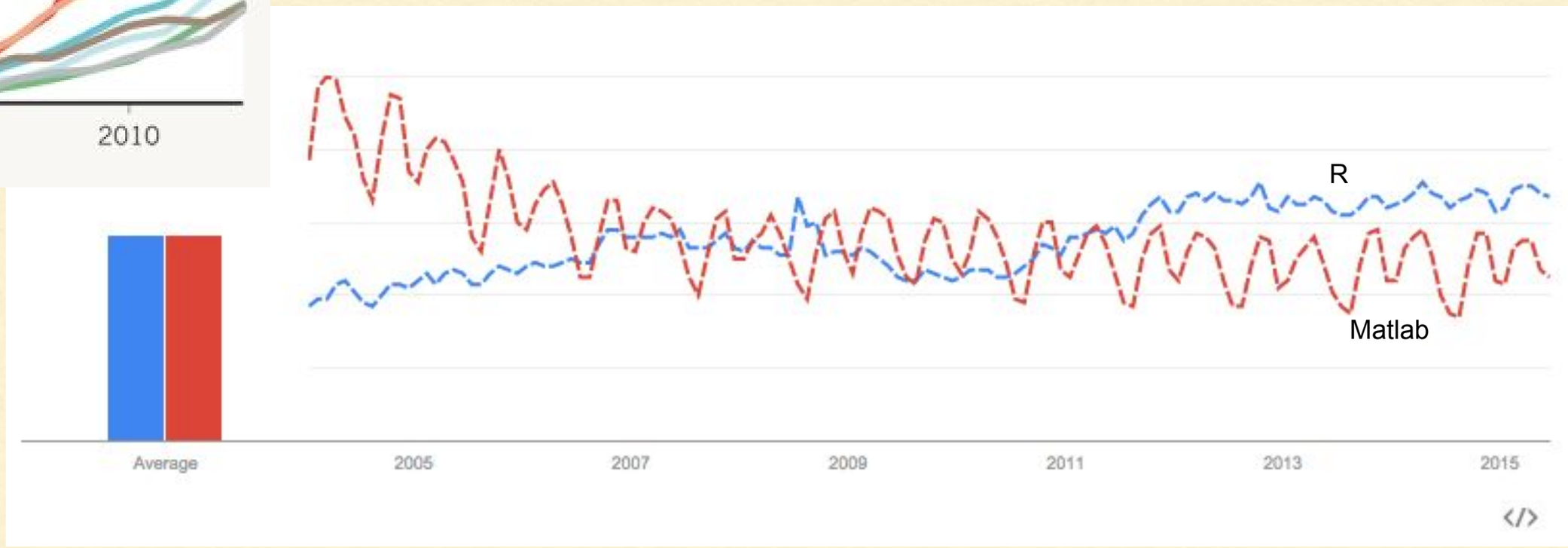
R

Big Data Analytics
- R – Hadoop
- R – SAP HANA
- R – MongoDB (NoSQL databases)
- R – MySQL
- R - Excel

Visualization
- Tableau + R
- Static Graphics using GGPLOT2 and D3
- Dynamic Graphics using GoogleViz etc
- Interactive Graphics using GGOBI

Trends


A RISING TIDE OF R
An increasing proportion of research articles explicitly reference R or an R package.
Articles citing R (%)
4
3
2
1
0
2000
2005
2010
Agricultural and biological sciences
Biochemistry, genetics and molecular biology
Earth and planetary sciences
Environmental science
Immunology and microbiology
Mathematics
Neuroscience


R
Matlab
Average
2005
2007
2009
2011
2013
2015

GETTING STARTED

R (http://www.r-project.org/)

Install Rstudio (www.rstudio.com)

EquiSkill Lab for RStudio

http://lab1.equiskill.com

Login with your credentials

Setup the project

- All the source code, data files and presentation is stored in Github repository
- In EquiSkill lab, Go to File -> New Project -> Version Control -> Git -> Project Setup
- Respository url - https://github.com/singhabhinav/R-introduction
- Project Directory name: R-introduction

RSTUDIO

An IDE that was built just for R

- Syntax highlighting, code completion, and smart indentation
- Execute R code directly from the source editor
- Quickly jump to function definitions

Bring your workflow together

- Integrated R help and documentation
- Easily manage multiple working directories using projects
- Workspace browser and data viewer

Powerful authoring & Debugging

- Interactive debugger to diagnose and fix errors quickly
- Extensive package development tools
- Authoring with Sweave and R Markdown

RStudio

Go to file/function

Project: (None)

Untitled.Rmd | TEst2.Rmd | mtcars | ozone

5 observations of 6 variables

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5

Console ~/

```
R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

During startup - Warning messages:
1: Setting LC_CTYPE failed, using "C"
2: Setting LC_COLLATE failed, using "C"
3: Setting LC_TIME failed, using "C"
4: Setting LC_MESSAGES failed, using "C"
5: Setting LC_MONETARY failed, using "C"
[Workspace loaded from ~/.RData]

>
```

Environment | History

Import Dataset | Clear | List

Global Environment

Data

A	num [1:2, 1:3] 2 1 4 5 3 7
arr	int [1:10, 1:10] 1 2 3 4 5 6 7 8 9 10 ...
B	num [1:3, 1:2] 2 4 3 1 5 7
C	num [1:3, 1] 7 4 2
c_painters	6 obs. of 5 variables
D	num [1:2, 1:3] 2 1 4 5 3 7
df	3 obs. of 3 variables
diffr	1 obs. of 11 variables
dq	1 obs. of 11 variables
E	num [1, 1:3] 7 4 2

Files | Plots | Packages | Help | Viewer

R: Print Values | Find in Topic

print {base} R Documentation

Print Values

Description

`print` prints its argument and returns it *invisibly* (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new `class`es.

Usage

```
print(x, ...)

## S3 method for class 'factor'
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table'
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0", justify = "none", ...)

## S3 method for class 'function'
print(x, useSource = TRUE, ...)
```

Arguments

GETTING STARTED

> x = "hello world"
> print(x)
[1] "hello world"

> x <- "hello world"
> print(x)
[1] "hello world"

> x
[1] "hello world"

> q()

> help(print)

*> 2+2*3*
[1] 8

>? print
> example (print)

> ??solve

> help.start()

> every object is saved

DATA TYPES

```
> a = 49
> sqrt(a)
[1] 7
```

numeric

```
> a = "The dog ate my homework"
> sub("dog","cat",a)
[1] "The cat ate my homework“
```

character

string

```
> a = (1+1==3)
> a
[1] FALSE
```

logical

BASIC DATA TYPES - NUMERIC

```
> x = 10.5        # assign a decimal value

> x               # print the value of x

[1] 10.5
```

```
> class(x)        # print the class name of x

[1] "numeric"
```

BASIC DATA TYPES - NUMERIC

```
> k = 1

> k                 # print the value of k

[1] 1

> class(k)          # print the class name of k

[1] "numeric"
```

BASIC DATA TYPES - INTEGER

```
> y = as.integer(3)

> y                # print the value of y

[1] 3

> class(y)       # print the class name of y

[1] "integer"

> is.integer(y)  # is y an integer?

[1] TRUE
```

BASIC DATA TYPES - INTEGER

```
> as.integer(3.14)    # coerce a numeric value

[1] 3
```

```
# parse a string for decimal

> as.integer("5.27")  # coerce a decimal string

[1] 5
```

BASIC DATA TYPES - INTEGER

```
> as.integer("Joe")    # coerce an non−decimal string

[1] NA

#TRUE is 1, FALSE has value 0.

> as.integer(TRUE)     # the numeric value of TRUE

[1] 1

> as.integer(FALSE)    # the numeric value of FALSE

[1] 0
```

DATA TYPES - COMPLEX

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

```
> z = 1 + 2i     # create a complex number

> z              # print the value of z

[1] 1+2i

> class(z)       # print the class name of z

[1] "complex"
```

The following gives an error as -1 is not a complex value.

DATA TYPES - COMPLEX

```
> sqrt(−1)        # square root of −1
[1] NaN
Warning message:  In sqrt(−1) : NaNs produced

#Instead, we have to use the complex value −1 + 0i.
> sqrt(−1+0i)     # square root of −1+0i
[1] 0+1i

#An alternative is to coerce −1 into a complex value.
> sqrt(as.complex(−1))
[1] 0+1i
```

DATA TYPES - 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"
```

Standard logical operations are "&" (and), "|" (or), and "!" (negation).

DATA TYPES - LOGICAL

```
> u = TRUE; v = FALSE

> u & v          # u AND v
[1] FALSE

> u | v          # u OR v
[1] TRUE

> !u             # negation of u
[1] FALSE

> help("&")
```

DATA TYPES - CHARACTER

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"
```

DATA TYPES - CHARACTER

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

> sprintf("%s has %d dollars", "Sam", 100)  #compose
[1] "Sam has 100 dollars"

> substr("Mary has a little lamb.", start=3, stop=12)
[1] "ry has a l"

> sub("little", "big", "Mary has a little lamb.") #replace
[1] "Mary has a big lamb."

> help("sub")
```

VECTORS

- **Vector:** an ordered collection of data of the same type

> a = c(1,2,3)

*> a*2*

[1] 2 4 6

- In R, a single number is the special case of a vector with 1 element.

- Other vector types: character strings, logical

VECTORS

```
> c(2, 3, 5)       ## containing three numeric values 2, 3 and 5
 [1] 2 3 5

> c(TRUE, FALSE, TRUE, FALSE, FALSE)  ## logical vector
 [1]  TRUE FALSE  TRUE FALSE FALSE

> c("aa", "bb", "cc", "dd", "ee")  ## character vector
 [1] "aa" "bb" "cc" "dd" "ee”

> length(c("aa", "bb", "cc", "dd", "ee"))  ## to find length of vector
 [1] 5
```

VECTORS - COMBINING

```
> n = c(2, 3, 5)

> s = c("aa", "bb", "cc", "dd", "ee")

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

Value Coarcion – above example is of Numeric value to string conversion

VECTORS

- **Vector Arithmetics:** Happens member by member (similar to Matlab)

```
> a = c(1, 3, 5, 7)

> b = c(1, 2, 4, 8)

> 5 * a                ## Scalar multiplication
 [1]  5 15 25 35

> a + b
 [1]  2  5  9 15

> a - b
 [1]  0  1  1 -1
```

VECTORS

```
> a * b

 [1]  1  6 20 56

> a / b

 [1] 1.000 1.500 1.250 0.875
```

- **Recycling Rule** states that if two vectors are of unequal lengths, the shorter one will be recycled in order to match the longer vector.

```
> u = c(10, 20, 30)

> v = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

> u + v

 [1] 11 22 33 14 25 36 17 28 39
```

VECTORS

- Accessing and Retrieving the elements of the vector: Done by declaring an index inside a *single square bracket* "[]“. Vector index is 1 based and not 0 based.

```
> s = c("aa", "bb", "cc", "dd", "ee")

> s[3]
 [1] "cc“
```

the result of the square bracket operator is another vector, and s[3] is a vector **slice** containing a single member "cc".

VECTORS

- **Negative Index:** If the index is negative, it would strip the member whose position has the same absolute value as the negative index.

> s = c("aa", "bb", "cc", "dd", "ee")

> s[-3]
[1] "aa" "bb" "dd" "ee“

- Out of Range Index: If the index is out-of-range, it is reported by symbol NA.

> s[10]
[1] NA

VECTORS

- **Numeric Index Vector:** Consists of member positions of the original vector to be retrieved.

> s = c("aa", "bb", "cc", "dd", "ee")

```
> s[c(2, 3)]
 [1] "bb" "cc“
```

- **Duplicate Vector:** Retrieves a member twice in one operation.

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

VECTORS

- **Out-of-order Vector:** Order of the index does not matter.

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

- **Range Index:** To produce a vector slice between two indexes.

> s[2:4]
[1] "bb" "cc" "dd“

VECTORS

- **Logical Index Vector:** Has the same length as the original vector. Its members are TRUE if the corresponding members in the original vector are to be included in the slice, and FALSE if otherwise.

. > L = c(FALSE, TRUE, FALSE, TRUE, FALSE)
  > s[L]
  [1] "bb" "dd"

. > s[c(FALSE, TRUE, FALSE, TRUE, FALSE)] ## Alternative to above two lines
  [1] "bb" "dd"

VECTORS

- **Named Vectors:** Provide names to members.

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

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

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

MATRICES

- **Matrix**: a rectangular table of data of the same type

- **Example**: Sales of 10000 products across 4 Stores will be a matrix with 4 rows and 10000 columns.

```
> mat = matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3, byrow=TRUE)
```

MATRICES

```
> 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 matrix by rows

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

MATRICES

```
> A[ ,c(1,3)]  # the 1st and 3rd columns
> dimnames(A) = list(
+   c("row1", "row2"),          # row names
+   c("col1", "col2", "col3")) # column names
> A["row2", "col3"] # element at 2nd row, 3rd column
```

MATRICES

Transposing matrix

```
> B = matrix( c(2, 4, 3, 1, 5, 7), nrow=3, ncol=2)

>B              # B has 3 rows and 2 columns
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7

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

MATRICES

Destructing A Matrix

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

MATRICES

```
A = matrix(c(1,2,3,4,5,6,9,1,36,1), 5, 2)

B = matrix(c("X","Y","Z","Z","A"), 5, 1)

cbind(A,B)
```

	V1	V2
1	1	6
2	2	9
3	3	1
4	4	36
5	5	1

	V1
1	X
2	Y
3	Z
4	Z
5	A

=>

	V1	V2	V3
1	1	6	X
2	2	9	Y
3	3	1	Z
4	4	36	Z
5	5	1	A

MATRICES

A = matrix(c(1,2,3,4,5,6,9,1,36,1), 5, 2)
C = matrix(c(1,2),1,2)
rbind(A,B)

	V1	V2
1	1	6
2	2	9
3	3	1
4	4	36
5	5	1

	V1	V2
1	1	2

=>

	V1	V2
1	1	6
2	2	9
3	3	1
4	4	36
5	5	1
6	1	2

LISTS

List: an ordered collection of data of arbitrary types.

```
> doe = list(name="john",age=28,married=F)

> doe$name

[1] "john“

> doe$age

[1] 28
```

Typically, vector elements are accessed by their index (an integer), list elements by their name (a character string). But both types support both access methods.

LISTS

List Slicing and Member Reference:

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc", "dd", "ee")
> b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
> x = list(n, s, b, 3)   # x contains copies of n, s, b
```

```
> x[2]       ## List slicing using member index

> x[c(2, 4)]    ## List slicing using index vector

> x[[2]]          ## Referencing member directly

> x[[2]][1] = "ta"    ## Changing the content of the member directly
```

LISTS

Named List members:

```
> v = list(bob=c(2, 3, 5), john=c("aa", "bb"))
> v["bob"]          ## List Slicing


> v[["bob"]]

> v$bob             ## Direct name reference using $

> attach(v)         ## To access variables directly by name

> bob

> detach(v) ## Will leave the changed variables locally.
```

DATA FRAMES

Data Frame:

- Rectangular table with rows and columns

- Like a spreadsheet

- Data within each column has the same type

- (e.g. number, text, logical),

- but different columns may have different types.

DATA FRAMES

> a

	Price	Floor	Area	Rooms	Age	Cent.heat
52.00	111.0	830	5	6.2	no	
54.75	128.0	710	5	7.5	no	
3	57.50	101.0	1000	5	4.2	no

> mtcars

> str(mtcars)

DATA FRAMES

```
> data.frame(k1 = c(NA,NA,3,4,5),
    k2 = c(1,NA,NA,4,5),
    data = 1:5)
```

	k1	k2	data
1	NA	1	1
2	NA	NA	2
3	3	NA	3
4	4	4	4
5	5	5	5

DATA FRAMES

```
> mtcars["Mazda RX4", "cyl"] # prints value of cyl column

> nrow(mtcars)    # number of data rows

> ncol(mtcars)    # number of columns

> head(mtcars)    # for peeking in the dataframe top 6 rows

> mtcars[[9]]     # Accessing the specific column of df by column no

> mtcars$am   # Accessing the specific column of df by col name

> mtcars[,"am"]   # Accessing the specific column of df by col name
```

DATA FRAMES

```
> mtcars[24,]                    # Accessing the specific row of df by row no

> mtcars[c(3, 24),]              # Accessing the specific rows by row nos

>mtcars[c(grep("Toy",rownames(mtcars))),]
```

FACTORS

- Used for categorical values which can only have limited values like gender can be male, female, etc
- A factor is a variable that can only take such a limited number of values, which are called levels.

FACTORS

```
# Code to build factor_survey_vector
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)

# Specify the levels of factor_survey_vector
levels(factor_survey_vector) <- c('Female', 'Male')

factor_survey_vector
```

FACTORS - Ordering

```
# Create speed_vector
speed_vector <- c("fast", "slow", "slow", "fast", "insane")

# Convert speed_vector to ordered factor vector
factor_speed_vector <- factor(speed_vector, ordered=TRUE,
    levels=c("slow", "fast", "insane"))

# Print factor_speed_vector
factor_speed_vector
summary(factor_speed_vector)
```

FACTORS - Comparing ordered

```
# Create factor_speed_vector
speed_vector <- c("fast", "slow", "slow", "fast", "insane")
factor_speed_vector <- factor(speed_vector, ordered = TRUE,
levels = c("slow", "fast", "insane"))

# Factor value for second
da2 <- factor_speed_vector[2]

# Factor value for fifth
da5 <- factor_speed_vector[5]

# Is da2 faster than da5?
da2 > da5
```

STRINGS

- Functions for searching and modifying strings are based on regular expressions, which are like simple programs for representing strings.

- **String function Examples:**

>s <- "Knowbigdata.com Data Analytics with R Tutorial"

>substr(s,0,7)

[1] "Knowbig"

Get string length:

>nchar(s)

[1] 46

STRING FUNCTION EXAMPLES:

To uppercase:

```
>x <- toupper(s)
>x
[1] "KNOWBIGDATA DATA ANALYTICS WITH R TUTORIAL"
```

To lowercase:

```
>x <- tolower(s)
>x
[1] "knowbigdata data analytics with r tutorial"
```

STRING FUNCTION EXAMPLES:

Split the string at letter "o":

```
>x <- strsplit(s,"o")
[[1]]
[1] "Kn"                              "wbigdata.c"
[3] "m Data Analytics with R Tut" "rial"
```

Concatenate two strings:

```
>x <- paste(s," -- String Functions",sep="")
>x
[1] "Knowbigdata Data Analytics with R Tutorial -- String
   Functions"
```

STRING FUNCTION EXAMPLES:

Substring replacement:

```
>x <- sub("Tutorial","Examples",s)
>x
[1] "Knowbigdata Data Analytics with R Examples"
```

Examples:

- "a" matches the letter a
- "TATAA" matches the string TATAA
- "ACGT.C" matches ACGT followed by any single character, followed by C
- "a*b*" matches zero or more a's followed by zero or more b's

STRINGS

- "[A-Za-z]+" matches one or more characters between A and Z or between a and z (in English, any letter).
- "[[:alpha:]]+" matches one or more letters, defined according to the current locale
- "\\.csv$" matches the string .csv at the end of a string
- "([[:alpha:]]+)[[:space:][:punct:]]+\\1[[:space:][:punct:]]+“ matches any repeated word.
- grep
- gsub("([0-9]+)\\,([0-9])", "\\1\\2", lines)

PLAYING WITH R

- Basic manipulations to understand the nature of any language:

  > x <-seq(1,10)

  > length(x); str(x); head(x)

  > rep(1:3,6)

  > mean(x)

PLAYING WITH R

```
> var(x)
> summary(x)
> x<-c(5,7,9)  ;  y<-c(6,3,4) ;  z<-cbind(x,y) ; z ; rbind(x,y)
```

READING DATA

- Text Files
- Web Pages
- Datasets from other statistical softwares (e.g. SAS, STATA etc.)
- Databases – RDBMS / NoSQL.

READING DATA (SPACE DEL. FILES)

If the dataset is space separated like below:

case id gender deg yrdeg field startyr year rank admin
1 1 F Other 92 Other 95 95 Assist 0
2 2 M Other 91 Other 94 94 Assist 0
3 2 M Other 91 Other 94 95 Assist 0
4 4 M PhD 96 Other 95 95 Assist 0

```
> salary <- read.table("salary.txt", header=TRUE)
>View(salary)
```

to read the data from the file salary.txt into the data frame *salary*.

POINT TO NOTE:

- Spaces in commands don't matter (except for readability), but Capitalisation Does Matter.

- Unlike many systems, R does not distinguish between commands that do something and commands that compute a value.

Everything is a function: ie returns a value.

POINT TO NOTE:

- Arguments to functions can be named (header=TRUE) or unnamed ("salary.txt")
- A whole data set (called a data frame is stored in a variable (salary), so more than one dataset can be available at the same time.

FILE PATH & WORKING DIRECTORY

The working directory can be set in the following way:

- Session -> Set Working Directory
- Using setwd() e.g. setwd("C:\myfolder")
- To find out existing working directory: use getwd()

READING DATA (CSV FILES)

If the dataset is comma separated like below:

Ozone,Solar.R,Wind,Temp,Month,Day
41,190,7.4,67,5,1
36,118,8,72,5,2
12,149,12.6,74,5,3
18,313,11.5,62,5,4
NA,NA,14.3,56,5,5

>ozone <- read.table("ozone.csv", header=TRUE, sep=",")
OR
>ozone <- read.csv("ozone.csv", header=TRUE)

POINT TO NOTE:

- Functions can have optional arguments (sep wasn't used the first time).

- Check help(read.table) for a complete description

- read.csv2 handles comma in numbers

- NA is the code for missing data. Think of it as "Don't Know". R handles it sensibly in computations:

READING DATA (WITHOUT HEADER)

- Sometimes variable names are not included:

1 0.2 115 90 1 3 68 42 yes
2 0.7 193 90 3 1 61 48 yes
3 0.2 58 90 1 3 63 40 yes

```
> psa <- read.table("psa.txt", col.names=c("ptid","nadirpsa",
"pretxpsa",  "ps", "bss", "grade", "age", "obstime", "inrem"))
```

OR

```
> psa <- read.table("psa.txt")
> names(psa) <- c("ptid","nadirpsa","pretxpsa", "ps", "bss", "grade",
"age", "obstime", "inrem"))
```

POINT TO NOTE:

- Assigning a single vector (or anything else) to a variable uses the same syntax as assigning a whole data frame.
- c() is a function that makes a single vector from its arguments.
- names() is a function that accesses the variable names of a data frame
- Some functions (such as names) can be used on the LHS of an assignment.

READING DATA (FROM WEB)

Files for read.table can live on the web

```
> readWeb <-
read.table("http://faculty.washington.edu/tlumley/data/FLvote.dat",
header=TRUE)
```

READING DATA (DIFFERENT DEL.)

Files for read.table could be separated by something different than space or comma. It can be read by using:

> readAny <- read.delim(“file.txt“, header=TRUE, sep = "\t",)

POINT TO NOTE:

- The default separator is tab, it can be changed to anything else.
- read.table is not the right tool for reading large matrices, especially those with many columns: it is designed to read data frames which may have columns of very different classes.
- Scan is the alternative

READING DATA (OTHER STAT. SOFT.)

For reading outputs or formats from other statistical software vendors:

```
> library(foreign)
> stata <- read.dta("salary.dta")
> spss <- read.spss("salary.sav", to.data.frame=TRUE)
> sasxport <- read.xport("salary.xpt")
> epiinfo <- read.epiinfo("salary.rec")
```

POINT TO NOTE:

- library() function lists packages, shows help, or loads packages from the package library.
- install.packages() allows the users to install any of the packages from the CRAN, along with their dependencies.
- foreign is a standard package from the distribution which handles import and export of data.

WRITING DATA OUT

Standard commands similar to those for inputs exist for writing output:

```
> write.table(x, file="x.txt", sep="\t")
> write.csv(x, file = "foo.csv")
> write.csv(x, file = "foo.csv", row.names = FALSE)
```

POINT TO NOTE:

- row.names=F sometimes is required if you don't want the rows to have labels.

PROGRAMMING CONSTRUCTS

- Conditional Controls
- Loops
- Vector Operators
- Functions

CONDITIONAL CONTROLS

- Conditional Controls

if statements : The language has available a conditional construction of the form

if (expr 1) {expr 2 }else {expr 3}

where expr 1 must evaluate to a logical value and the result of the entire expression is then evident.

A vectorized version of the if/else construct, the *ifelse* function. This has the form:

ifelse(condition, a, b)

switch statement is used for a reason similar to that of if statement, form being:

switch(EXPR=var, expr1 = action1, …)

POINT TO NOTE:

- *if statement* is required when you want to control the flow of programs conditionally, e.g.

```
> x <- -5
   if(x > 0) {
     print("Non-negative number")
   } else {
      print("Negative number")
   }
```

- *ifelse statement* is used for assigning when you want to assign values to the variables conditionally, e.g.

```
> y = ifelse(x >= 0, sqrt(x), NA)
```

POINT TO NOTE:

- Example of switch statement:

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- c(1,2,3,4,5,6,7,8,9)
centre(x, "mean")
centre(x, "median")
centre(x, "trimmed")
```

LOOPS

- Below are the repetitive execution constructs:
    - *for(var in seq) expr*
    - *while(cond) expr*
    - *repeat expr*
    - *break*
    - *next*

POINT TO NOTE:

- *for* loop has *seq* as a vector, which defines the loop variable (*var*) range of variation, e.g.

```
> for(i in 1:5) print(1:i)
> for(n in c(1,2,3,4,5)) {
    if (n == 3) next;
    if (n > 4) break;
    print(n);
}
```

POINT TO NOTE:

• *for* loops should be avoided as the vector operations are available and most generally are much faster compared to the loop.

• Example of while and repeat loops (typically used when the number of iterations are not known in advance):

```
> i <- 0; term <- 1; sum <- 1; x <- 5
> while( term > 0.0001) {
    i <- i + 1; term <- x^i / factorial(i)
    sum <- sum + term
}
```

Loops - Repeat

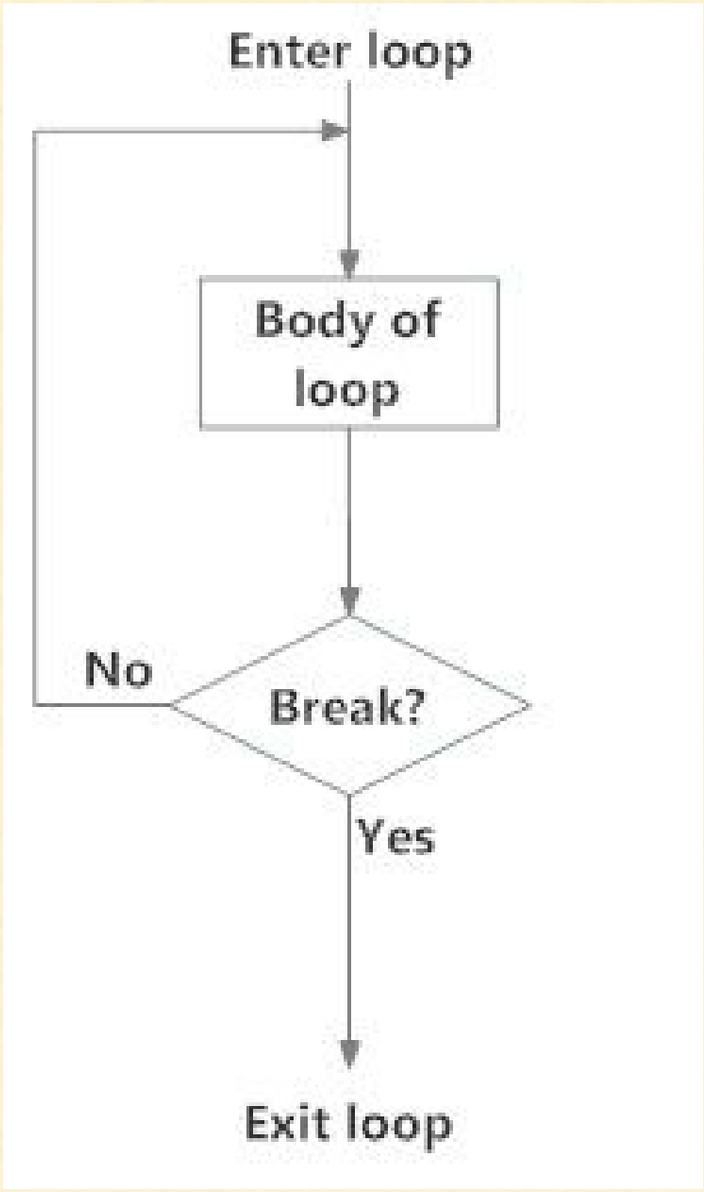
Infinite Loop

Syntax:

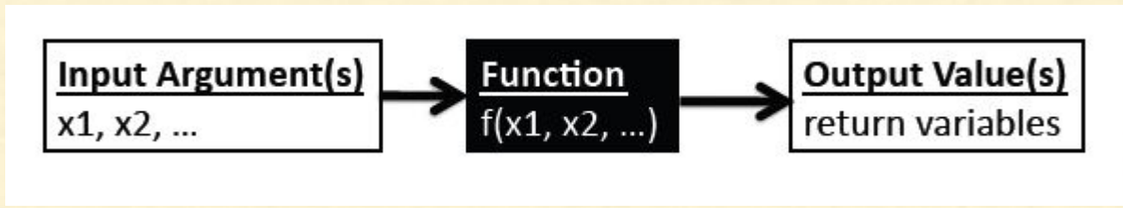
```
repeat {
   statement
}
```

Example:

```
> repeat {
     i <- i + 1; term <- x^i / factorial(i)
     if (term <= .0001) break
     sum <- sum + term
     }
```



FUNCTIONS



Question: What is a function in R?

Answer: Everything!

- Functions promote reuse, make program modular and easy to maintain, and easy to debug.

USING FUNCTIONS

Generic function call
> y <- function(x1, x2, ...)

where x1, x2 etc are called arguments and are used as function inputs

or *BETTER* yet

> y <- function(var1=x1, var2=x2,…)

where var1, var2 are names of arguments and x1, x2 are the corresponding values

- Better to have names specified for readability and debugging, if not given, the order of input decides the allocation

CREATING FUNCTIONS

- Functions are written as:

```
> myFunc <- function(x1, x2, ...) {

    # code
    return(value) # results from function

}
```

- Example:

```
> myMean <- function(x) {
return(sum(x)/length(x))
}
```

CREATING FUNCTIONS

- Although R only returns single object, lists could be used to pass more than one variables at a time.
- Warning: We can overwrite existing R functions.

DATA WRANGLING

- Subsetting
- Sorting
- Merging
- Reshaping
- Aggregation functions
- String Operations

SUBSETTING

- Individual elements of a vector, matrix, array or data frame are accessed with "[ ]" by specifying their index, or their name

Demonstrations:

```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
```

```
> mtcars[3, 2]
[1] 4
```

Isolating the element using row and column indices

```
> mtcars["Valiant", "mpg"]
[1] 18.1
```

Isolating the element using row number and column name

```
> mtcars["Valiant", ]
        mpg cyl disp  hp drat   wt  qsec vs am gear carb
Valiant 18.1   6  225 105 2.76 3.46 20.22  1  0    3    1
```

Isolating the columns using row number

SUBSETTING - DEMONSTRATIONS

```
> mtcars[c(1,3),]
               mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.62 16.46  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.32 18.61  1  1    4    1
```

subset rows by a vector of indices

SUBSETTING - DEMONSTRATIONS

> mtcars$mpg

subset a column

> mtcars[, c(”mpg”, “wt”)]

Subsetting multiple columns by column names

SUBSETTING - DEMONSTRATIONS

```
>  mtcars$cyl==6
 [1]  TRUE  TRUE FALSE  TRUE FALSE ....
```

comparison resulting in logical vector c

```
> Yesornot = mtcars[mtcars$cyl==6,]
```

comparison results stored as a logical vector

```
> mtcars[mtcars$cyl==6,]
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

MOST IMP - subset the selected rows

SORTING

- One of the most basic and important transition of the data.
- For vectors, *sort()* does the trick.
- For data frames etc, ordering can be done as applicable using *order* argument, default being ascending

SORTING - Sort()

Syntax

sort(x, decreasing = FALSE, na.last = NA, …)

- decreasing - if true, sorts in ascending
- na.last - if true, displays NAs in the last rather than beginning, if NA, removed

SORTING - Sort()

1. attach(mtcars)

2. cars = rownames(mtcars)

3. sort(cars)

```
 [1] "AMC Javelin"          "Cadillac Fleetwood"  "Camaro Z28"
 [8] "Ferrari Dino"         "Fiat 128"            "Fiat X1-9"
[15] "Lincoln Continental"  "Lotus Europa"        "Maserati Bora"
[22] "Merc 280"             "Merc 280C"           "Merc 450SE"
[29] "Toyota Corolla"       "Toyota Corona"       "Valiant"
```

SORTING - order()

A permutation which rearranges its first argument into ascending or descending order

Syntax: *sort(x, decreasing = FALSE, na.last = NA, …)*

```
> a = c("rat", "bat", "cat")
> a
[1] "rat" "bat" "cat"
      1         2     3
> sort(a)
[1] "bat" "cat" "rat"
      2         3     1
> order(a)
[1] 2 3 1
```

SORTING - order mtcars

```
> attach(mtcars)
> mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 2 …. ….

>order(mpg)
 [1] 15 16 24  7 17 31 14 23 22 29 12 13 11  6  5 10 25 3 … …

> mtcars[order(mpg),]
                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood   10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental  10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Camaro Z28           13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
… ….
```

SORTING - order mtcars by multiple columns

```
# sort by mpg and cyl
mtcars[order(mpg, cyl),]
```

	row.names	mpg	cyl	disp	hp
1	Lincoln Continental	10.4	8	460.0	215
2	Cadillac Fleetwood	10.4	8	472.0	205
3	Camaro Z28	13.3	8	350.0	245
4	Duster 360	14.3	8	360.0	245
5	Chrysler Imperial	14.7	8	440.0	230
6	Maserati Bora	15.0	8	301.0	335
7	Merc 450SLC	15.2	8	275.8	180
8	AMC Javelin	15.2	8	304.0	150
9	Dodge Challenger	15.5	8	318.0	150
10	Ford Pantera L	15.8	8	351.0	264

	row.names	mpg	cyl	disp
1	Cadillac Fleetwood	10.4	8	472.0
2	Lincoln Continental	10.4	8	460.0
3	Camaro Z28	13.3	8	350.0
4	Duster 360	14.3	8	360.0
5	Chrysler Imperial	14.7	8	440.0
6	Maserati Bora	15.0	8	301.0
7	AMC Javelin	15.2	8	304.0
8	Merc 450SLC	15.2	8	275.8
9	Dodge Challenger	15.5	8	318.0

```
#sort by mpg (ascending) and cyl (descending)
mtcars[order(mpg, -cyl),]
```

STRING MANIPULATIONS

- *grep(regexp, vector)* finds all the strings in the vector that contain a substring matching the regular expression .

```
txt <- c("arm","foot","lefroo", "bafoobar")
if(length(i <- grep("foo",txt)))
   cat("'foo' appears at least once in\n\t",txt,"\n")
i # index of the containing terms
txt[i]
```

STRING MANIPULATIONS

- *regexpr(regexp, vector)* returns the position of the first match within each string, *gregexpr* is the same except that it returns all matches.

```
txt <- c("The", "licenses", "for", "most", "software", "are",
 "designed", "to", "take", "away", "your", "freedom",
 "to", "share", "and", "change", "it.",
  "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
  "is", "intended", "to", "guarantee", "your", "freedom", "to",
  "share", "and", "change", "free", "software", "--",
  "to", "make", "sure", "the", "software", "is",
  "free", "for", "all", "its", "users")

regexpr("en", txt)

gregexpr("e", txt)
```

STRING MANIPULATIONS

- *strsplit()* splits a string at each match to a regular expression

```
x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x, "e")
```

Please make a note - these are just basics and a lot can be done depending on the requirements.

Playing more with R and Summary

- *Let's load the data and understand the summary, mean, median etc*

cars <- read.csv("cars.csv")

View(cars)

head(cars)

tail(cars)

str(cars)

dim(cars)

str(cars)

names(cars)

ncol(cars)

nrow(cars)

class(cars)

Descriptive stats

```
attach(cars)
mean(MPG)
median(MPG)
mode(MPG)
var(MPG)
sd(MPG)
range(MPG)
min(MPG)
max(MPG)
quantile(MPG, seq(0,1,0.05))
```

Counting & Aggregation

```
# Count of cars by Origin
library(plyr) # install.packages("plyr")
count(cars, 'Origin')

# Means, max, min by Origin
aggregate(MPG, list(Origin), mean)
aggregate(MPG, list(Origin), max)
aggregate(MPG, list(Origin), min)
```

Find rows with max, min etc

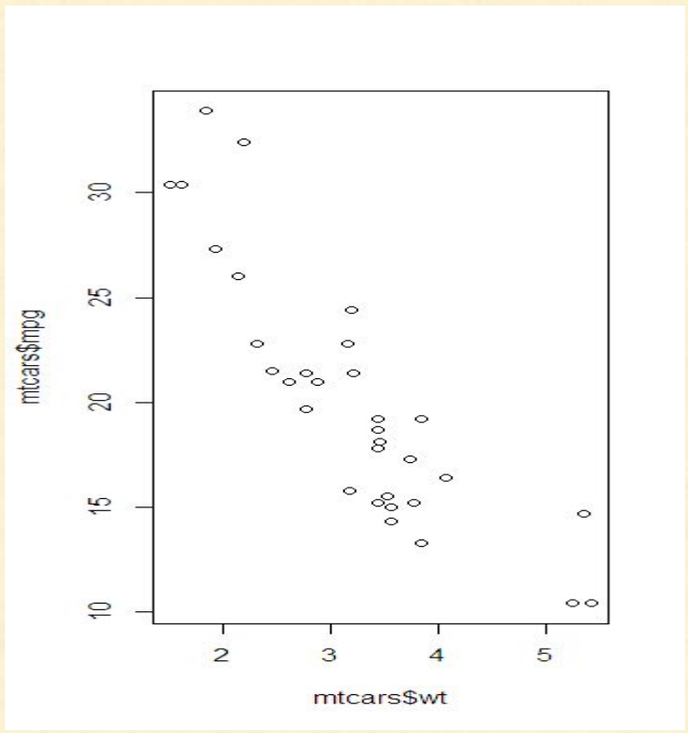
```
newcars4<- cars[which.max(cars$MPG),]
newcars5<- cars[cars$MPG == max(cars$MPG),]
newcars6<- cars[which.min(cars$MPG),]
newcars7<- cars[cars$MPG == min(cars$MPG),]
```

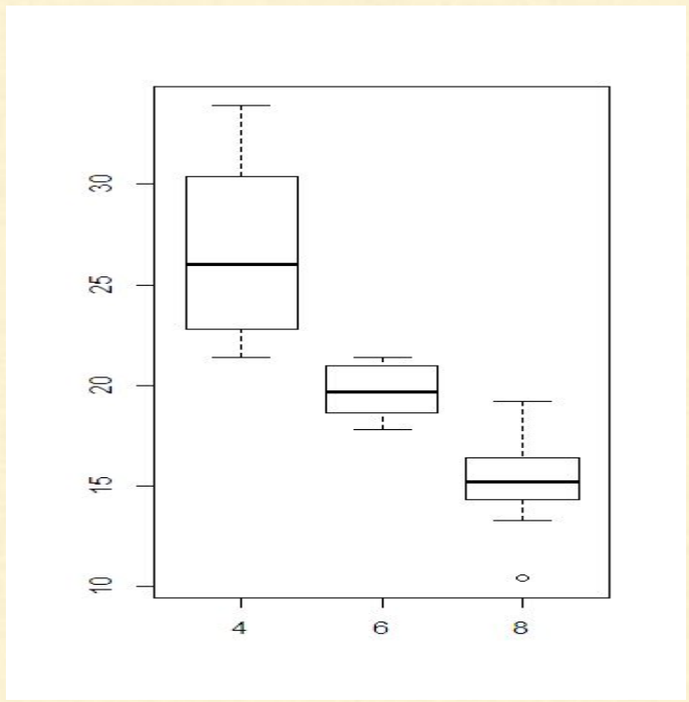
GRAPHICS - BASICS

- R can produce graphics in many formats, including:
- On screen
- PDF files for LATEX or emailing to people
- PNG or JPEG bitmap formats for web pages (or on non-Windows platforms to produce graphics for MS Office). PNG is also useful for graphs of large data sets.

GRAPHICS – PLOT COMMAND

```
## scatterplot
plot(mtcars$wt, mtcars$mpg)
## boxplot
boxplot(mpg~cyl,data=mtcars, main="Car Milage Data",
    xlab="Number of Cylinders", ylab="Miles Per Gallon")
```

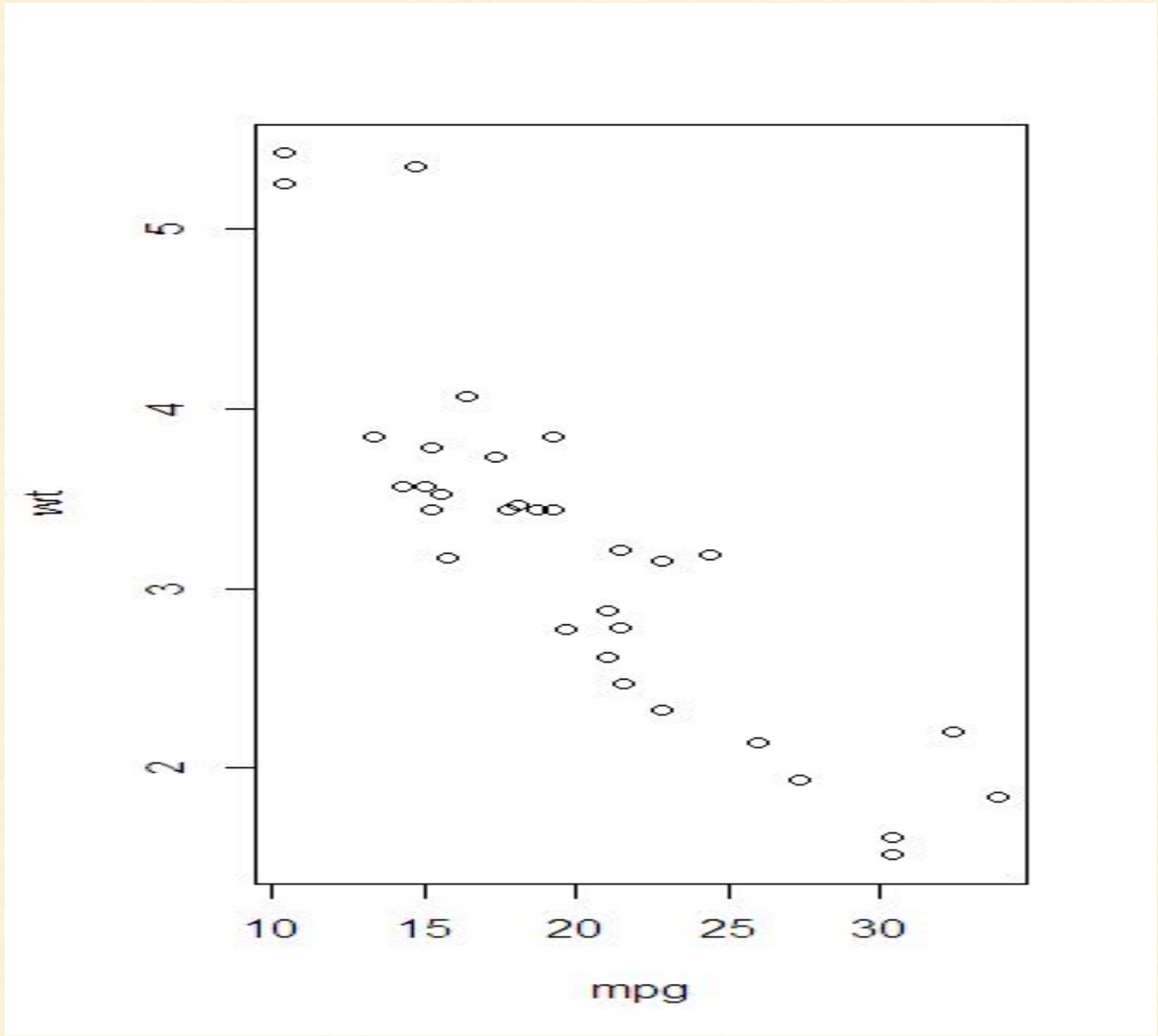




GRAPHICS – PLOT COMMAND

- It also supports the formula style of writing commands that's going to be used in Analytics as well:

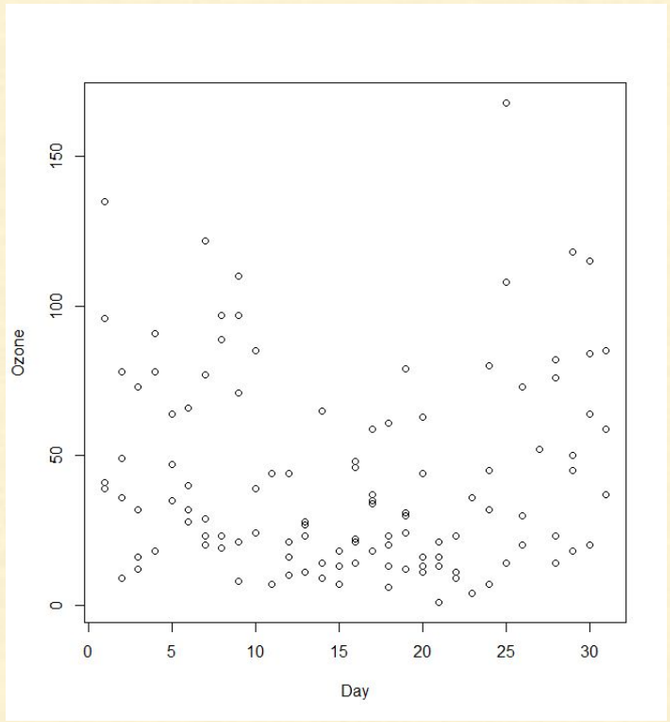
plot(wt~mpg, data=mtcars)
Y-axis X-axis Dataset

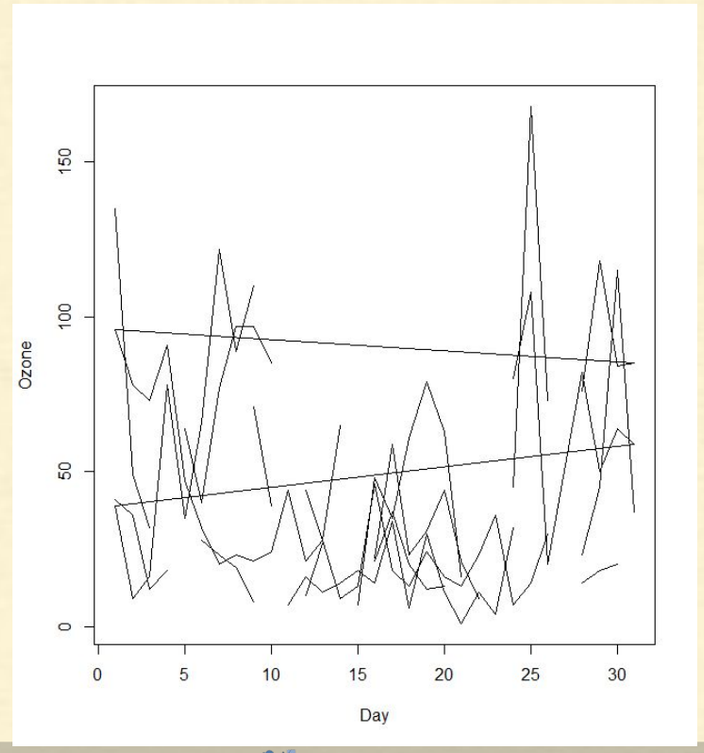


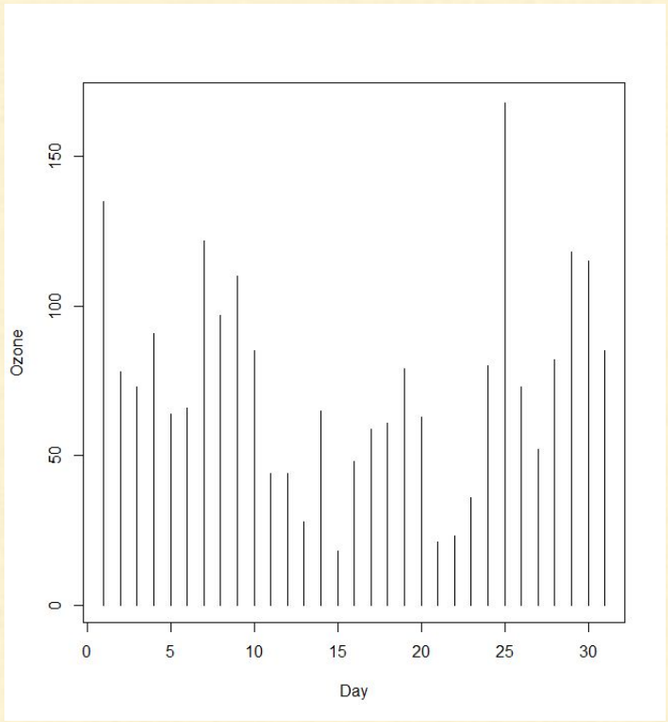
GRAPHICS – PLOT ARGUMENTS

- *type("x")* – Used to specify type of plot, "l" for line, "h" for bar, point by default.

```
library(datasets)
## scatterplot
plot(Ozone~Day, data=airquality)
## lineplot
plot(Ozone~Day, data=airquality,type="l")
## barplot
plot(Ozone~Day, data=airquality,type="h")
```



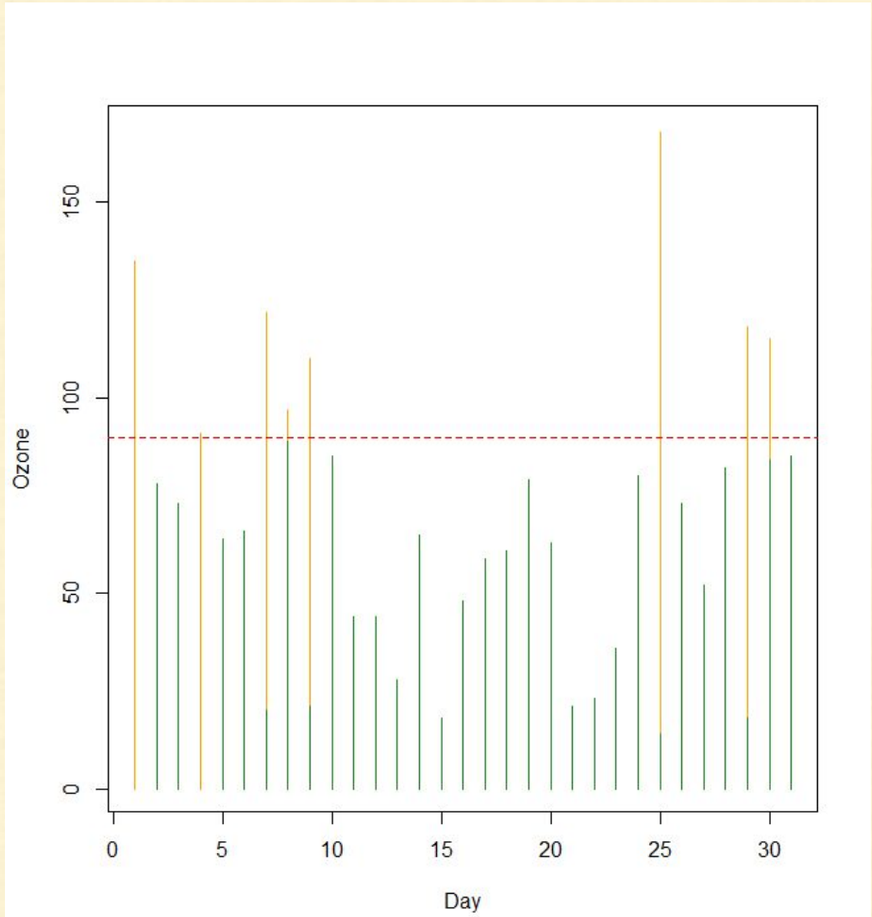




GRAPHICS – PLOT ARGUMENTS

- *abline(h,lty,col)* – adds a single line to the plot, inputs the height, line type (1-solid, 2-dashed, 3-dotted), colour (many available)

```
bad<-ifelse(airquality$Ozone>=90, "orange","forestgreen")
plot(Ozone~Day, data=airquality,type="h",col=bad)
abline(h=90,lty=2,col="red")
```



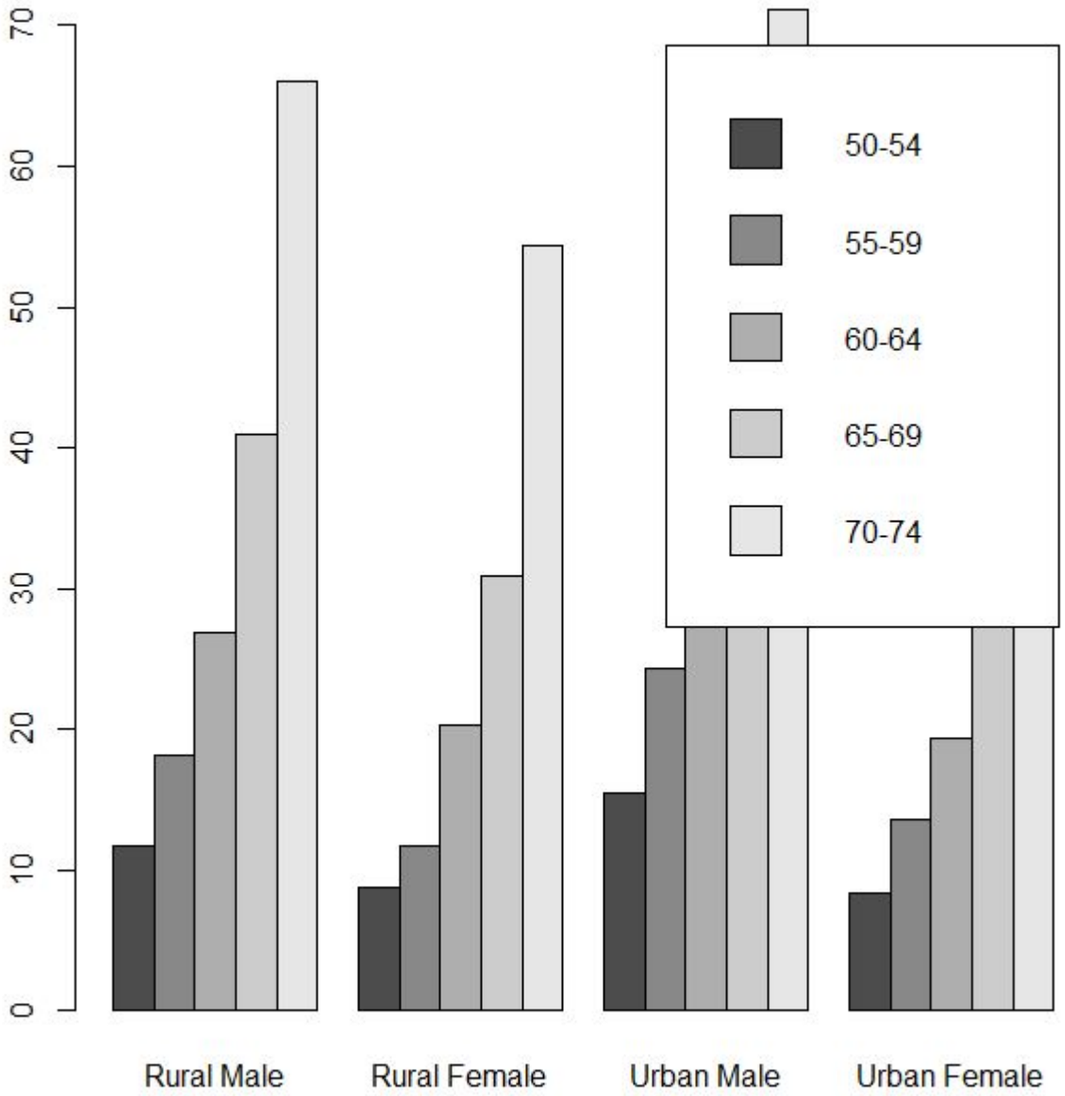
BARPLOTS & HISTOGRAMS

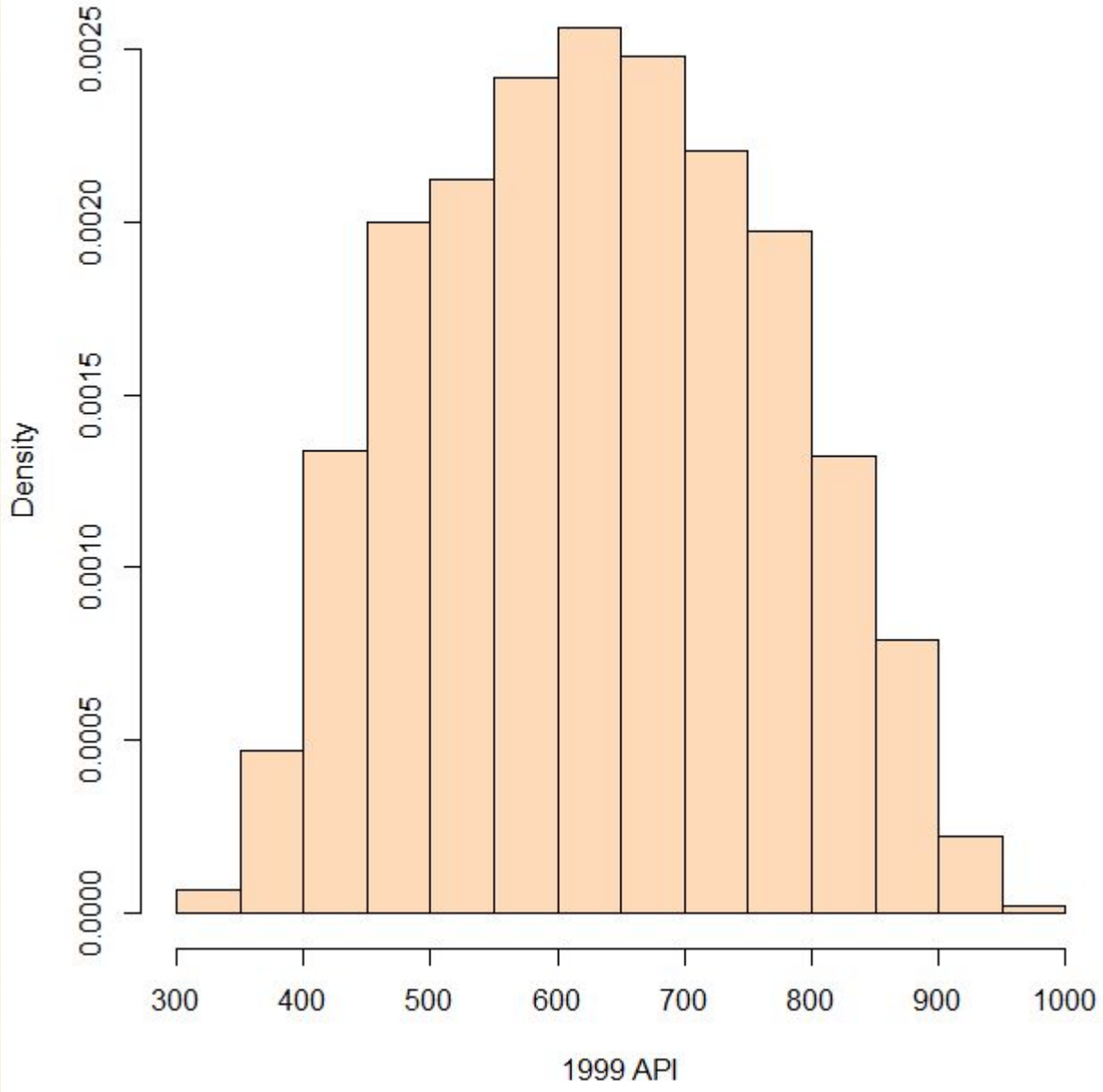
```
barplot(VADeaths,beside=TRUE,legend=TRUE)
attach(apipop)                  ## to use the variable without spacifying dataset name
hist(api99, col="peachpuff", xlab="1999 API", main="",prob=TRUE)
## notice the direct use of the variable name instead of apipop$api99.
```

- *barplot(height, names.arg, beside, legend)* – plot the bars using the variable mentioned in height argument and name them accordingly, while plotting and creating legend.

- *hist(variable, probability, …)* – creates histogram for the given variables either using frequency or probability as the plotting quantity.

BARPLOTS & HISTOGRAMS





ggplot

```
library(ggplot2) # may need to install this package using install.packages()
library(gridExtra)
attach(mtcars)
p <- ggplot(mtcars, aes(mpg, wt))

p + geom_boxplot(aes(fill = mpg))

p1 <- ggplot(cars, aes(x=wt, y= mpg))
p1+geom_point(aes(color=factor(wt))) + scale_color_manual(values = c("Green", "Purple",
"Orange"))
```