

# **Introduction to R Software**

## **Introduction**

::::

## **Help, Demonstration, Examples, Packages and Libraries**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Starting with R

To start R, double click on the icon



Then we get the following Gui (Graphic user interface) window screen

A screenshot of the R GUI window titled "RGui (64-bit)". The window has a menu bar with File, Edit, View, Misc, Packages, Windows, and Help. Below the menu is a toolbar with various icons. The main area is titled "R Console". It displays the standard R startup message, including the version (3.2.3), copyright information, and licensing details. It also mentions natural language support, collaborative project status, and help resources like demos and online help. A red cursor is visible at the bottom left of the console area.

```
R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

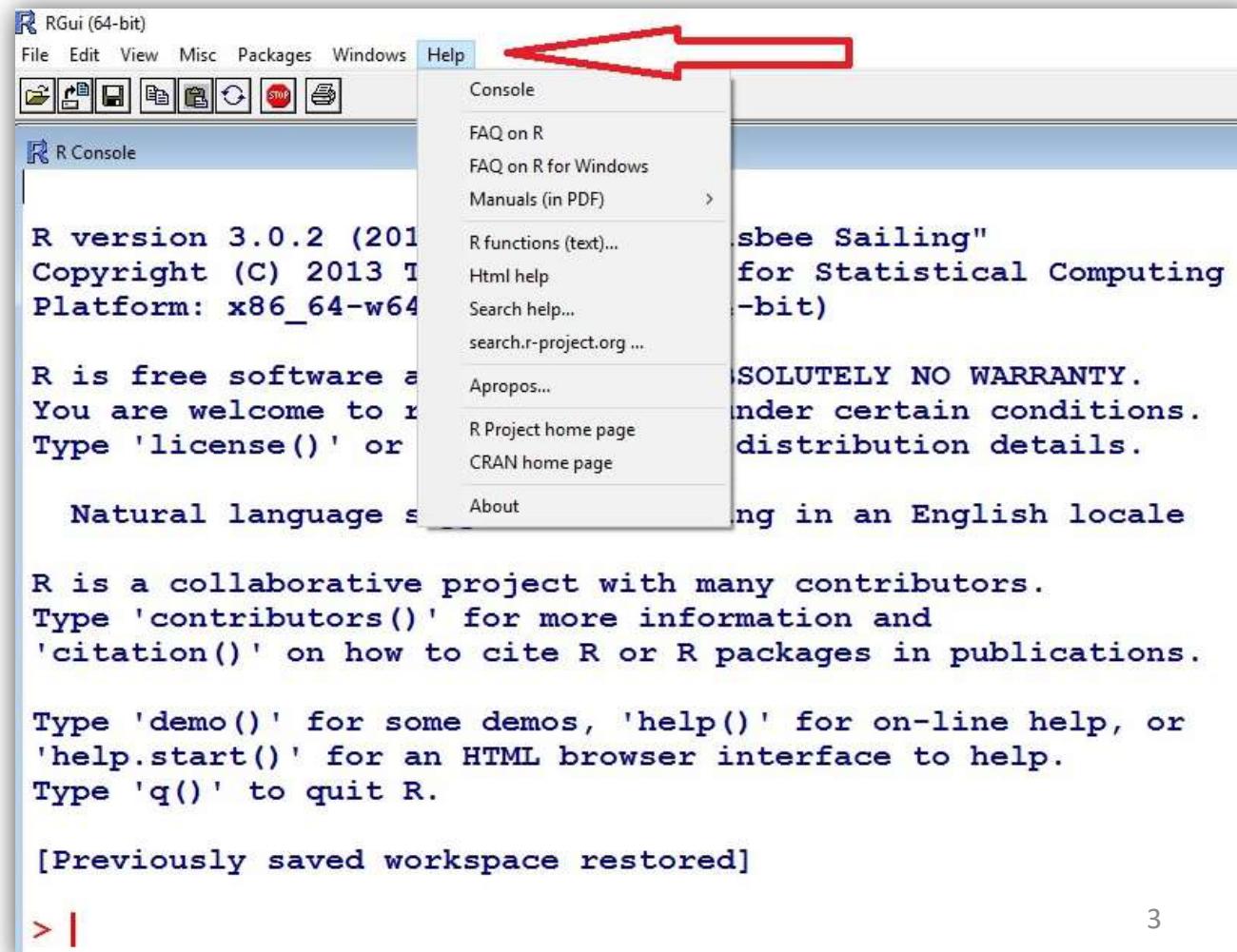
[Previously saved workspace restored]

> |
```

# Getting Help in R

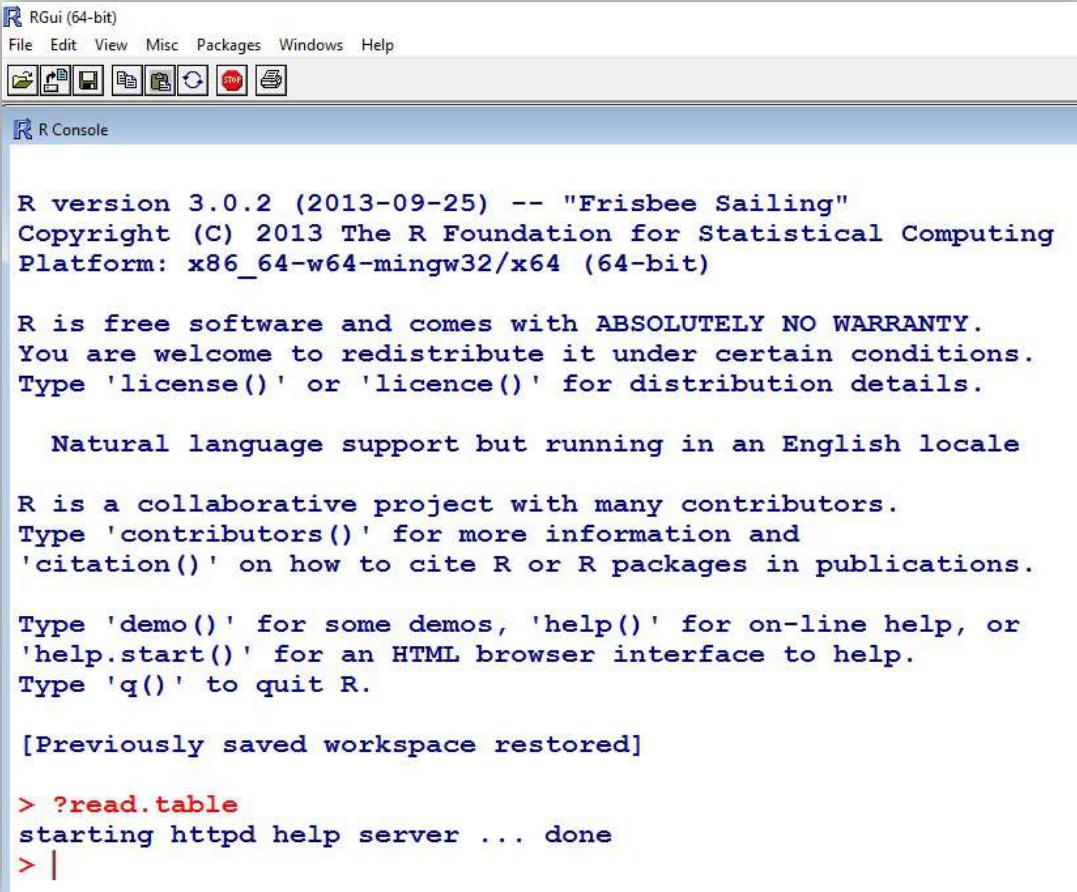
This can be done in one of the following ways:

- 1) Start R software and click the help button in the toolbar of the R Gui (Graphic user interface) window.



# Getting Help in R

2. Search for help in Google [www.google.com](http://www.google.com)
3. If you need help with a function, then type question mark followed by the name of the function. For example, **?read.table** to get help for function **read.table**.



R Gui (64-bit)

File Edit View Misc Packages Windows Help

R Console

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

[Previously saved workspace restored]

> ?read.table
starting httpd help server ... done
> |
```

## read.table {utils}

### Data Input

#### Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

#### Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"\"",  
          dec = ".", row.names, col.names,  
          as.is = !stringsAsFactors,  
          na.strings = "NA", colClasses = NA, nrows = -1,  
          skip = 0, check.names = TRUE, fill = !blank.lines.skip,  
          strip.white = FALSE, blank.lines.skip = TRUE,  
          comment.char = "#",  
          allowEscapes = FALSE, flush = FALSE,  
          stringsAsFactors = default.stringsAsFactors(),  
          fileEncoding = "", encoding = "unknown", text)
```

```
read.csv(file, header = TRUE, sep = ",", quote = "\"\"",  
        dec = ".", fill = TRUE, comment.char = "", ...)
```

```
read.csv2(file, header = TRUE, sep = ";", quote = "\"\"",  
         dec = ",", fill = TRUE, comment.char = "", ...)
```

```
read.delim(file, header = TRUE, sep = "\t", quote = "\"\"",  
          dec = ".", fill = TRUE, comment.char = "", ...)
```

```
read.delim2(file, header = TRUE, sep = "\t", quote = "\"\"",  
            dec = ",", fill = TRUE, comment.char = "", ...)
```

#### Arguments

- 
- 
- 
-

# ...continued

## Arguments

`file`

the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an *absolute* path, the file name is *relative* to the current working directory, `getwd()`. Tilde-expansion is performed where supported. This can be a compressed file (see `file`).

Alternatively, `file` can be a readable text-mode `connection` (which will be opened for reading if necessary, and if so `closed` (and hence destroyed) at the end of the function call). (If `stdin()` is used, the prompts for lines may be somewhat confusing. Terminate input with a blank line or an EOF signal, `Ctrl-D` on Unix and `Ctrl-Z` on Windows. Any pushback on `stdin()` will be cleared before return.)

`file` can also be a complete URL. (For the supported URL schemes, see the ‘URLs’ section of the help for `url`.)

`header`

a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.

`sep`

the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default for `read.table`) the separator is ‘white space’, that is one or more spaces, tabs, newlines or carriage returns.

`quote`

the set of quoting characters. To disable quoting altogether, use `quote = ""`. See `scan` for the behaviour on quotes embedded in quotes. Quoting is only considered for columns read as character, which is all of them unless `colClasses` is specified.

`dec`

the character used in the file for decimal points.

`row.names`

a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered.

`col.names`

Using `row.names = NULL` forces row numbering. Missing or `NULL` `row.names` generate row names that are considered to be ‘automatic’ (and not preserved by `as.matrix`).  
a vector of optional names for the variables. The default is to use “v” followed by the column number.

`as.is`

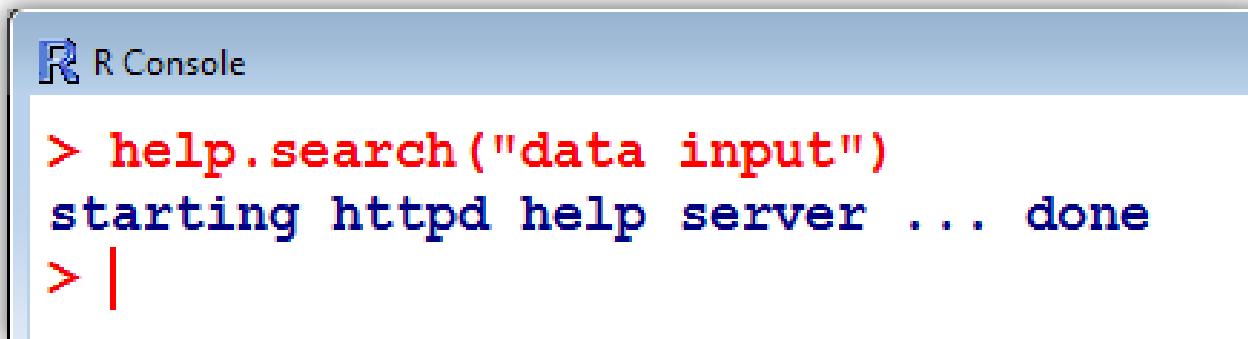
the default behavior of `read.table` is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable `as.is` controls the conversion of columns not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

**All minor details and explanations of all arguments are given.**

# Getting Help in R

4. Sometimes, you want to search by the subject on which we want help (e.g. data input). In such a case, type `help.search("data input")`



R Console

```
> help.search("data input")
starting httpd help server ... done
> |
```

Then we get....

# Then we get....

The screenshot shows a web browser window with the URL 127.0.0.1:24981/doc/html/Search?pattern=data input. The page title is "Search Results" and features the R logo. Below the title, it says "The search string was **"data input"**". Under "Help pages:", there are two entries: [utils::read.DIF](#) (Data Input from Spreadsheet) and [utils::read.table](#) (Data Input). A red arrow points upwards from the text "Clicking over the link give required information" to the link [utils::read.table](#).

127.0.0.1:24981/doc/html/Search?pattern=data input

Search Results 

The search string was **"data input"**

Help pages:

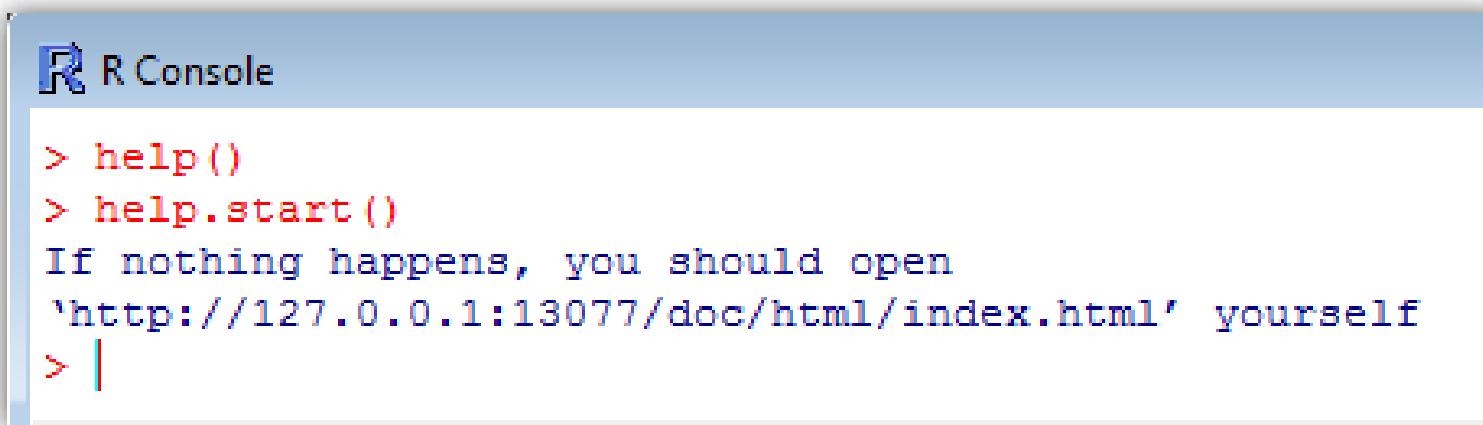
<a href="#">utils::read.DIF</a>	Data Input from Spreadsheet
<a href="#">utils::read.table</a>	Data Input

**Clicking over the link give required information**

# Getting Help in R

4. '`help()`' for on-line help,

or '`help.start()`' for an HTML browser interface to help.



R Console

```
> help()
> help.start()
If nothing happens, you should open
'http://127.0.0.1:13077/doc/html/index.html' yourself
> |
```

A screenshot of an R console window titled "R Console". The window shows the results of running the "help" and "help.start" commands. It outputs a message telling the user to open a specific URL if nothing happens. A cursor is visible at the bottom of the console window.

Then we get....

## Statistical Data Analysis



### Manuals

[An Introduction to R](#)  
[Writing R Extensions](#)  
[R Data Import/Export](#)

[The R Language Definition](#)  
[R Installation and Administration](#)  
[R Internals](#)

### Reference

[Packages](#)

[Search Engine & Keywords](#)

### Miscellaneous Material

[About R](#)  
[License](#)  
[NEWS](#)

[Authors](#)

[Resources](#)  
[Thanks](#)  
[Technical papers](#)

[Frequently Asked Questions](#)

[User Manuals](#)

### Material specific to the Windows port

[CHANGES up to R 2.15.0](#)

[Windows FAQ](#)

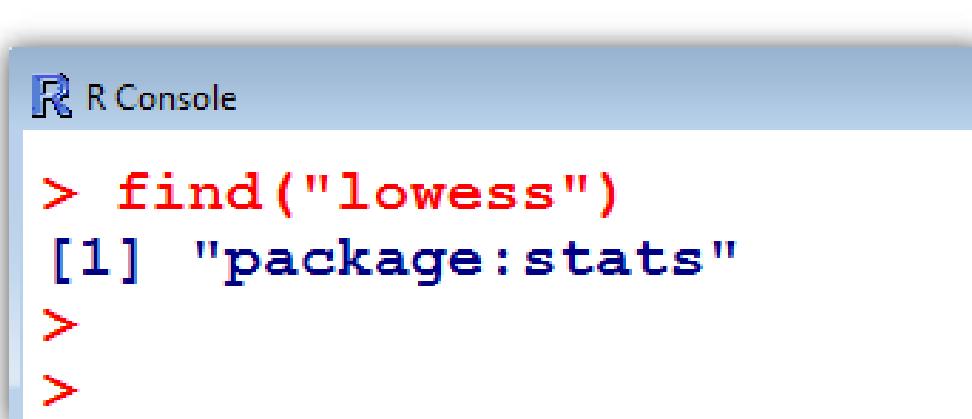
# Getting Help in R

5) Other useful functions are **find** and **apropos**.

6) The **find** function tells us what package something is in.

For example

```
> find("lowess") returns  
[1] "package:stats"
```



The image shows a screenshot of an R console window. The title bar says "R Console". The main area contains the following text:

```
> find("lowess")
[1] "package:stats"
>
>
```

# Getting Help in R

7) The apropos returns a character vector giving the names of all objects in the search list that match your enquiry.

`apropos("lm")` returns

R Console

```
> apropos("lm")
[1] ".__C__anova.glm"      ".__C__anova.glm.null" ".__C__glm"
[4] ".__C__glm.null"       ".__C__lm"                  ".__C__mlm"
[7] ".__C__optionalMethod" ".colMeans"               "anova.glm"
[10] "anova.glmlist"        "anova.lm"                 "anova.mlmlist"
[13] "anova.mlmlist"        "colMeans"                "contr.helmert"
[16] "getAllMethods"         "glm"                     "glm.control"
[19] "glm.fit"                "hatvalues.lm"             "KalmanForecast"
[22] "KalmanLike"              "KalmanRun"                "KalmanSmooth"
[25] "kappa.lm"                "lm"                      "lm.fit"
[28] "lm.influence"            "lm.wfit"                 "model.frame.glm"
[31] "model.frame.lm"          "model.matrix.lm"          "nlm"
[34] "nlminb"                   "plot.lm"                 "plot.mlmlist"
[37] "predict.glm"              "predict.lm"                "predict.mlmlist"
[40] "print.glm"                 "print.lm"                 "residuals.glm"
[43] "residuals.lm"              "rstandard.glm"             "rstandard.lm"
[46] "rstudent.glm"              "rstudent.lm"               "summary.glm"
[49] "summary.lm"
```

# Worked Examples of Functions

To see a worked `example` just type the function name, e.g., `lm` for linear models:

```
example(lm)
```

and we see the printed and graphical output produced by the `lm` function.

```
> example(lm)

lm> require(graphics)

lm> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
lm> ## Page 9: Plant Weight Data.
lm> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)

lm> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)

lm> group <- gl(2, 10, 20, labels = c("Ctl","Trt"))

lm> weight <- c(ctl, trt)

lm> lm.D9 <- lm(weight ~ group)

lm> lm.D90 <- lm(weight ~ group - 1) # omitting intercept

lm> ## No test:
lm> anova(lm.D9)
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7292 0.48496

lm> summary(lm.D90)
```

...and other details follow further

# Demonstration of R Functions

This can be useful for seeing the type of things that R can do.

`demo(persp)` [persp is a command for 3d surface plots]

```
R R Console
> demo(persp)

demo(persp)
---- ~~~~~

Type <Return> to start :

> ### Demos for persp() plots -- things not in example(persp)
> ### -----
>
> require(datasets)
>
> require(grDevices); require(graphics)
>
> ## (1) The Obligatory Mathematical surface.
> ##      Rotated sinc function.
>
> x <- seq(-10, 10, length.out = 50)
>
> y <- x
>
> rotsinc <- function(x,y)
+ {
+   sinc <- function(x) { y <- sin(x)/x ; y[is.na(y)] <- 1; y }
+   10 * sinc( sqrt(x^2+y^2) )
+ }

> sinc.exp <- expression(z == Sinc(sqrt(x^2 + y^2)))

> z <- outer(x, y, rotsinc)
>
> oldpar <- par(bg = "white")
>
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
Waiting to confirm page change...
```

Click or hit ENTER for next page

$z = \text{Sinc}(\sqrt{x^2 + y^2})$

...and it continues

# Demonstration of R Functions

This can be useful for seeing the type of things that R can do.

`demo(graphics)`

```
R Console
> demo(graphics)
  demo(graphics)
  -----
  Type <Return> to start :

> # Copyright (C) 1997-2009 The R Core Team
>
> require(datasets)

> require(grDevices); require(graphics)

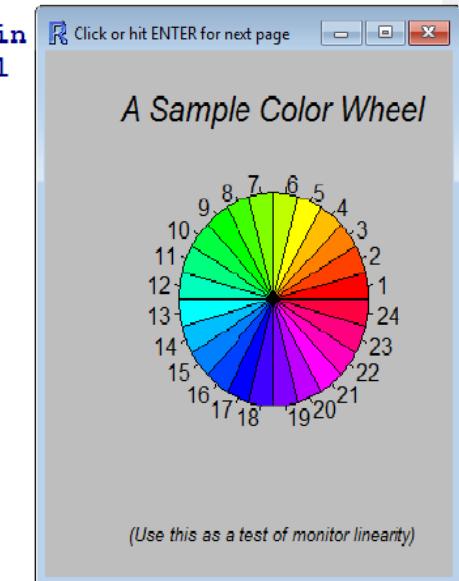
> ## A little color wheel.      This code just plots equally spaced hues in
> ## a pie chart.      If you have a cheap SVGA monitor (like me) you will
> ## probably find that numerically equispaced does not mean visually
> ## equispaced. On my display at home, these colors tend to cluster at
> ## the RGB primaries. On the other hand on the SGI Indy at work the
> ## effect is near perfect.
>
> par(bg = "gray")

> pie(rep(1,24), col = rainbow(24), radius = 0.9)
Waiting to confirm page change...

> title(main = "A Sample Color Wheel", cex.main = 1.4, font.main = 3)

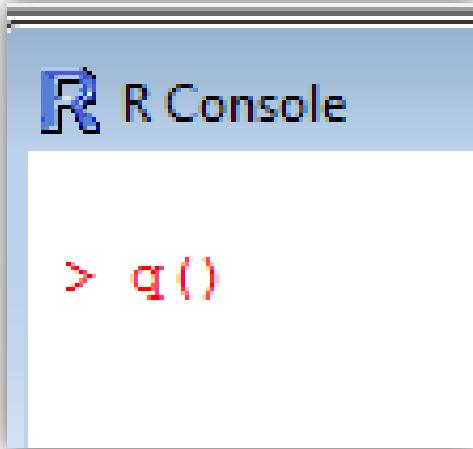
> title(xlab = "(Use this as a test of monitor linearity)",
+        cex.lab = 0.8, font.lab = 3)

> ## We have already confessed to having these. This is just showing off X11
> ## color names (and the example (from the postscript manual) is pretty "cute".
>
```



# How to quit in R

Type '`q()`' to quit R.



# Libraries in R

R provides many functions and one can also write own.

Functions and datasets are organised into libraries

To use a library, simply type the **library** function with the name of the library in brackets.

**library( . )**

For example, to load the **spatial** library type:

**library(spatial)**

# Libraries in R

Examples of libraries that come as a part of base package in R.

**MASS** : package associated with Venables and Ripley's book entitled *Modern Applied Statistics using S-Plus*.

**mgcv** : generalized additive models.

# Contents of Libraries

It is easy to use the `help` function to discover the contents of library packages.

Here is how we find out about the contents of the `spatial` library:

`library(help=spatial)` returns

Information on package 'spatial'

Description:

Package: `spatial`

Priority: recommended

Version: 7.3-8

followed by a list of all the functions and data sets.

Then we get....

```
> library(help=spatial)
```

```
>
```

```
R Documentation for package 'spatial'
```

### Information on package 'spatial'

#### Description:

Package: spatial  
Priority: recommended  
Version: 7.3-11  
Date: 2015-08-29  
Depends: R (>= 3.0.0), graphics, stats, utils  
Suggests: MASS  
Authors@R:  
c(person("Brian", "Ripley", role = c("aut", "cre",  
"cph"), email = "ripley@stats.ox.ac.uk"),  
person("Roger", "Bivand", role = "ctb"),  
person("William", "Venables", role = "cph"))  
Description: Functions for kriging and point pattern analysis.  
Title: Functions for Kriging and Point Pattern Analysis  
LazyLoad: yes  
ByteCompile: yes  
License: GPL-2 | GPL-3  
URL: <http://www.stats.ox.ac.uk/pub/MASS4/>  
NeedsCompilation: yes  
Packaged: 2015-08-28 15:25:37 UTC; ripley

# **Installing Packages and Libraries**

**The base R package contains programs for basic operations.**

**It does not contain some of the libraries necessary for advanced statistical work.**

**Specific requirements are met by special packages.**

**They are downloaded and their downloading is very simple.**

# Installing Packages and Libraries

To install any package,

- run the R program,
- then on the command line, use the `install.packages` function to download the libraries we want.

# Installing Packages and Libraries

Examples :

- The package **rmeta** contains the statistical tools for meta analysis.
- The package **Agreement** contains statistical tools for measuring agreement.

The packages **rmeta** or **Agreement** can be installed by

```
install.packages("rmeta")
```

```
install.packages("Agreement")
```

Then we get<sup>24</sup>...

```
> install.packages("rmeta")
Installing package into 'C:/Users/Shalabh/Documents/R/win-library/3.2'
(as 'lib' is unspecified)
trying URL 'https://cran.uni-muenster.de/bin/windows/contrib/3.2/rmeta_2.16.zip'
Content type 'application/zip' length 65469 bytes (63 KB)
downloaded 63 KB

package 'rmeta' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
      C:\Users\Shalabh\AppData\Local\Temp\RtmppekoVts\downloaded_packages
> |
```

```
> install.packages("Agreement")
Installing package into 'C:/Users/Shalabh/Documents/R/win-library/3.2'
(as 'lib' is unspecified)
also installing the dependency 'R2HTML'

trying URL 'https://cran.uni-muenster.de/bin/windows/contrib/3.2/R2HTML_2.3.2.zip'
Content type 'application/zip' length 447502 bytes (437 KB)
downloaded 437 KB

trying URL 'https://cran.uni-muenster.de/bin/windows/contrib/3.2/Agreement_0.8-0.zip'
Content type 'application/zip' length 69235 bytes (67 KB)
downloaded 67 KB

package 'R2HTML' successfully unpacked and MD5 sums checked
package 'Agreement' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
  C:\Users\Shalabh\AppData\Local\Temp\RtmppekoVts\downloaded_packages
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Basics and R as a Calculator**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Basics

- > is the prompt sign in R.
- The assignment operators are the left arrow with dash <- and equal sign =.

> **x** <- 20 assigns the value 20 to **x**.

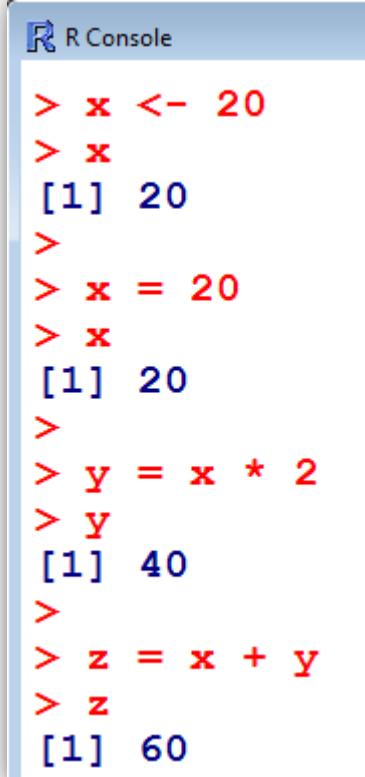
> **x** = 20 assigns the value 20 to **x**.

Initially only <- was available in R.

• > **x** = 20 assigns the value 20 to **x**.

> **y** = **x** \* 2 assigns the value **2\*x** to **y**.

> **z** = **x** + **y** assigns the value **x + y** to **z**.



R Console

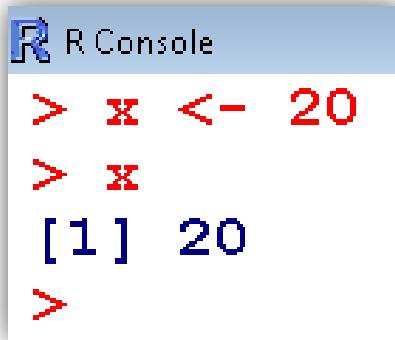
```
> x <- 20
> x
[1] 20
>
> x = 20
> x
[1] 20
>
> y = x * 2
> y
[1] 40
>
> z = x + y
> z
[1] 60
```

# Basics

- # : The character # marks the beginning of a comment. All characters until the end of the line are ignored.

```
> # mu is the mean
```

```
> # x <- 20 is treated as comment only
```



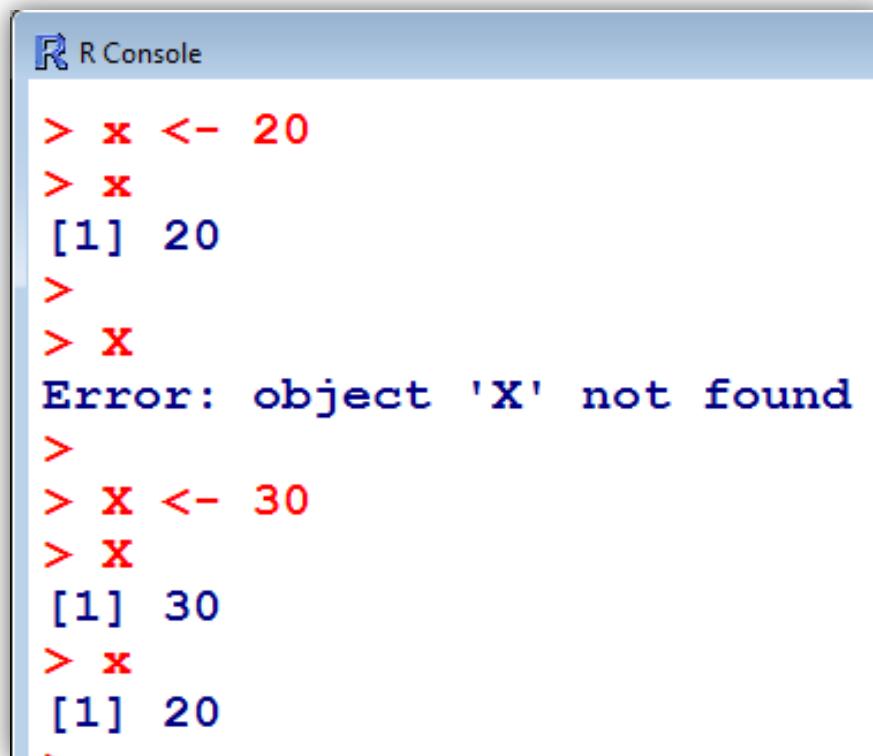
A screenshot of the R console window. The title bar says "R Console". The console area contains the following text:

```
> x <- 20
> x
[1] 20
>
```

# Basics

- Capital and small letters are different.

> **x** <- 20 and > x <- 20 are different



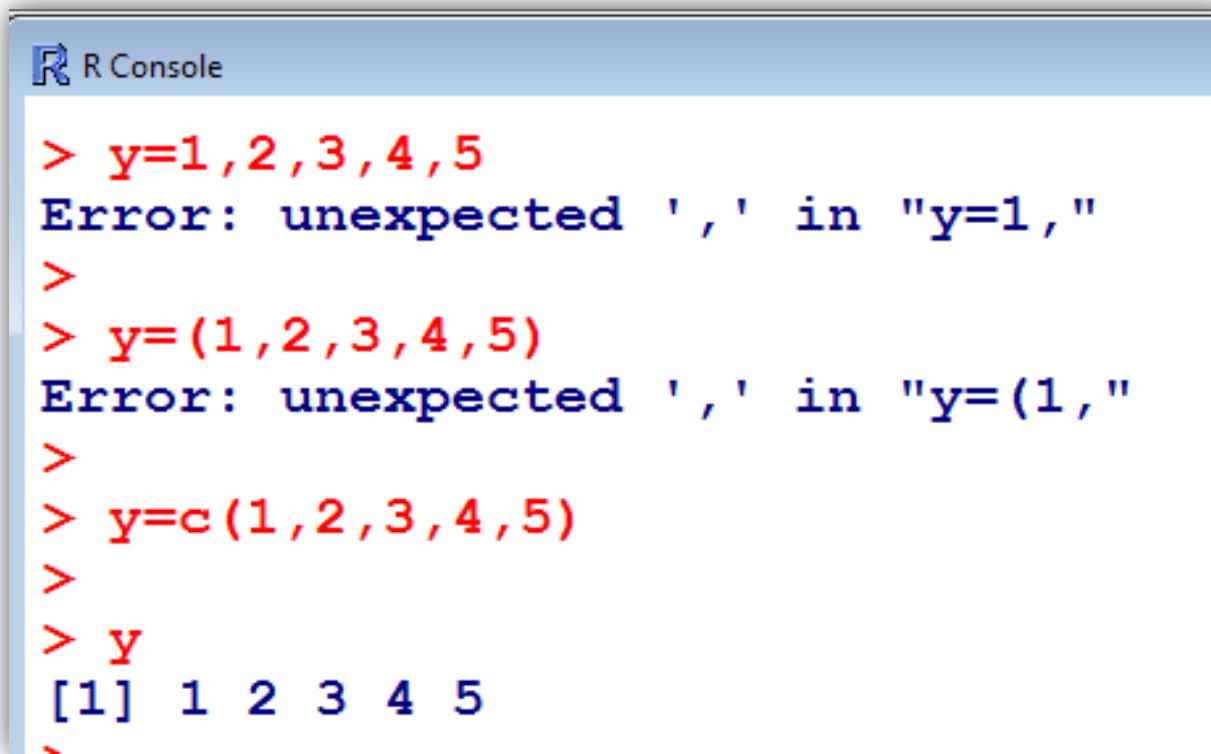
The screenshot shows an R console window titled "R Console". It displays the following R session:

```
> x <- 20
> x
[1] 20
>
> x
Error: object 'X' not found
>
> x <- 30
> x
[1] 30
> x
[1] 20
```

The session demonstrates that while the variable assignment `x <- 20` is successful, attempting to access it as `X` results in an error: "object 'X' not found". This serves as a visual example of how R distinguishes between capital and small letters in variable names.

# Basics

- The command `c(1,2,3,4,5)` combines the numbers 1,2,3,4 and 5 to a vector.



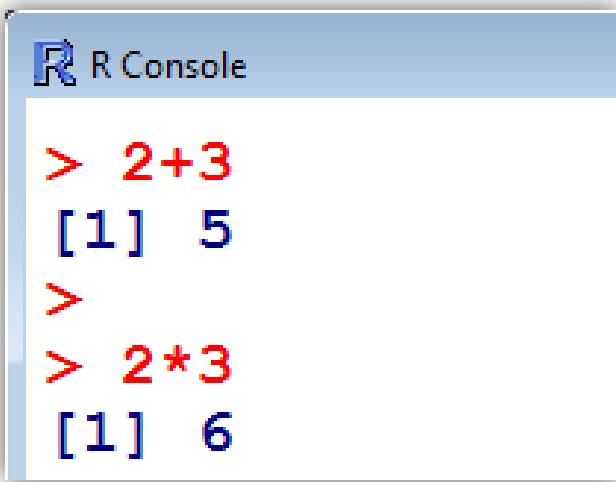
R Console

```
> y=1,2,3,4,5
Error: unexpected ',' in "y=1,"
>
> y=(1,2,3,4,5)
Error: unexpected ',' in "y=(1,"
>
> y=c(1,2,3,4,5)
>
> y
[1] 1 2 3 4 5
```

# R as a calculator

```
> 2+3          # Command  
[1] 5          # Output
```

```
> 2*3          # Command  
[1] 6          # Output
```



A screenshot of the R Console window. The title bar says "R Console". The main area contains the following text:

```
> 2+3  
[1] 5  
>  
> 2*3  
[1] 6
```

# R as a calculator

```
> 2-3          # Command
```

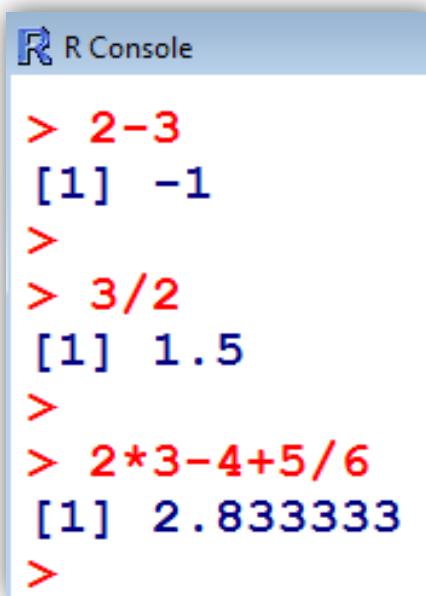
```
[1] -1         # Output
```

```
> 3/2          # Command
```

```
[1] 1.5        # Output
```

```
> 2*3-4+5/6   # Command
```

```
[1] 2.8333     # Output
```



A screenshot of the R Console window. The title bar says "R Console". The main area shows the command-line history:

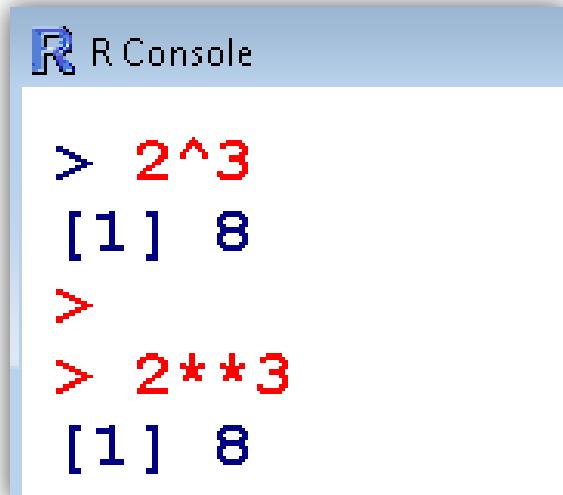
```
R Console

> 2-3
[1] -1
>
> 3/2
[1] 1.5
>
> 2*3-4+5/6
[1] 2.833333
>
```

# R as a calculator

```
> 2^3          # Command  
[1] 8          # Output
```

```
> 2**3         # Command  
[1] 8          # Output
```



A screenshot of the R Console window. The title bar says "R Console". The console area contains two sets of commands and outputs. The first set shows the command `> 2^3` in red and its output [1] 8 in blue. The second set shows the command `> 2**3` in red and its output [1] 8 in blue. The cursor is visible at the start of the next input line.

```
> 2^3  
[1] 8  
>  
> 2**3  
[1] 8
```

# R as a calculator

```
> 2^0.5      # Command
```

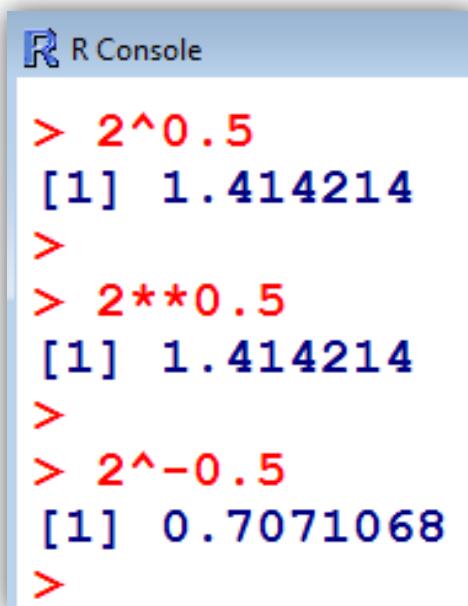
```
[1] 1.414214 # Output
```

```
> 2**0.5      # Command
```

```
[1] 1.414214 # Output
```

```
> 2^-0.5      # Command
```

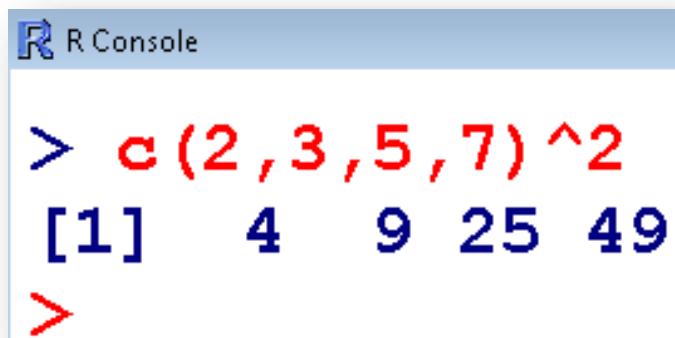
```
[1] 0.7071068 # Output
```



# R as a calculator

```
> c(2,3,5,7)^2      # command: application to a  
                     vector  
[1] 4 9 25 49       # output
```

$$2^2, 3^2, 5^2, 7^2$$



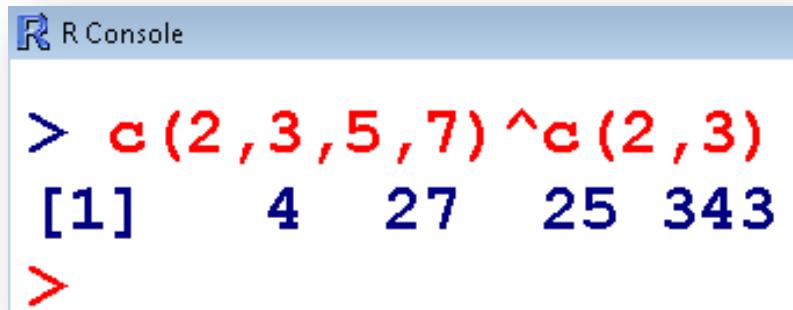
R Console

```
> c(2,3,5,7)^2
[1] 4 9 25 49
>
```

# R as a calculator

```
> c(2,3,5,7)^c(2,3) # !!ATTENTION! Observe the  
# operation  
[1] 4 27 25 343 # output
```

$$2^2, 3^3, 5^2, 7^3$$



R Console

```
> c(2,3,5,7)^c(2,3)
[1] 4 27 25 343
>
```

# R as a calculator

```
> c(1,2,3,4,5,6)^c(2,3,4) # command: application  
                           to a vector with vector  
[1] 1 8 81 16 125 1296   # output
```

$$1^2, 2^3, 3^4, 4^2, 5^3, 6^4$$

R R Console

```
> c(1,2,3,4,5,6)^c(2,3,4)  
[1] 1 8 81 16 125 1296  
>
```

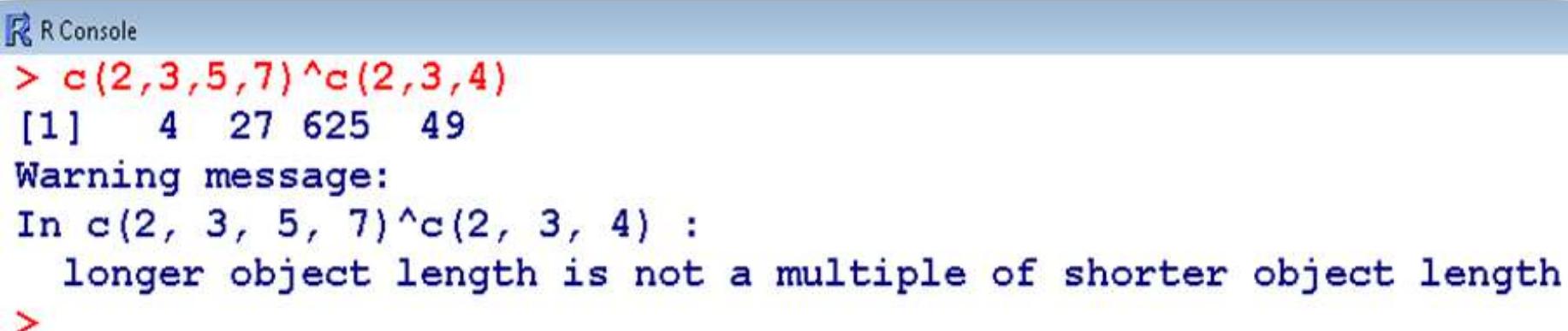
# R as a calculator

```
> c(2,3,5,7)^c(2,3,4)      #error message  
[1] 4 27 625 49            # output
```

Warning message:

longer object length is not a multiple of  
shorter object length in: c(2,3,5,7)^c(2,3,4)

$$2^2, 3^3, 5^4, 7^2$$



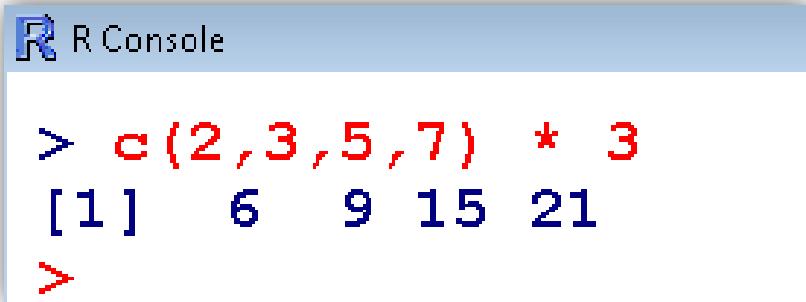
R Console

```
> c(2,3,5,7)^c(2,3,4)  
[1] 4 27 625 49  
Warning message:  
In c(2, 3, 5, 7)^c(2, 3, 4) :  
  longer object length is not a multiple of shorter object length  
>
```

# Multiplication and Division $x * y$ , $x/y$ :

```
> c(2,3,5,7) * 3  
[1] 6 9 15 21
```

$2 \times 3$ ,  $3 \times 3$ ,  $5 \times 3$ ,  $7 \times 3$



A screenshot of an R console window. The title bar says "R Console". The main area contains the following text:

```
> c(2,3,5,7) * 3  
[1] 6 9 15 21  
>
```

# Multiplication and Division $x * y$ , $x/y$ :

```
> c(2,3,5,7) * c(-2,-3,-5,8)  
[1] -4 -9 -25 56
```

$$2 \times (-2), \quad 3 \times (-3), \quad 5 \times (-5), \quad 7 \times 8$$

R R Console

```
> c(2,3,5,7) * c(-2,-3,-5,8)  
[1] -4 -9 -25 56  
>
```

# Multiplication and Division $x * y$ , $x/y$ :

```
> c(2,3,5,7) * c(8,9) # !!! ATTENTION  
[1] 16 27 40 63
```

$$2 \times 8, \quad 3 \times 9, \quad 5 \times 8, \quad 7 \times 9$$

R R Console

```
> c(2,3,5,7) * c(8,9)  
[1] 16 27 40 63
```

# Multiplication and Division $x * y$ , $x/y$ :

```
> c(2,3,5,7) * c(8,9,10) # error message  
[1] 16 27 50 56
```

Warning message:

longer object length

is not a multiple of shorter object length

in: c(2, 3, 5, 7) \* c(8, 9, 10)

$2 \times 8$ ,  $3 \times 9$ ,  $5 \times 10$ ,  $7 \times 8$

R Console

```
> c(2,3,5,7) * c(8,9,10)  
[1] 16 27 50 56  
Warning message:  
In c(2, 3, 5, 7) * c(8, 9, 10) :  
  longer object length is not a multiple of shorter object length  
> |
```

# Addition and Subtraction $x + y$ , $x - y$

```
> c(2,3,5,7) + 10  
[1] 12 13 15 17
```

2+10, 3+10, 5+10, 7+10

R R Console

```
> c(2,3,5,7) + 10  
[1] 12 13 15 17
```

# Addition and Subtraction $x + y$ , $x - y$

```
> c(2,3,5,7) + c(-2,-3, -5, 8)  
[1] 0 0 0 15
```

$$2+(-2), \quad 3+(-3), \quad 5+(-5), \quad 7+8$$

R R Console

```
> c(2,3,5,7) + c(-2,-3, -5, 8)  
[1] 0 0 0 15
```

# Addition and Subtraction $x + y$ , $x - y$

```
> c(2,3,5,7) + c(8,9) # !!! ATTENTION!  
[1] 10 12 13 16
```

$$2+8, \ 3+9, \ 5+2, \ 7+9$$



```
> c(2,3,5,7) + c(8,9)  
[1] 10 12 13 16
```

# Addition and Subtraction $x + y$ , $x - y$

```
> c(2,3,5,7) + c(8,9,10) # error message
[1] 10 12 15 15
Warning message:
longer object length
is not a multiple of shorter object length
in: c(2, 3, 5, 7) + c(8, 9, 10)
```

2+8, 3+9, 5+10, 7+8

```
R R Console
> c(2,3,5,7) + c(8,9,10)
[1] 10 12 15 15
Warning message:
In c(2, 3, 5, 7) + c(8, 9, 10) :
  longer object length is not a multiple of shorter object length
>
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **R as Calculator, Built in Functions and Assignments**

**Shalabh**

**Department of Mathematics and Statistics**

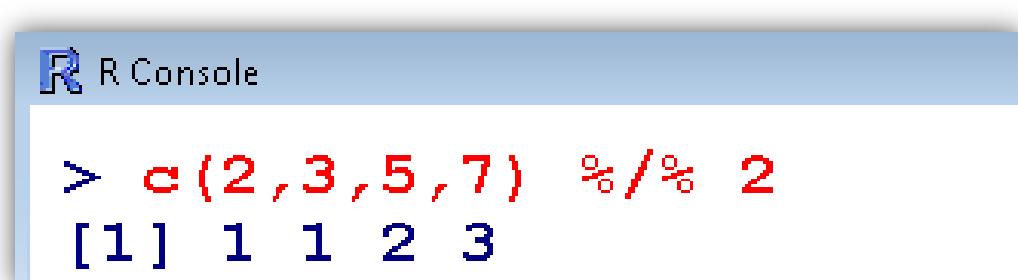
**Indian Institute of Technology Kanpur**

# Integer Division %/%

**Integer Division:** Division in which the fractional part (remainder) is discarded

```
> c(2,3,5,7) %/% 2  
[1] 1 1 2 3
```

```
2% / %2, 3% / %2, 5% / %2, 7% / %2
```



A screenshot of an R console window titled "R Console". The window shows the command `> c(2,3,5,7) %/% 2` followed by the output `[1] 1 1 2 3`. The text in the console is colored red and blue.

```
> c(2,3,5,7) %/% 2  
[1] 1 1 2 3
```

# Integer Division %/%

**Integer Division:** Division in which the fractional part (remainder) is discarded

```
> c(2,3,5,7) %/% c(2,3)  
[1] 1 1 2 2
```

```
2% / %2, 3% / %3, 5% / %2, 7% / %3
```

R Console

```
> c(2,3,5,7) %/% c(2,3)  
[1] 1 1 2 2
```

# Modulo Division (x mod y) %%:

x mod y : modulo operation finds the remainder after division of one number by another

```
> c(2,3,5,7) %% 2  
[1] 0 1 1 1
```

```
2%%2, 3%%2, 5%%2, 7%%2
```

R R Console

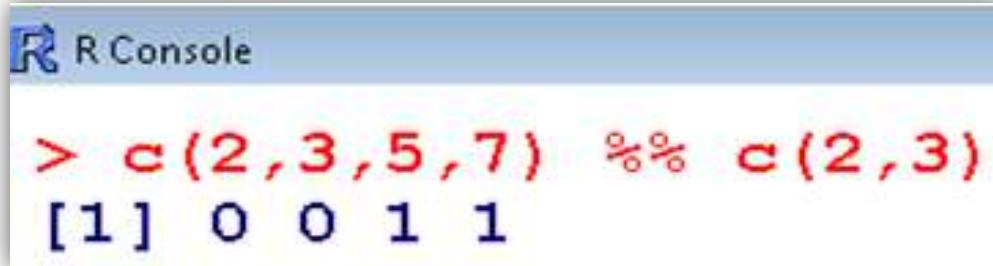
```
> c(2,3,5,7) %% 2  
[1] 0 1 1 1
```

## Modulo Division (x mod y) %%:

x mod y : modulo operation finds the remainder after division of one number by another

```
> c(2,3,5,7) %% c(2,3)  
[1] 0 0 1 1
```

```
2%%2, 3%%3, 5%%2, 7%%3
```



A screenshot of an R console window titled "R Console". The window shows the command `> c(2,3,5,7) %% c(2,3)` in red, followed by the output `[1] 0 0 1 1` in blue.

```
R Console  
> c(2,3,5,7) %% c(2,3)  
[1] 0 0 1 1
```

# Maximum: max

```
> max(1.2, 3.4, -7.8)  
[1] 3.4
```

R R Console

```
> max(1.2, 3.4, -7.8)  
[1] 3.4  
>  
>  
> max( c(1.2, 3.4, -7.8) )  
[1] 3.4  
>  
> |
```

```
> max( c(1.2, 3.4, -7.8) )  
[1] 3.4
```

# Minimum : min

```
> min(1.2, 3.4, -7.8)
[1] -7.8
```

R R Console

```
> min(1.2, 3.4, -7.8)
[1] -7.8
>
> min( c(1.2, 3.4, -7.8) )
[1] -7.8
>
```

```
> min( c(1.2, 3.4, -7.8) )
[1] -7.8
```

# Overview Over Further Functions

<code>abs()</code>	Absolute value
<code>sqrt()</code>	Square root
<code>round()</code> , <code>floor()</code> , <code>ceiling()</code>	Rounding, up and down
<code>sum()</code> , <code>prod()</code>	Sum and product
<code>log()</code> , <code>log10()</code> , <code>log2()</code>	Logarithms
<code>exp()</code>	Exponential function
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , <code>asin()</code> , <code>acos()</code> , <code>atan()</code>	Trigonometric functions
<code>sinh()</code> , <code>cosh()</code> , <code>tanh()</code> , <code>asinh()</code> , <code>acosh()</code> , <code>atanh()</code>	Hyperbolic functions

# Examples

```
> abs(-4)
```

```
[1] 4
```

```
> abs(c(-1,-2,-3,4,5))
```

```
[1] 1 2 3 4 5
```

R R Console

```
> abs(-4)
```

```
[1] 4
```

```
>
```

```
> abs(c(-1,-2,-3,4,5))
```

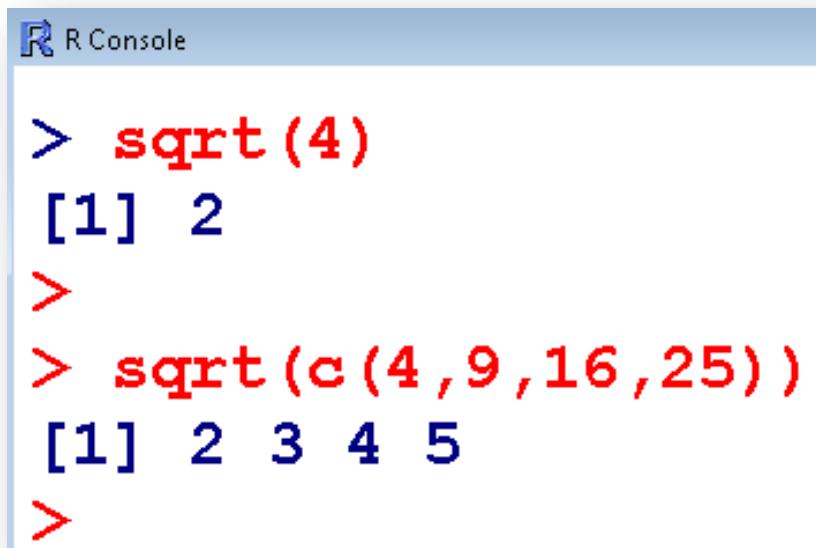
```
[1] 1 2 3 4 5
```

```
>
```

# Examples

```
> sqrt(4)  
[1] 2
```

```
> sqrt(c(4,9,16,25))  
[1] 2 3 4 5
```

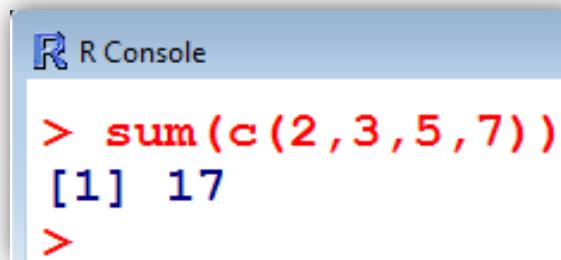


R Console

```
> sqrt(4)  
[1] 2  
>  
> sqrt(c(4,9,16,25))  
[1] 2 3 4 5  
>
```

# Examples

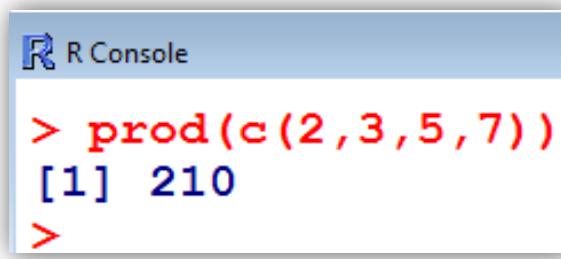
```
> sum(c(2,3,5,7))  
[1] 17
```



R Console

```
> sum(c(2,3,5,7))  
[1] 17  
>
```

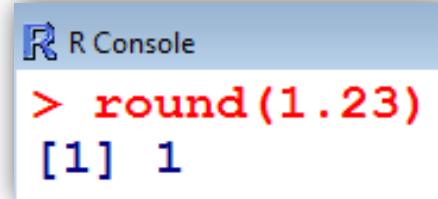
```
> prod(c(2,3,5,7))  
[1] 210
```



R Console

```
> prod(c(2,3,5,7))  
[1] 210  
>
```

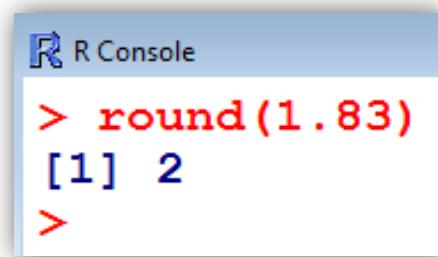
```
> round(1.23)  
[1] 1
```



R Console

```
> round(1.23)  
[1] 1
```

```
> round(1.83)  
[1] 2
```



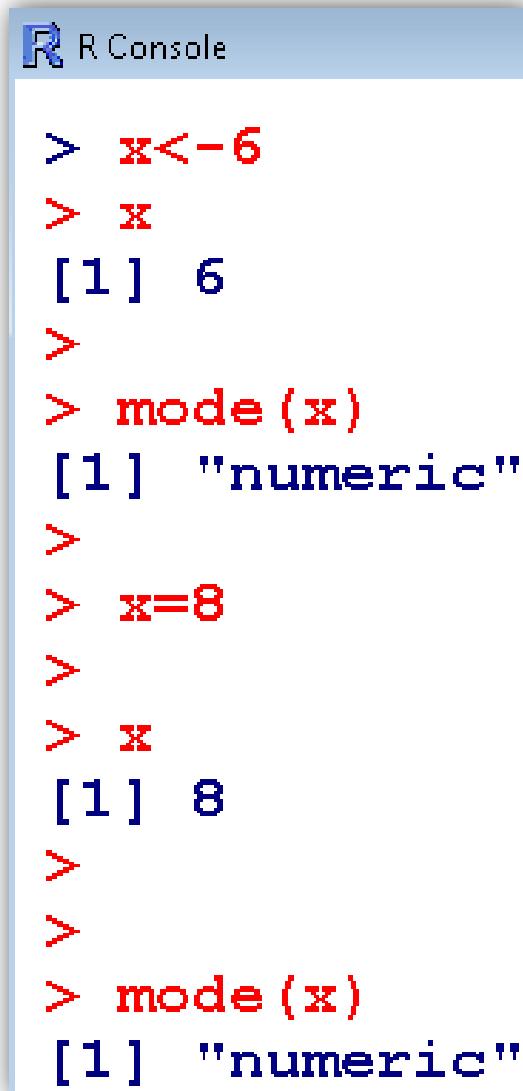
R Console

```
> round(1.83)  
[1] 2  
>
```

# Assignments

Assignments can be made in two ways:

```
> x<-6  
> x  
[1] 6  
  
> mode(x)  
[1] "numeric"  
  
> x=8  
> x  
[1] 8  
  
> mode(x)  
[1] "numeric"
```



A screenshot of the R console window titled "R Console". The window shows two sets of R code side-by-side. The left set shows assignment via the operator, and the right set shows assignment via the equals sign. Both sets show the variable being assigned the value 6 or 8, and then checking its mode, which returns "numeric".

```
R Console  
> x<-6  
> x  
[1] 6  
>  
> mode(x)  
[1] "numeric"  
>  
> x=8  
>  
> x  
[1] 8  
>  
> mode(x)  
[1] "numeric"
```

# Assignments

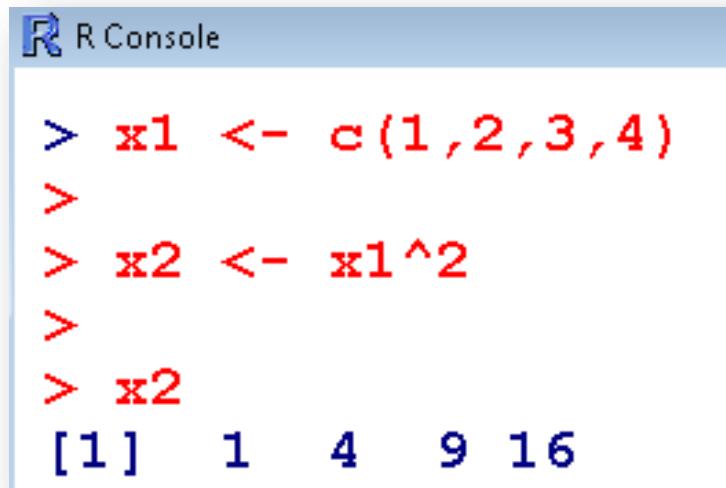
An assignment can also be used to save values in variables:

```
> x1 <- c(1,2,3,4)
```

```
> x2 <- x1^2
```

```
> x2
```

```
[1] 1 4 9 16
```



A screenshot of the R Console window. The title bar says "R Console". The main area contains the following R code and its output:

```
> x1 <- c(1,2,3,4)
>
> x2 <- x1^2
>
> x2
[1] 1 4 9 16
```

ATTENTION: R is case sensitive (X is not the same as x)

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Functions and Matrices**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Functions

- Functions are a bunch of commands grouped together in a sensible unit
- Functions take input arguments, do calculations (or make some graphics, call other functions) and produce some output and return a result in a variable. The returned variable can be a complex construct, like a list.

# Functions

## Syntax

```
Name <- function(Argument1, Argument2, . . .)  
{  
    expression  
}
```

where **expression** is a single command or a group of commands

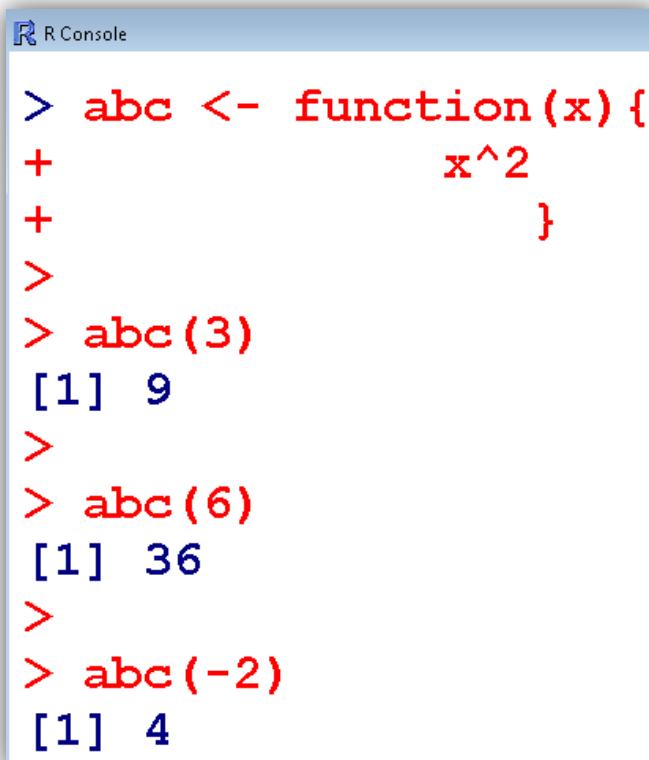
## **Function arguments with description and default values**

- Function arguments can be given a meaningful name
- Function arguments can be set to default values
- Functions can have the special argument '...'

# Functions (Single variable)

The sign `<-` is furthermore used for defining functions:

```
> abc <- function(x) {  
+   x^2  
+ }  
  
> abc(3)  
[1] 9  
  
> abc(6)  
[1] 36  
  
> abc(-2)  
[1] 4
```

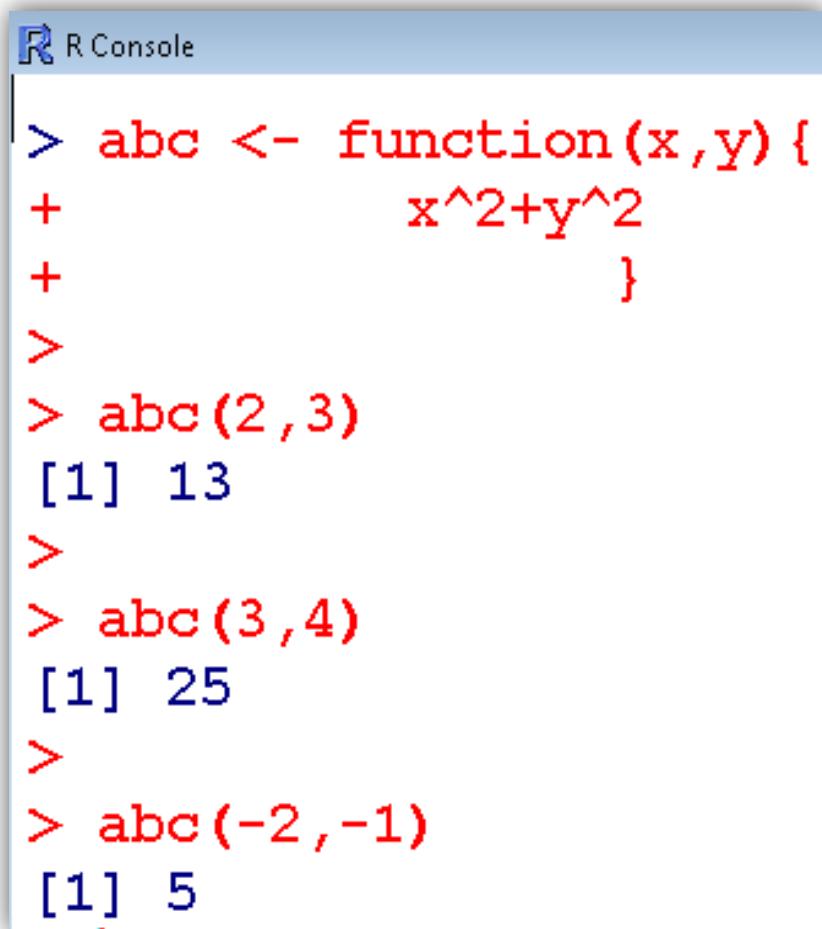


A screenshot of the R Console window. The title bar says "R Console". The console area contains the following text:

```
> abc <- function(x) {  
+   x^2  
+ }  
>  
> abc(3)  
[1] 9  
>  
> abc(6)  
[1] 36  
>  
> abc(-2)  
[1] 4
```

# Functions (Two variables)

```
> abc <- function(x,y) {  
+   x^2+y^2  
+ }  
  
> abc(2,3)  
[1] 13  
  
> abc(3,4)  
[1] 25  
  
> abc(-2,-1)  
[1] 5
```



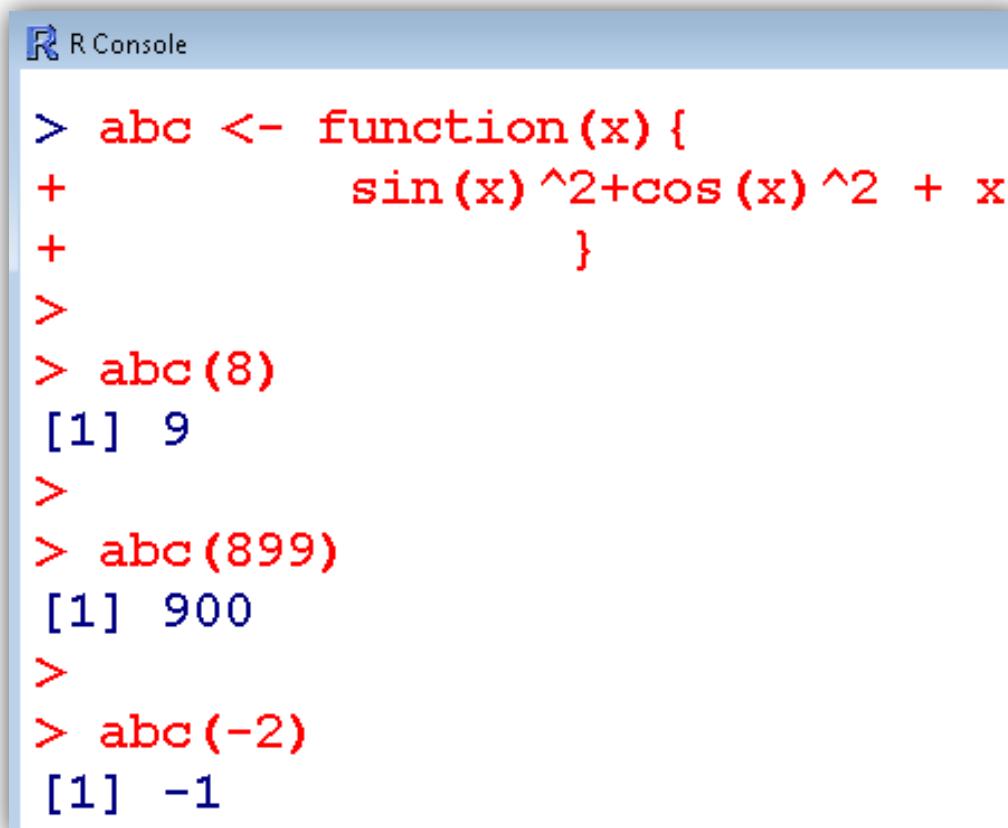
The screenshot shows the R console interface with a blue header bar labeled "R Console". Below the header, there is a scrollable text area containing R code. The code defines a function "abc" that takes two arguments, x and y, and returns their sum of squares. It then demonstrates four calls to this function with different argument pairs, resulting in outputs [1] 13, [1] 25, and [1] 5 respectively.

```
R Console  
> abc <- function(x,y) {  
+   x^2+y^2  
+ }  
>  
> abc(2,3)  
[1] 13  
>  
> abc(3,4)  
[1] 25  
>  
> abc(-2,-1)  
[1] 5
```

# Functions- Another example

```
> abc <- function(x) {  
  sin(x)^2+cos(x)^2 + x  
}
```

```
> abc(8)  
[1] 9  
  
> abc(899)  
[1] 900  
  
> abc(-2)  
[1] -1
```



The image shows a screenshot of an R console window titled "R Console". The console displays the following text:

```
> abc <- function(x) {  
+   sin(x)^2+cos(x)^2 + x  
+ }  
  
>  
> abc(8)  
[1] 9  
  
>  
> abc(899)  
[1] 900  
  
>  
> abc(-2)  
[1] -1
```

The text is color-coded: blue for user input and red for R's internal processing. The R logo icon is visible in the top-left corner of the window.

# Matrix

Matrices are important objects in any calculation.

A matrix is a rectangular array with  $p$  rows and  $n$  columns.

An element in the  $i$ -th row and  $j$ -th column is denoted by  $X_{ij}$  (book version) or  $X[i, j]$  ("program version"),  $i = 1, 2, \dots, n$ ,  $j = 1, 2, \dots, p$ .

An element of a matrix can also be an object, for example a string.

However, in mathematics, we are mostly interested in numerical matrices, whose elements are generally real numbers

In R, a  $4 \times 2$ -matrix  $X$  can be created with a following command:

```
> x <- matrix( nrow=4, ncol=2,
                 data=c(1,2,3,4,5,6,7,8) )
> x
      [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

R Console

```
> x <- matrix( nrow=4, ncol=2, data=c(1,2,3,4,5,6,7,8) )
>
> x
      [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

We see:

The parameter `nrow` defines the row number of a matrix.

The parameter `ncol` defines the column number of a matrix.

The parameter `data` assigns specified values to the matrix elements.

The values from the parameters are written column-wise in matrix.

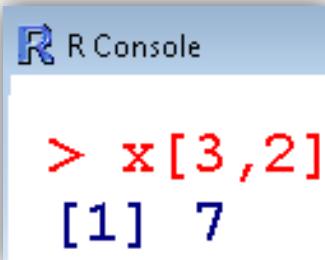
```
> x
```

	[,1]	[,2]
[1 ,]	1	5
[2 ,]	2	6
[3 ,]	3	7
[4 ,]	4	8

One can access a single element of a matrix with `x[i,j]`:

```
> x[3,2]
```

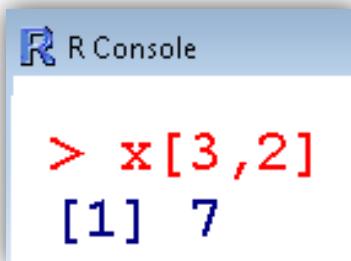
```
[1] 7
```



```
> x  
      [,1] [,2]  
[1,]    1    5  
[2,]    2    6  
[3,]    3    7  
[4,]    4    8
```

One can access a single element of a matrix with **x[i,j]** :

```
> x[3,2]  
[1] 7
```



In case, the data has to be entered row wise, then a  $4 \times 2$ -matrix  $X$  can be created with

```
> x <- matrix( nrow=4, ncol=2,
+               data=c(1,2,3,4,5,6,7,8), byrow = TRUE)
> x
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

R RGui (64-bit)

```
> x <- matrix( nrow=4, ncol=2, data=c(1,2,3,4,5,6,7,8) )  
> x  
     [,1] [,2]  
[1,]    1    5  
[2,]    2    6  
[3,]    3    7  
[4,]    4    8  
>  
> x <- matrix( nrow=4, ncol=2, data=c(1,2,3,4,5,6,7,8), byrow = TRUE)  
> x  
     [,1] [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Matrix Operations**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

In R, a  $4 \times 2$ -matrix  $X$  can be created with a following command:

```
> x <- matrix( nrow=4, ncol=2,
                 data=c(1,2,3,4,5,6,7,8) )
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

R R Console

```
> x <- matrix( nrow=4, ncol=2, data=c(1,2,3,4,5,6,7,8) )
>
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

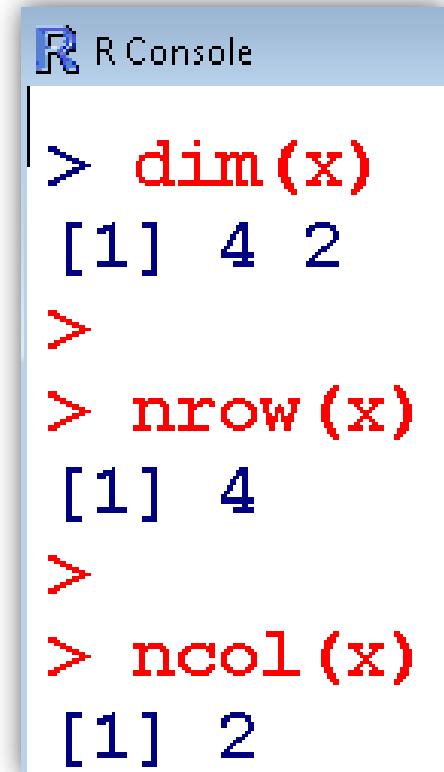
# Properties of a Matrix

We can get specific *properties* of a matrix:

```
> dim(x) # tells the  
[1] 4 2      dimension of matrix
```

```
> nrow(x) # tells  
[1] 4      the number of rows
```

```
> ncol(x) # tells  
[1] 2      the number of columns
```

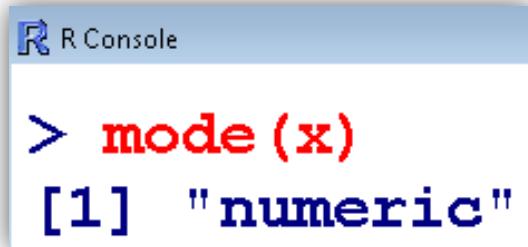


The image shows a window titled "R Console" with a light blue header bar. Inside, there is R code demonstrating how to find the dimensions of a matrix. The code consists of several red and blue lines:

- A red line starting with ">" followed by "dim(x)" in blue.
- A blue line showing the output "[1] 4 2".
- A red line starting with ">".
- A red line starting with ">" followed by "nrow(x)" in blue.
- A blue line showing the output "[1] 4".
- A red line starting with ">".
- A red line starting with ">" followed by "ncol(x)" in blue.
- A blue line showing the output "[1] 2".

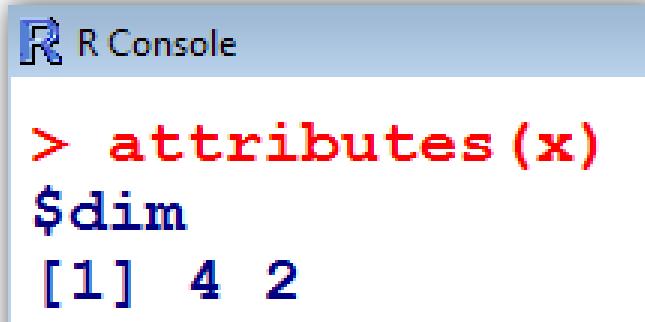
# Properties of a Matrix

```
> mode(x)      # Informs the type or storage  
mode of an object, e.g.,  
numerical, logical etc.  
[1] "numeric"
```



**attributes** provides all the attributes of an object

```
> attributes(x) #Informs the dimension of matrix  
$dim [1] 4 2
```



# Help on the Object "Matrix"

To know more about these important objects, we use R-help on "matrix".

```
> help("matrix")
```

```
matrix      package:base      R Documentation
```

## Matrices

### Description:

'matrix' creates a matrix from the given set of values.

'as.matrix' attempts to turn its argument into a matrix.

'is.matrix' tests if its argument is a (strict) matrix. It is generic: you can write methods to handle specific classes of objects, see Internal Methods.

Then we get an overview on how a matrix can be created and what parameters are available:

Usage:

```
matrix(data [= NA, nrow = 1, ncol = 1, byrow = FALSE,  
dimnames = NULL)  
as.matrix(x)  
is.matrix(x)
```

Arguments:

**data**: an optional data vector.  
**nrow**: the desired number of rows  
**ncol**: the desired number of columns  
**byrow**: logical. If 'FALSE' (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.

**dimnames**: A 'dimnames' attribute for the matrix: a 'list' of length 2.

**x**: an R object.

Then, the meaning of each parameter is explained:

**Details:**

If either of 'nrow' or 'ncol' is not given, an attempt is made to infer it from the length of 'data' and the other parameter.

If there are too few elements in 'data' to fill the array, then the elements in 'data' are recycled. If 'data' has length zero, 'NA' of an appropriate type is used for atomic vectors and 'NULL' for lists.

'is.matrix' returns 'TRUE' if 'x' is a matrix (i.e., it is not a 'data.frame' and has a 'dim' attribute of length 2) and 'FALSE' otherwise.

'as.matrix' is a generic function. The method for data frames will convert any non-numeric/complex column into a character vector using 'format' and so return a character matrix, except that all-logical data frames will be coerced to a logical matrix.

**Finally, references and cross-references are displayed...**

**References:**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.

**See Also:**

'`data.matrix`', which attempts to convert to a numeric matrix.

.. as well as an example:

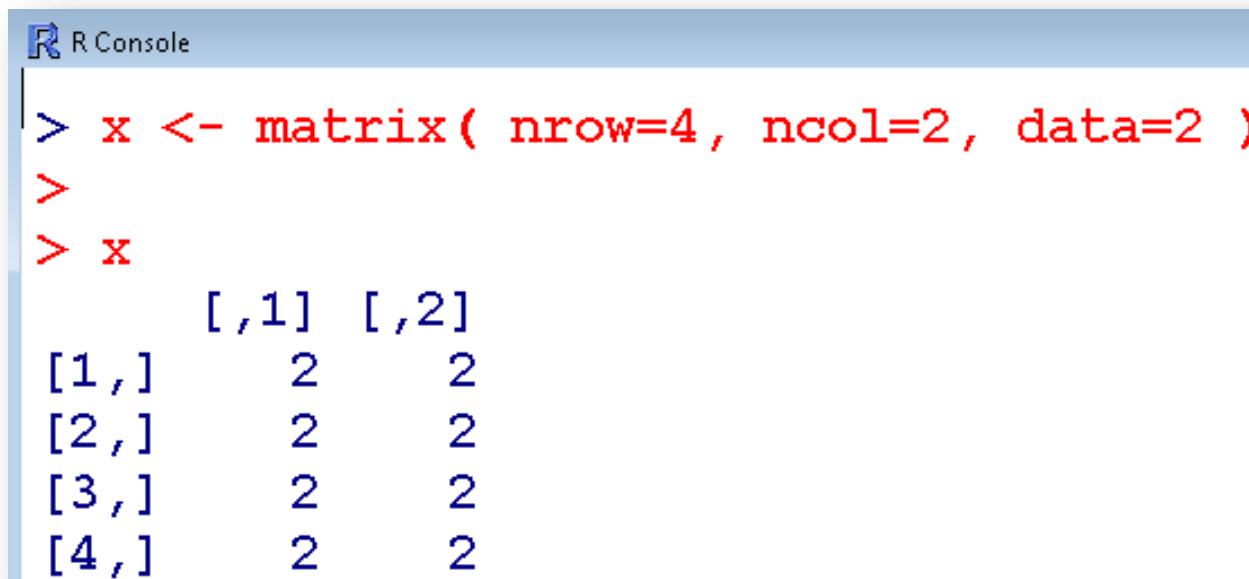
**Examples:**

```
is.matrix(as.matrix(1:10))
data(warpbreaks)
!is.matrix(warpbreaks) # data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,]) #using
  as.matrix.data.frame(.) method
```

# Matrix Operations

- Assigning a specified number to all matrix elements:

```
> x <- matrix( nrow=4, ncol=2, data=2 )  
> x  
      [,1] [,2]  
[1,]    2    2  
[2,]    2    2  
[3,]    2    2  
[4,]    2    2
```



The screenshot shows the R console interface. The title bar says "R Console". The main area contains the R code and its output. The code is identical to the one shown above, creating a 4x2 matrix with all elements set to 2. The output shows the matrix structure with columns labeled [,1] and [,2], and rows labeled [1,] through [4,].

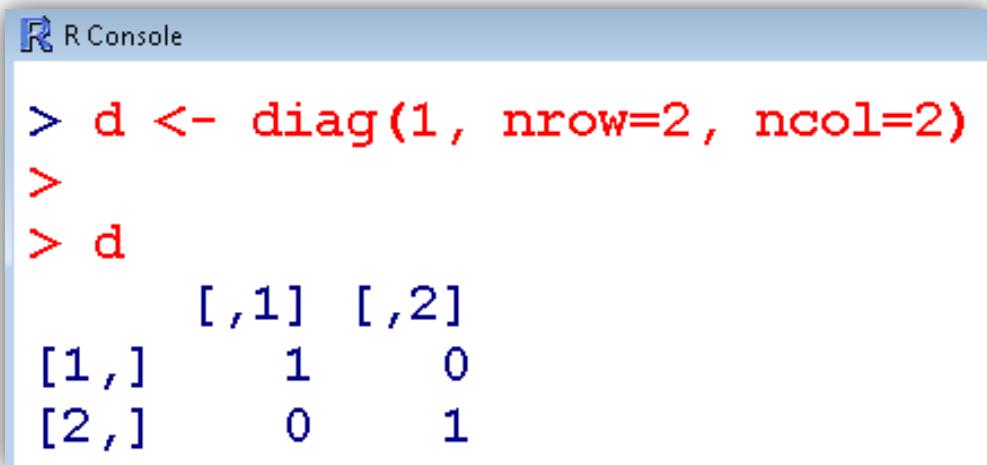
```
> x <- matrix( nrow=4, ncol=2, data=2 )  
>  
> x  
      [,1] [,2]  
[1,]    2    2  
[2,]    2    2  
[3,]    2    2  
[4,]    2    2
```

# Matrix Operations

- Construction of a diagonal matrix, here the identity matrix of a dimension 2:

```
> d <- diag(1, nrow=2, ncol=2)

> d
     [,1] [,2]
[1,]   1   0
[2,]   0   1
```



The screenshot shows the R Console window with the following text:

```
R Console

> d <- diag(1, nrow=2, ncol=2)
>
> d
     [,1] [,2]
[1,]   1   0
[2,]   0   1
```

- **Transpose of a matrix  $X$ :  $X'$**

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
> x  
      [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8
```

R R Console

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
>  
> x  
      [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8
```

- **Transpose of a matrix  $X$ :  $X'$**

```
> xt <- t(x)
```

```
> xt
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	3	5	7
[2,]	2	4	6	8

R R Console

```
> xt <- t(x)
> xt
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> |
```

- **Multiplication of a matrix with a constant**

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
  
> x  
     [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8
```

R R Console

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
>  
> x  
     [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8  
> |
```

- **Multiplication of a matrix with a constant**

```
> 4*x
```

	[,1]	[,2]
[1,]	4	8
[2,]	12	16
[3,]	20	24
[4,]	28	32

R R Console

```
> 4*x
```

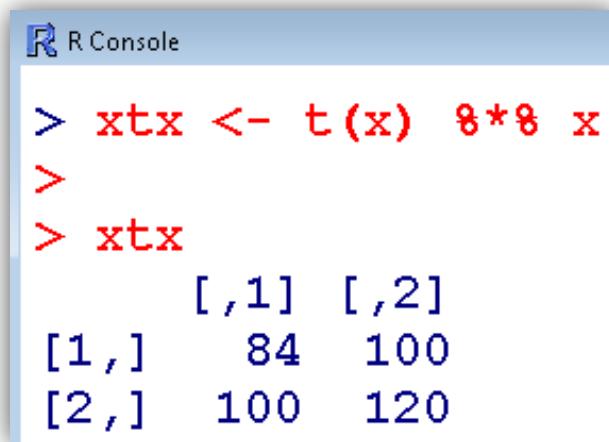
	[,1]	[,2]
[1,]	4	8
[2,]	12	16
[3,]	20	24
[4,]	28	32

- **Matrix multiplication: operator %\*%**

Consider the multiplication of  $X'$  with  $X$

```
> xtx <- t(x) %*% x
```

```
> xtx
      [,1]    [,2]
[1,]    84    100
[2,]   100    120
```



A screenshot of an R console window titled "R Console". The window shows the following R code and its output:

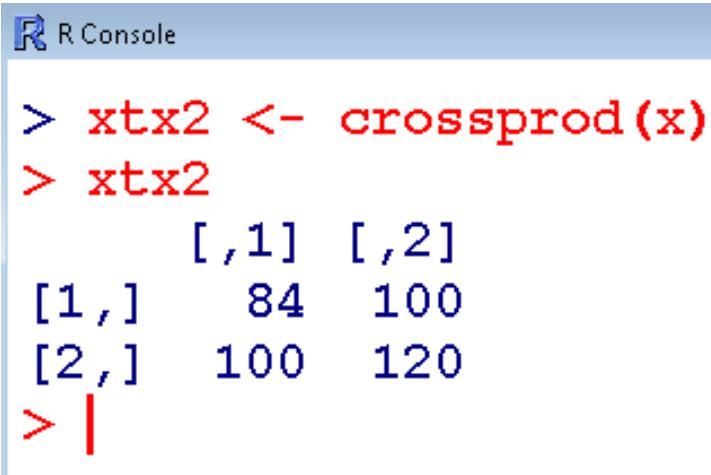
```
> xtx <- t(x) %*% x
>
> xtx
      [,1]    [,2]
[1,]    84    100
[2,]   100    120
```

- Cross product of a matrix  $X$ ,  $X'X$ , with a function `crossprod`

```
> xtx2 <- crossprod(x)
```

```
> xtx2
```

	[,1]	[,2]
[1,]	84	100
[2,]	100	120



R Console

```
> xtx2 <- crossprod(x)
> xtx2
      [,1] [,2]
[1,]    84   100
[2,]   100   120
> |
```

Note: Command `crossprod()` executes the multiplication faster than the conventional method with `t(x) %*% x`

- **Addition and subtraction of matrices (of same dimensions) can be executed with the usual operators + and -**

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T)

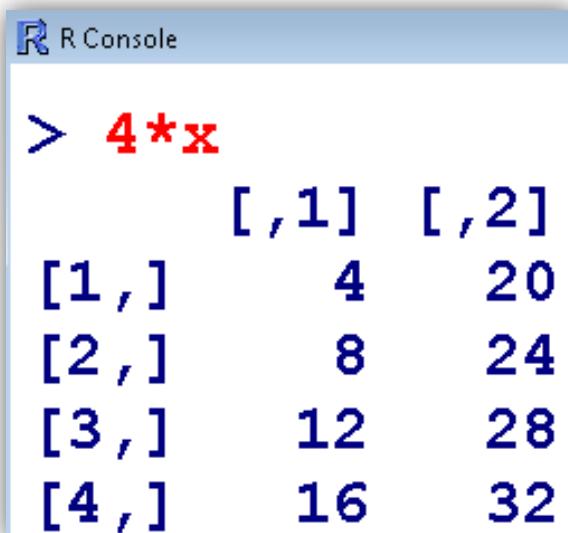
> x
     [,1]    [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

R R Console

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T)
> x
     [,1]    [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
>
```

- **Addition and subtraction of matrices (of same dimensions!) can be executed with the usual operators + and -**

```
> 4*x  
      [,1]    [,2]  
[1,]    4     8  
[2,]   12    16  
[3,]   20    24  
[4,]   28    32
```



A screenshot of the R console window titled "R Console". The window shows the command `> 4*x` and its output. The output is a 4x2 matrix with values 4, 8, 12, 16 in the first column and 20, 24, 28, 32 in the second column.

```
R Console  
> 4*x  
      [,1]    [,2]  
[1,]    4     8  
[2,]   12    16  
[3,]   20    24  
[4,]   28    32
```

- **Addition and subtraction of matrices (of same dimensions!) can be executed with the usual operators + and -**

```
> x + 4*x
      [,1]    [,2]
[1,]    5    10
[2,]   15    20
[3,]   25    30
[4,]   35    40
```

R Console

```
> x + 4*x
      [,1]    [,2]
[1,]    5    25
[2,]   10    30
[3,]   15    35
[4,]   20    40
```

```
> 4*x - x
      [,1]    [,2]
[1,]    3     6
[2,]    9    12
[3,]   15    18
[4,]   21    24
```

R Console

```
> 4*x - x
      [,1]    [,2]
[1,]    3    15
[2,]    6    18
[3,]    9    21
[4,]   12    24
```

- **Access to rows, columns or submatrices:**

```
> x <- matrix( nrow=5, ncol=3, byrow=T, data=1:15)
> x
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
[5,]   13   14   15
```

R Console

```
> x <- matrix( nrow=5, ncol=3, byrow=T, data=1:15)
> x
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
[5,]   13   14   15
```

- **Access to rows, columns or submatrices:**

```
> x[3,]  
[1] 7 8 9  
  
> x[,2]  
[1] 2 5 8 11 14  
  
> x[4:5, 2:3]  
     [,1] [,2]  
[1,]    11   12  
[2,]    14   15
```

R Console

```
> x[3,]  
[1] 7 8 9  
>  
> x[,2]  
[1] 2 5 8 11 14  
>  
> x[4:5, 2:3]  
     [,1] [,2]  
[1,]    11   12  
[2,]    14   15
```

- **Inverse of a matrix:**

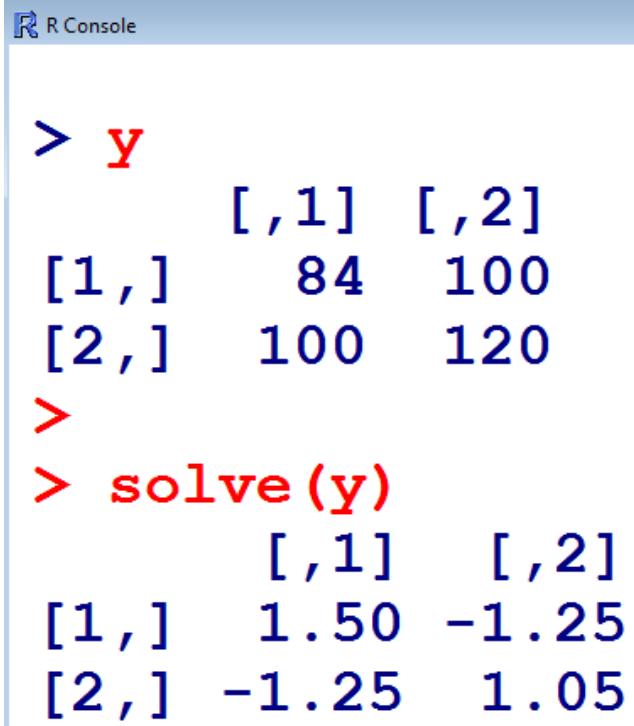
`solve()` finds the inverse of a positive definite matrix

Example:

```
> y<- matrix( nrow=2, ncol=2, byrow=T,  
data=c(84,100,100,120))
```

```
> y  
     [,1] [,2]  
[1,]   84   100  
[2,]   100   120
```

```
> solve(y)  
     [,1] [,2]  
[1,]  1.50 -1.25  
[2,] -1.25  1.05
```



R Console

```
> y  
     [,1] [,2]  
[1,]   84   100  
[2,]   100   120  
>  
> solve(y)  
     [,1] [,2]  
[1,]  1.50 -1.25  
[2,] -1.25  1.05
```

- **Eigen Values and Eigen Vectors:**

**eigen()** finds the eigen values and eigen vectors of a positive definite matrix

**Example:**

```
> y
      [,1] [,2]
[1,]    84   100
[2,]   100   120
```

```
> eigen(y)
$values
[1] 203.6070864    0.3929136
```

**\$vectors**

```
      [,1]          [,2]
[1,] 0.6414230 -0.7671874
[2,] 0.7671874  0.6414230
```

R Console

```
> y
      [,1] [,2]
[1,]    84   100
[2,]   100   120
> eigen(y)
$values
[1] 203.6070864    0.3929136

$vectors
      [,1]          [,2]
[1,] 0.6414230 -0.7671874
[2,] 0.7671874  0.6414230
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Matrix Operations**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

- **Multiplication of a matrix with a constant**

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
  
> x  
     [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8
```

R R Console

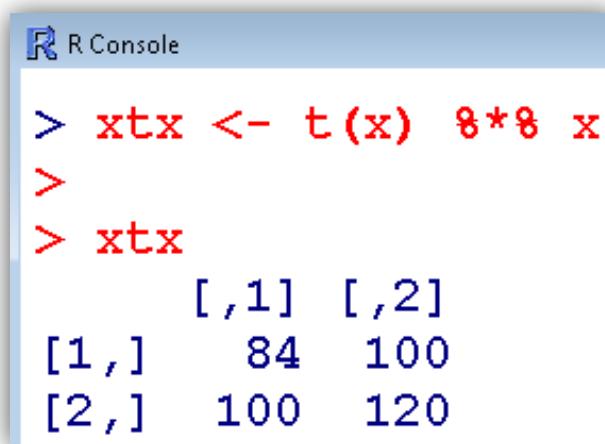
```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T )  
>  
> x  
     [,1]    [,2]  
[1,]    1    2  
[2,]    3    4  
[3,]    5    6  
[4,]    7    8  
> |
```

- **Matrix multiplication: operator %\*%**

Consider the multiplication of  $X'$  with  $X$

```
> xtx <- t(x) %*% x
```

```
> xtx
      [,1]    [,2]
[1,]    84    100
[2,]   100    120
```



A screenshot of an R console window titled "R Console". The window shows the following R code and its output:

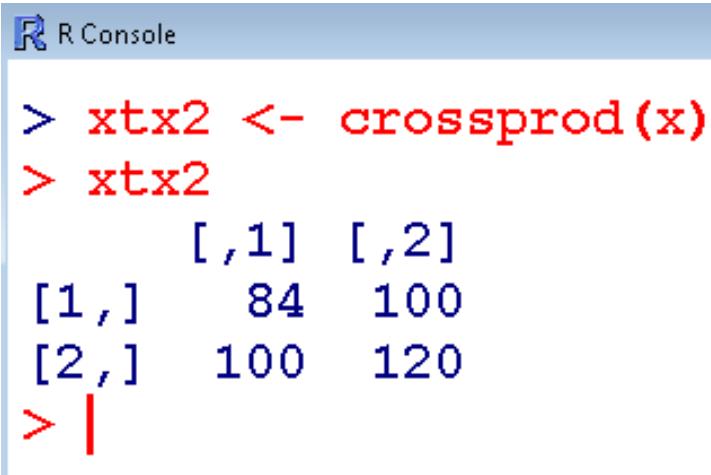
```
> xtx <- t(x) %*% x
>
> xtx
      [,1]    [,2]
[1,]    84    100
[2,]   100    120
```

- Cross product of a matrix  $X$ ,  $X'X$ , with a function `crossprod`

```
> xtx2 <- crossprod(x)
```

```
> xtx2
```

	[,1]	[,2]
[1,]	84	100
[2,]	100	120



R Console

```
> xtx2 <- crossprod(x)
> xtx2
      [,1] [,2]
[1,]    84   100
[2,]   100   120
> |
```

Note: Command `crossprod()` executes the multiplication faster than the conventional method with `t(x) %*% x`

- **Addition and subtraction of matrices (of same dimensions) can be executed with the usual operators + and -**

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T)

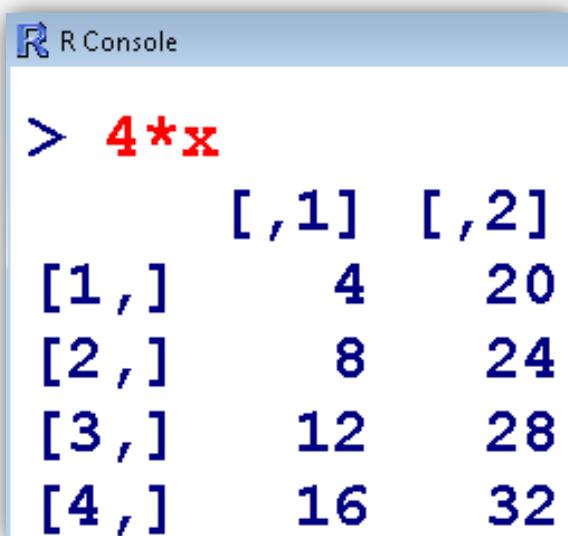
> x
     [,1]    [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

R R Console

```
> x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T)
> x
     [,1]    [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
>
```

- **Addition and subtraction of matrices (of same dimensions!) can be executed with the usual operators + and -**

```
> 4*x  
      [,1]    [,2]  
[1,]    4     8  
[2,]   12    16  
[3,]   20    24  
[4,]   28    32
```



A screenshot of the R console window titled "R Console". The console shows the command `> 4*x` and its output. The output is a 4x2 matrix where each element is the result of multiplying the corresponding element in the input matrix `x` by 4. The matrix has columns labeled `[,1]` and `[,2]`, and rows indexed from 1 to 4.

	[,1]	[,2]
[1,]	4	8
[2,]	12	16
[3,]	20	24
[4,]	28	32

- **Addition and subtraction of matrices (of same dimensions!) can be executed with the usual operators + and -**

```
> x + 4*x
      [,1]    [,2]
[1,]    5    10
[2,]   15    20
[3,]   25    30
[4,]   35    40
```

R Console

```
> x + 4*x
      [,1]    [,2]
[1,]    5    25
[2,]   10    30
[3,]   15    35
[4,]   20    40
```

```
> 4*x - x
      [,1]    [,2]
[1,]    3     6
[2,]    9    12
[3,]   15    18
[4,]   21    24
```

R Console

```
> 4*x - x
      [,1]    [,2]
[1,]    3    15
[2,]    6    18
[3,]    9    21
[4,]   12    24
```

- **Access to rows, columns or submatrices:**

```
> x <- matrix( nrow=5, ncol=3, byrow=T, data=1:15)
> x
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
[5,]   13   14   15
```

R Console

```
> x <- matrix( nrow=5, ncol=3, byrow=T, data=1:15)
> x
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
[5,]   13   14   15
```

- **Access to rows, columns or submatrices:**

```
> x[3,]  
[1] 7 8 9  
  
> x[,2]  
[1] 2 5 8 11 14  
  
> x[4:5, 2:3]  
     [,1] [,2]  
[1,]    11   12  
[2,]    14   15
```

R Console

```
> x[3,]  
[1] 7 8 9  
>  
> x[,2]  
[1] 2 5 8 11 14  
>  
> x[4:5, 2:3]  
     [,1] [,2]  
[1,]    11   12  
[2,]    14   15
```

- **Inverse of a matrix:**

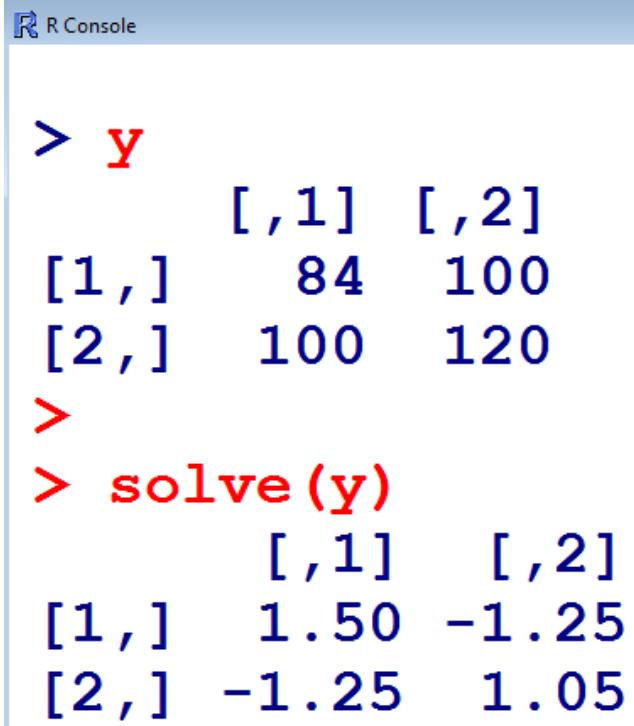
`solve()` finds the inverse of a positive definite matrix

Example:

```
> y<- matrix( nrow=2, ncol=2, byrow=T,  
data=c(84,100,100,120))
```

```
> y  
     [,1] [,2]  
[1,]   84   100  
[2,]   100   120
```

```
> solve(y)  
     [,1] [,2]  
[1,]  1.50 -1.25  
[2,] -1.25  1.05
```



R Console

```
> y  
     [,1] [,2]  
[1,]   84   100  
[2,]   100   120  
>  
> solve(y)  
     [,1] [,2]  
[1,]  1.50 -1.25  
[2,] -1.25  1.05
```

- **Eigen Values and Eigen Vectors:**

**eigen()** finds the eigen values and eigen vectors of a positive definite matrix

**Example:**

```
> y
      [,1] [,2]
[1,]    84   100
[2,]   100   120
```

```
> eigen(y)
$values
[1] 203.6070864    0.3929136
```

**\$vectors**

```
      [,1]          [,2]
[1,] 0.6414230 -0.7671874
[2,] 0.7671874  0.6414230
```

R Console

```
> y
      [,1] [,2]
[1,]    84   100
[2,]   100   120
> eigen(y)
$values
[1] 203.6070864    0.3929136

$vectors
      [,1]          [,2]
[1,] 0.6414230 -0.7671874
[2,] 0.7671874  0.6414230
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Missing Data and Logical Operators**

**Shalabh**

**Department of Mathematics and Statistics**

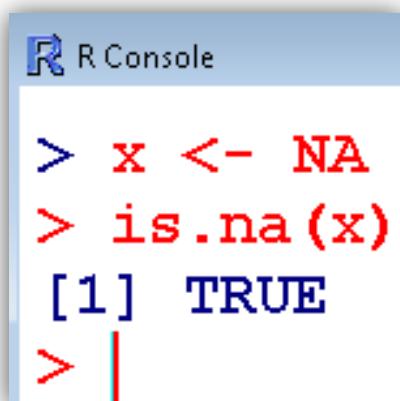
**Indian Institute of Technology Kanpur**

# Missing data

R represents missing observations through the data value **NA**

We can detect missing values using **is.na**

```
> x <- NA      # assign NA to variable x  
> is.na(x)    # is it missing?  
[1] TRUE
```

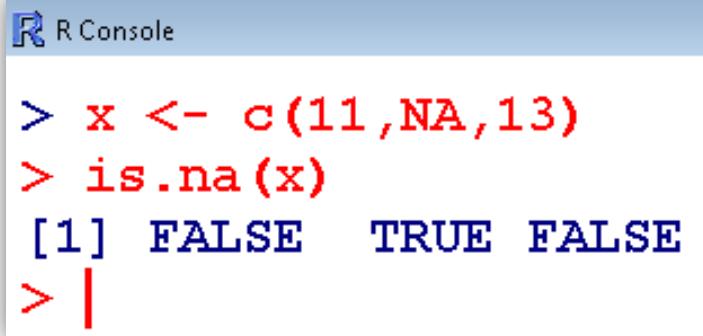
A screenshot of an R console window titled "R Console". The window shows the command "x <- NA" in red, indicating it was entered by the user. Below it, the command "is.na(x)" is also in red. The output "[1] TRUE" is displayed in blue, indicating the result of the function call. A cursor is visible at the bottom of the console window.

```
> x <- NA  
> is.na(x)  
[1] TRUE  
> |
```

# Missing data

Now try a vector to know if any value is missing?

```
> x <- c(11, NA, 13)  
  
> is.na(x)  
  
[1] FALSE TRUE FALSE
```



A screenshot of an R console window titled "R Console". The window shows the following R code and its output:

```
> x <- c(11,NA,13)
> is.na(x)
[1] FALSE TRUE FALSE
> |
```

## Example : How to work with missing data

```
> x <- c(11,NA,13) # vector  
  
> mean(x)      
$$\frac{11+NA+13}{2}$$
  
[1] NA  
  
> mean(x, na.rm = TRUE) # NAs can be removed  
[1] 12      
$$\frac{11+13}{2} = 12$$

```

The null object, called **NULL**, is returned by some functions and expressions.

Note that **NA** and **NULL** are not the same.

**NA** is a placeholder for something that exists but is missing.

**NULL** stands for something that never existed at all.

# Logical Operators and Comparisons

The following table shows the operations and functions for logical comparisons (True or False).

**TRUE** and **FALSE** are reserved words denoting logical constants.

Operator	Executions
>	Greater than
$\geq$	Greater than or equal
<	Less than
$\leq$	Less than or equal
$\equiv$	Exactly equal to
$\neq$	Not equal to
!	Negation (not)

# Logical Operators and Comparisons

Operator	Executions
&, &&	and
,	or

- The shorter form performs element-wise comparisons in almost the same way as arithmetic operators.
- The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.
- The longer form is appropriate for programming control-flow and typically preferred in if clauses (conditional).

# Logical Operators and Comparisons

TRUE and FALSE are reserved words denoting logical constants

Operator	Executions
<b>xor ()</b>	either... or (exclusive)
<b>isTRUE (x)</b>	test if <b>x</b> is TRUE
<b>TRUE</b>	true
<b>FALSE</b>	false

## Examples:

```
> 8 > 7
```

```
[1] TRUE
```

```
> 7 < 5
```

```
[1] FALSE
```

Is 8 less than 6?

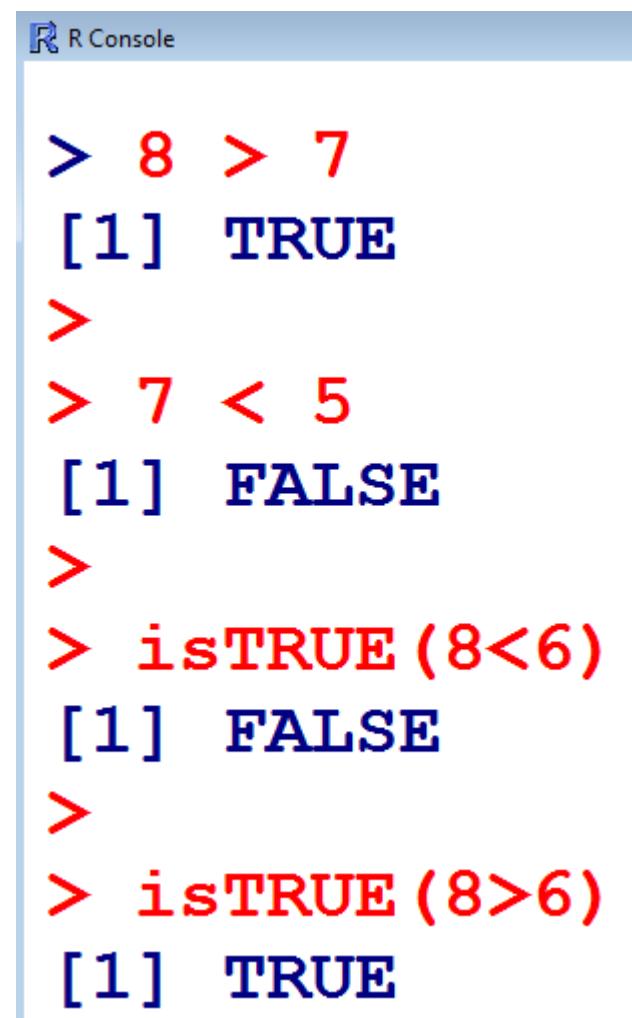
```
> isTRUE(8<6)
```

```
[1] FALSE
```

Is 8 greater than 6?

```
> isTRUE(8>6)
```

```
[1] TRUE
```



The screenshot shows an R console window titled "R Console". It displays several R commands and their outputs. The commands are colored red, while the outputs are in blue. The session starts with the command `> 8 > 7`, followed by the output [1] TRUE. Then, the command `> 7 < 5` is entered, resulting in [1] FALSE. Next, the question "Is 8 less than 6?" is asked, followed by the command `> isTRUE(8<6)` and its output [1] FALSE. Finally, the question "Is 8 greater than 6?" is asked, followed by the command `> isTRUE(8>6)` and its output [1] TRUE.

```
> 8 > 7
[1] TRUE
> 7 < 5
[1] FALSE
>
> isTRUE(8<6)
[1] FALSE
>
> isTRUE(8>6)
[1] TRUE
```

## Examples:

```
> x <- 5  
> (x < 10) && (x > 2)      # && means AND  
[1] TRUE
```

R R Console

```
> x <- 5  
> (x < 10) && (x > 2)  
[1] TRUE
```

## Examples:

```
> x <- 5
```

Is **x** less than 10 or **x** is greater than 5 ?

```
> (x < 10) || (x > 5)      # || means OR  
[1] TRUE
```

Is **x** greater than 10 or **x** is greater than 5 ?

```
> (x > 10) || (x > 5)  
[1] FALSE
```

R Gui (64-bit)

```
>  
> (x < 10) || (x > 5)  
[1] TRUE  
>  
> (x > 10) || (x > 5)  
[1] FALSE  
>
```

## Examples:

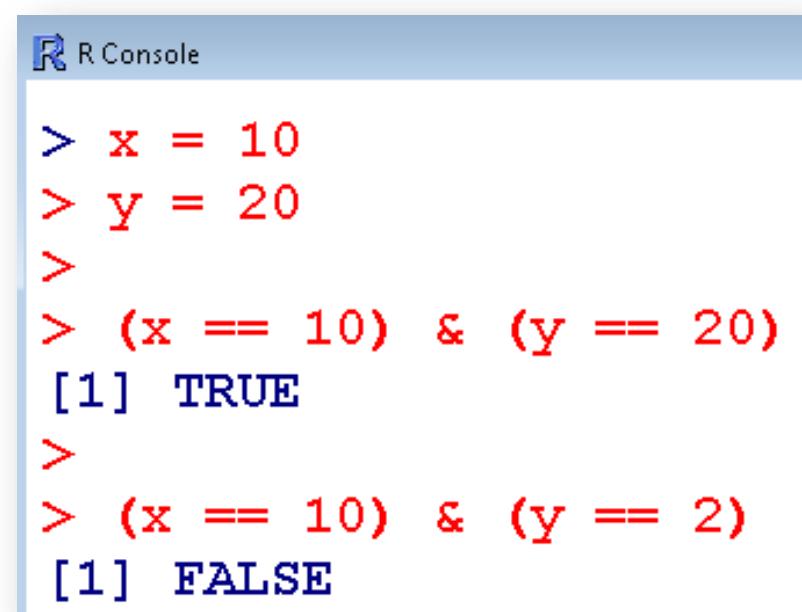
```
> x = 10  
> y = 20
```

Is **x** equal to 10 and is **y** equal to 20?

```
> (x == 10) & (y == 20)      # == means exactly  
                                equal to  
[1] TRUE
```

Is **x** equal to 10 and is **y** equal to 2?

```
> (x == 10) & (y == 2)  
[1] FALSE
```



The screenshot shows the R Console interface. The title bar says "R Console". The console window displays the following R session:

```
R Console  
> x = 10  
> y = 20  
>  
> (x == 10) & (y == 20)  
[1] TRUE  
>  
> (x == 10) & (y == 2)  
[1] FALSE
```

## Examples:

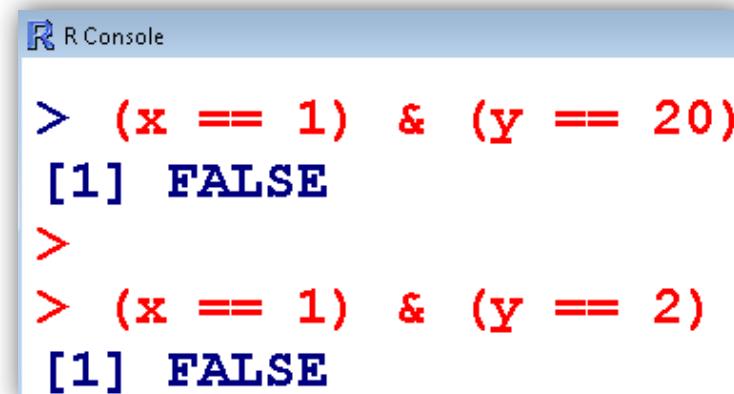
```
> x = 10  
> y = 20
```

Is **x** equal to 1 and is **y** equal to 20?

```
> (x == 1) & (y == 20)      # == means exactly  
                           equal to  
[1] FALSE
```

Is **x** equal to 1 and is **y** equal to 2?

```
> (x == 1) & (y == 2)  
[1] FALSE
```



R Console

```
> (x == 1) & (y == 20)  
[1] FALSE  
>  
> (x == 1) & (y == 2)  
[1] FALSE
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Logical Operators**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Logical Operators and Comparisons

The following table shows the operations and functions for logical comparisons (True or False).

Operator	Executions
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Exactly equal to
!=	Not equal to
!	Negation (not)
&, &&	and
,	or

Operator	Executions
xor()	either... or (exclusive)
isTRUE(x)	test if x is TRUE
TRUE	true
FALSE	false

## Examples:

```
> x = 1:6      # Generates x=1,2,3,4,5,6
> (x > 2) & (x < 5)  # Checks whether the
                        values are greater
                        than 2 and less than 2
[1] FALSE FALSE TRUE TRUE FALSE FALSE
> x[(x > 2) & (x < 5)] # Finds which values
                            are greater than 2 and
                            smaller than 5
[1] 3 4
```

## Examples:

R R Console

```
> x = 1:6
> x
[1] 1 2 3 4 5 6
>
> (x > 2) & (x < 5)
[1] FALSE FALSE TRUE TRUE FALSE FALSE
>
> x[(x > 2) & (x < 5)]
[1] 3 4
```

## Examples:

```
> x = 1:6      # Generates x=1,2,3,4,5,6  
> (x > 2) | (x < 5) # Checks whether the  
values are greater  
than 2 or less than 5
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
> x[(x > 2) | (x < 5)] # Finds which values  
are greater than 2 or  
smaller than 5
```

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

## Examples:

R Console

```
> x = 1:6
> (x > 2) | (x < 5)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
>
> x[(x > 2) | (x < 5)]
[1] 1 2 3 4 5 6
```

## Examples:

```
> x = 1:6      # Generates x=1,2,3,4,5,6  
> (x > 2) | (x > 10) # Checks whether the  
                           values are greater than  
                           2 or greater than 10  
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

R R Console

```
> x = 1:6  
> (x > 2) | (x > 10)  
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

## Examples:

```
> x = 1:6      # Generates x = 1,2,3,4,5,6  
> (x > 2) | (x > 10) # Checks whether the  
                           values are greater than  
                           2 or greater than 10  
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  
  
> x[(x > 2) | (x > 10)] # Finds which values  
                           are greater than 2 or  
                           smaller than 10  
[1] 3 4 5 6
```

## Examples:

R Console

```
> x = 1:6
> (x > 2) | (x > 10)
[1] FALSE FALSE TRUE TRUE TRUE TRUE
>
> x[(x > 2) | (x > 10)]
[1] 3 4 5 6
```

# Logical Operators and Comparisons

Operator	Executions
&, &&	and
,	or

- The shorter form performs element-wise comparisons in almost the same way as arithmetic operators.
- The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.

## Example of “The longer form evaluates left to right examining only the first element of each vector”

```
> x = 1:6      # Generates x = 1,2,3,4,5,6
```

```
> (x > 2) && (x < 5)  
[1] FALSE
```

is equivalent to:

```
> (x[1] > 2) & (x[1] < 5)  
[1] FALSE
```

Note that `x[1]` is only the first element in `x`

R Console

```
> x = 1:6  
> (x > 2) && (x < 5)  
[1] FALSE
```

R Console

```
> (x[1] > 2) & (x[1] < 5)  
[1] FALSE
```

**Example of “The longer form evaluates left to right examining only the first element of each vector”**

**(Contd...)**

```
> x[(x > 2) && (x < 5)] # Finds which values  
integer(0)                      are greater than 2 and  
                                         smaller than 5
```

R R Console

```
> x[(x > 2) && (x < 5)]  
integer(0)
```

This statement is equivalent to

```
> x[(x[1] > 2) & (x[1] < 5)]  
integer(0)
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Truth Table and Conditional Executions**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Example of Standard logical operations

## Truth table

Statement 1 :: (x)	Statement 2 :: (y)	Outcome :: x and y	Outcome :: x or y
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

## Example of Standard logical operations

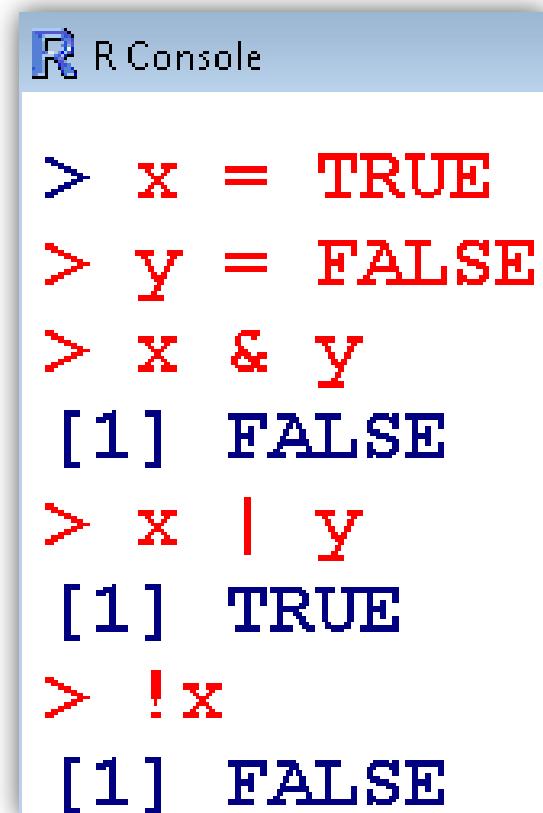
```
> x = TRUE
```

```
> y = FALSE
```

```
> x & y      # x AND y  
[1] FALSE
```

```
> x | y      # x OR y  
[1] TRUE
```

```
> !x         # negation of x  
[1] FALSE
```



A screenshot of the R Console window. The title bar says "R R Console". The console area contains the following R session:

```
> x = TRUE  
> y = FALSE  
> x & y  
[1] FALSE  
> x | y  
[1] TRUE  
> !x  
[1] FALSE
```

# Example

```
> x <- 5

> Logical1 <- (x > 2)

> is.logical(Logical1)

[1] TRUE

> Logical2 <- (x < 10)

> is.logical(Logical2)

[1] TRUE
```

```
R R Console

> x <- 5
> Logical1 <- (x > 2)
> is.logical(Logical1)
[1] TRUE
>
> Logical2 <- (x < 10)
> is.logical(Logical2)
[1] TRUE
```

## Example

```
> x <- 5  
  
> Logical3 <- (2*x > 11)  
  
> is.logical(Logical3)  
[1] TRUE
```

R Console

```
> x <- 5  
> Logical3 <- (2*x > 11)  
> is.logical(Logical3)  
[1] TRUE  
>  
> Logical4 <- (3*x <20 )  
> is.logical(Logical4)  
[1] TRUE
```

```
> Logical4 <- (3*x <20)  
  
> is.logical(Logical4)  
[1] TRUE
```

# **Control structures in R :**

**Control statements,**

**loops,**

**functions**

**Conditional execution**

# 1. Conditional execution

## Syntax

```
if (condition) {executes commands if condition is TRUE}  
if (condition) {executes commands if condition is TRUE}  
else { executes commands if condition is FALSE }
```

## Please note:

- The condition in this control statement may not be vector valued and if so, only the first element of the vector is used.
- The condition may be a complex expression where the logical operators "and" (`&&`) and "or" (`||`) can be used.

# 1. Conditional execution

## Example

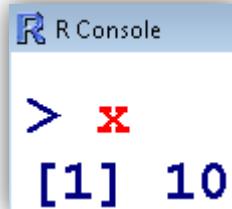
```
> x <- 5  
> if ( x==3 ) { x <- x-1 } else { x <- 2*x }
```

## Interpretation:

- If  $x = 3$ , then execute  $x = x - 1$ .
- If  $x \neq 3$ , then execute  $x = 2^*x$ .

In this case,  $x = 5$ , so  $x \neq 3$ . Thus  $x = 2^*5$

```
> x  
[1] 10
```



Now choose  $x = 3$  and repeat this example

```
R R Console  
> x <- 5  
> if ( x==3 ) { x <- x-1 } else { x <- 2*x }
```

# 1. Conditional execution

## Example

```
> x <- 3  
> if ( x==3 ) { x <- x-1 } else { x <- 2*x }
```

R R Console

```
> x <- 3  
> if ( x==3 ) { x <- x-1 } else { x <- 2*x }
```

## Interpretation:

- If  $x = 3$ , then execute  $x = x - 1$ .
- If  $x \neq 3$ , then execute  $x = 2*x$ .

In this case,  $x = 3$ , so  $x = 3 - 1$

```
> x  
[1] 2
```

R R Console

```
> x  
[1] 2
```

# **Introduction to R Software**

**Basics of Calculations**

::::

**Conditional Executions and Loops**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# 1. Conditional execution

## Syntax

```
if ( condition ) {executed commands if condition is  
                    TRUE }  
  
if ( condition ) {executed commands if condition is  
                    TRUE }  
  
else { executed commands if condition is FALSE }
```

## 2. Conditional execution

### Syntax

```
ifelse(test, yes, no)
```

- **Vector-valued evaluation of conditions .**
- **For the components in the vector-valued logical expression `test` which provide the value `TRUE`, the operations given by `yes` are executed.**
- **For the components in the vector-valued logical expression `test` which provide the value `FALSE`, the operations given by `no` are executed.**

## 2. Conditional execution

### Example

```
> x <- 1:10  
>x  
[1] 1 2 3 4 5 6 7 8 9 10  
> ifelse( x<6, x^2, x+1 )  
[1] 1 4 9 16 25 7 8 9 10 11
```

### Interpretation

- If  $x < 6$  (TRUE), then  $x = x^2$  (YES) .
  - If  $x \geq 6$  (FALSE), then  $x = x + 1$  (NO).
- 
- So for  $x = 1, 2, 3, 4, 5$ , we get  $x = x^2 = 1, 4, 9, 16, 25$
  - For  $x=6, 7, 8, 9, 10$ , we get  $x = x+1 = 7, 8, 9, 10, 11$

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
>
> ifelse( x<6, x^2, x+1 )
[1] 1 4 9 16 25 7 8 9 10 11
```

# **Control structures in R :**

## **Loops**

**Repetitive commands are executed by loops**

- **for loop**
- **while loop**
- **repeat loop**

# 1. The **for** loop

If the number of repetitions is known in advance (e.g. if all commands have to be executed for all cases  $i = 1, 2, \dots, n$  in the data), a **for()** loop can be used.

## Syntax

```
for (name in vector) {commands to be executed}
```

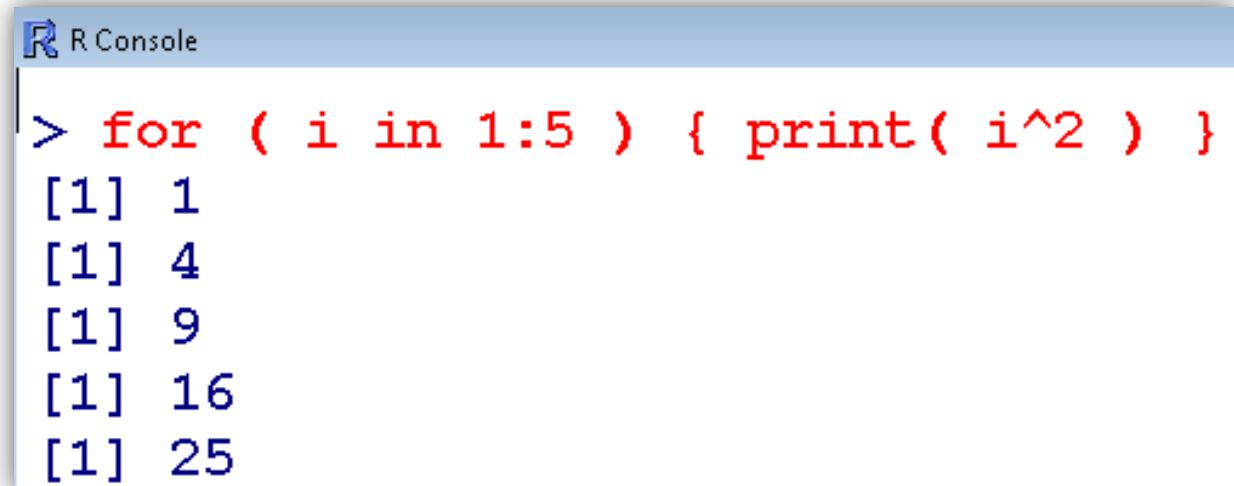
A variable with name **name** is sequentially set to all values, which contained in the vector **vector**.

All operations/commands are executed for all these values.

## Example

```
> for ( i in 1:5 ) { print( i^2 ) }
```

```
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25
```



The image shows a screenshot of an R console window. The title bar says "R Console". In the main area, there is a red border around the command and its output. The command is: `> for ( i in 1:5 ) { print( i^2 ) }`. The output is: [1] 1, [1] 4, [1] 9, [1] 16, [1] 25.

## Example

Note: `print` is a function to print the argument

```
> for ( i in c(2,4,6,7) ) { print( i^2 ) }
```

```
[1] 4
```

```
[1] 16
```

```
[1] 36
```

```
[1] 49
```

R Console

```
> for ( i in c(2,4,6,7) ) { print( i^2 ) }
```

```
[1] 4
```

```
[1] 16
```

```
[1] 36
```

```
[1] 49
```

## 2. The `while()` loop

If the number of loops is not known in before, e.g. when an iterative algorithm to maximize a likelihood function is used, one can use a `while()` loop.

### Syntax

```
while(condition) { commands to be executed as long as condition is TRUE }
```

If the condition is not true *before entering* the loop, no commands within the loop are executed.

## Example

```
> i <- 1  
  
> while (i<5) {  
+ print(i^2)  
+ i <- i+2  
  
+}  
  
[1] 1  
  
[1] 9
```

The programmer itself has to be careful that the counting variable **i** within the loop is incremented. Otherwise an infinite loop occurs.

R R Console

```
> i <- 1
> while (i<5) {
+ print(i^2)
+ i <- i+2
+
[1] 1
[1] 9
```

### 3. The **repeat** loop

The repeat loop doesn't test any condition — in contrast to the **while ()** loop — *before entering* the loop and also not during the execution of the loop.

Again, the programmer is responsible that the loop terminates after the appropriate number of iterations. For this the **break** command can be used.

#### Syntax

**repeat{ commands to be executed }**

## Example:

```
> i <- 1

> repeat{
+ print( i^2 )
+ i <- i+2
+ if ( i > 10 ) break
+}
[1] 1
[1] 9
[1] 25
[1] 49
[1] 81
```

## R R Console

```
> i <- 1
> repeat{
+ print( i^2 )
+ i <- i+2
+ if ( i>10 ) break
+
[1] 1
[1] 9
[1] 25
[1] 49
[1] 81
```

## Example:

Additionally, the command `next` is available, to return to the beginning of the loop (to return to the first command in the loop).

```
> i <- 1

> repeat{
+ i <- i+1
+ if (i < 10) next
+ print(i^2)
+ if (i >= 13) break
+}
[1] 100
[1] 121
[1] 144
[1] 169
```

# **Introduction to R Software**

## **Basics of Calculations**

::::

## **Loops**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# **Control structures in R :**

## **Loops**

**Repetitive commands are executed by loops**

- **for loop**
- **while loop**
- **repeat loop**

# 1. The **for** loop

If the number of repetitions is known in advance then a **for()** loop can be used.

## Syntax

```
for (name in vector) {commands to be executed}
```

All operations/commands are executed for all these values.

## 2. The `while()` loop

If the number of loops is not known in before, e.g. when an iterative algorithm to maximize a likelihood function is used, one can use a `while()` loop.

### Syntax

```
while(condition) { commands to be executed as long as condition is TRUE }
```

If the condition is not true *before entering* the loop, no commands within the loop are executed.

## Example

```
> i <- 1  
  
> while (i<5) {  
+ print(i^2)  
+ i <- i+2  
  
+}  
  
[1] 1  
  
[1] 9
```

The programmer itself has to be careful that the counting variable **i** within the loop is incremented. Otherwise an infinite loop occurs.



R Console

```
> i <- 1
> while (i<5) {
+ print(i^2)
+ i <- i+2
+
[1] 1
[1] 9
```

### 3. The **repeat** loop

The repeat loop doesn't test any condition — in contrast to the **while ()** loop — *before entering* the loop and also not during the execution of the loop.

Again, the programmer is responsible that the loop terminates after the appropriate number of iterations. For this the **break** command can be used.

#### Syntax

**repeat{ commands to be executed }**

## Example:

```
> i <- 1

> repeat{
+ print( i^2 )
+ i <- i+2
+ if ( i > 10 ) break
+}
[1] 1
[1] 9
[1] 25
[1] 49
[1] 81
```

## R R Console

```
> i <- 1
> repeat{
+ print( i^2 )
+ i <- i+2
+ if ( i>10 ) break
+
[1] 1
[1] 9
[1] 25
[1] 49
[1] 81
```

## Example:

Additionally, the command `next` is available, to return to the beginning of the loop (to return to the first command in the loop).

```
> i <- 1

> repeat{
+ i <- i+1
+ if (i < 10) next
+ print(i^2)
+ if (i >= 13) break
+}
[1] 100
[1] 121
[1] 144
[1] 169
```

# **Introduction to R Software**

## **Sequences**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Sequences

A sequence is a set of related numbers, events, movements, or items that follow each other in a particular order.

The regular sequences can be generated in R.

## Syntax

`seq()`

```
seq(from = 1, to = 1, by = ((to -  
from) / (length.out - 1)), length.out = NULL,  
along.with = NULL, ...)
```

Help for `seq`

```
> help("seq")
```

## seq {base}

### Sequence Generation

#### Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

#### Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

#### Arguments

... arguments passed to or from methods.

from, to the starting and (maximal) end values of the sequence. Of length 1 unless just `from` is supplied as an unnamed argument.



# Sequences

- The default increment is +1 or -1

```
> seq(from=2, to=4)
```

```
[1] 2 3 4
```

```
> seq(from=4, to=2)
```

```
[1] 4 3 2
```

```
> seq(from=-4, to=4)
```

```
[1] -4 -3 -2 -1 0 1 2 3 4
```

## Sequences

R Console

```
> seq(from=2, to=4)
[1] 2 3 4
>
> seq(from=4, to=2)
[1] 4 3 2
>
> seq(from=-4, to=4)
[1] -4 -3 -2 -1 0 1 2 3 4
```

# Sequences

## □ Sequence with constant increment:

Generate a sequence from 10 to 20 with an increment of 2 units

```
> seq(from=10, to=20, by=2)
```

```
[1] 10 12 14 16 18 20
```

R Console

```
> seq(from=10, to=20, by=2)
[1] 10 12 14 16 18 20
```

# Sequences

## □ Sequence with constant increment:

Generate a sequence from 20 to 10 with an increment of 2 units

```
> seq(from=20, to=10, by=-2)  
[1] 20 18 16 14 12 10
```

R R Console

```
> seq(from=20, to=10, by=-2)  
[1] 20 18 16 14 12 10
```

# Sequences

## □ Sequence with constant increment:

Generate a sequence from 20 to 10 with a decrement of 2 units

```
> seq(from=20, to=10, by=-2)  
[1] 20 18 16 14 12 10
```

R R Console

```
> seq(from=20, to=10, by=-2)  
[1] 20 18 16 14 12 10
```

## Sequences

### □ Downstream sequence with constant increment:

Generate a sequence from 3 to -2 with a decrement of 0.5 units

```
> seq(from=3, to=-2, by=-0.5)
[1] 3.0 2.5 2.0 1.5 1.0 0.5 0.0 -0.5 -1.0 -1.5 -2
```

R Console

```
> seq(from=3, to=-2, by=-0.5)
[1] 3.0 2.5 2.0 1.5 1.0 0.5 0.0 -0.5 -1.0 -1.5 -2.0
```

## Sequences

- Sequences with a predefined length with default increment +1

```
> seq(to=10, length=10)  
[1] 1 2 3 4 5 6 7 8 9 10
```

R R Console

```
> seq(to=10, length=10)  
[1] 1 2 3 4 5 6 7 8 9 10
```

## Sequences

- Sequences with a predefined length with default increment +1

```
> seq(from=10, length=10)
[1] 10 11 12 13 14 15 16 17 18 19
```

R R Console

```
> seq(from=10, length=10)
[1] 10 11 12 13 14 15 16 17 18 19
```

## Sequences

- Sequences with a predefined length with constant fractional increment

```
> seq(from=10, length=10, by=0.1)
```

```
[1] 10.0 10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9
```

R Console

```
> seq(from=10, length=10, by=0.1)
```

```
[1] 10.0 10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9
```

## Sequences

- Sequences with a predefined length with constant decrement

```
> seq(from=10, length=10, by=-2)  
[1] 10  8  6  4  2  0 -2 -4 -6 -8
```

## Sequences

- Sequences with a predefined length with constant fractional decrement

```
> seq(from=10, length=5, by=-.2)  
[1] 10.0 9.8 9.6 9.4 9.2
```

# Sequences

Sequences with a predefined variable and constant increment

```
> x<-2  
  
> seq(1, x, x/10)  
[1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x<-50  
  
> seq(0, x, x/10)  
[1] 0 5 10 15 20 25 30 35 40 45 50
```

## Sequences

R Console

```
> x<-2
> x
[1] 2
> seq(1, x, x/10)
[1] 1.0 1.2 1.4 1.6 1.8 2.0
>
> x<-50
> x
[1] 50
> seq(1, x, x/10)
[1] 1 6 11 16 21 26 31 36 41 46
```

# **Introduction to R Software**

## **Sequences**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Sequences

The regular sequences can be generated in R.

## Syntax

`seq()`

```
seq(from = 1, to = 1, by = ((to -  
from) / (length.out - 1)), length.out = NULL,  
along.with = NULL, ...)
```

## Sequences

```
> seq(10)
```

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

is the same as

```
> seq(1:10)
```

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

```
R Console
> seq(10)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(1:10)
[1] 1 2 3 4 5 6 7 8 9 10
```

# Sequences

## □ Assignment of an index-vector

```
> x <- c(9,8,7,6)  
  
> ind <- seq(along=x)  
  
> ind  
  
[1] 1 2 3 4
```

R Console

```
> x <- c(9,8,7,6)  
> ind <- seq(along=x)  
> ind  
[1] 1 2 3 4
```

Accessing a value in the vector through index vector

## □ Accessing an element of an index-vector

```
> x[ ind[2] ]  
  
[1] 8
```

R Console

```
> x[ ind[2] ]  
[1] 8
```

# **Sequences**

## **Generating sequences of dates**

### **Generating current time and date**

**Sys.time()** command provides the current time and date from the computer system.

```
> Sys.time()
```

```
[1] "2017-01-01 09:17:01 IST"
```

**Sys.Date()** command provides the current date from the computer system.

```
> Sys.Date()
```

```
[1] "2017-01-01"
```

## Sequences

R Console

```
> Sys.time()
[1] "2017-01-01 09:17:01 IST"
>
> Sys.Date()
[1] "2017-01-01"
```

# Sequences

## Generating sequences of dates

### Usage

```
seq(from, to, by, length.out = NULL, along.with  
= NULL, ...)
```

### Arguments

**from** starting date (Required)

**to** end date (Optional)

**by** increment of the sequence. "day", "week",  
"month", "quarter" or "year".

**length.out** integer, optional. Desired length of the sequence.

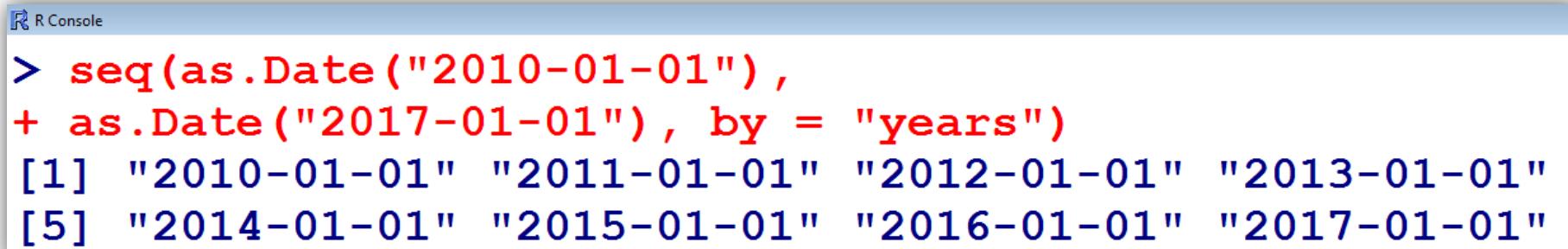
**along.with** take the length from the length of this argument. <sup>7</sup>

# Sequences

## Generating sequences of dates

### Sequence of first day of years

```
> seq(as.Date("2010-01-01"), as.Date("2017-01-01"),
+      by = "years")
[1] "2010-01-01" "2011-01-01" "2012-01-01" "2013-01-01"
[5] "2014-01-01" "2015-01-01" "2016-01-01" "2017-01-01"
```



R Console

```
> seq(as.Date("2010-01-01"),
+      as.Date("2017-01-01"), by = "years")
[1] "2010-01-01" "2011-01-01" "2012-01-01" "2013-01-01"
[5] "2014-01-01" "2015-01-01" "2016-01-01" "2017-01-01"
```

# Sequences

## Generating sequences of dates

### Sequence of days

```
> seq(as.Date("2017-01-01"), by = "days",  
length = 6)  
[1] "2017-01-01" "2017-01-02" "2017-01-03" "2017-01-04"  
[5] "2017-01-05" "2017-01-06"
```

R Console

```
> seq(as.Date("2017-01-01"), by = "days", length = 6)  
[1] "2017-01-01" "2017-01-02" "2017-01-03" "2017-01-04"  
[5] "2017-01-05" "2017-01-06"
```

# Sequences

## Generating sequences of dates

### Sequence of months

```
> seq(as.Date("2017-01-01"), by = "months",
length = 6)
[1] "2017-01-01" "2017-02-01" "2017-03-01" "2017-04-01"
[5] "2017-05-01" "2017-06-01"
```

R Console

```
> seq(as.Date("2017-01-01"), by = "months", length = 6)
[1] "2017-01-01" "2017-02-01" "2017-03-01" "2017-04-01"
[5] "2017-05-01" "2017-06-01"
```

# Sequences

## Generating sequences of dates

### Sequence by years

```
> seq(as.Date("2017-01-01"), by = "years",  
length = 6)  
[1] "2017-01-01" "2018-01-01" "2019-01-01" "2020-01-01"  
[5] "2021-01-01" "2022-01-01"
```

R Console

```
> seq(as.Date("2017-01-01"), by = "years", length = 6)  
[1] "2017-01-01" "2018-01-01" "2019-01-01" "2020-01-01"  
[5] "2021-01-01" "2022-01-01"
```

# Sequences

## Generating sequences of dates

To find sequence with defining start and end dates

```
> startdate <- as.Date("2016-1-1")
> enddate <- as.Date("2017-1-1")

> out <- seq(enddate, startdate, by = "-1
month")
[1] "2017-01-01" "2016-12-01" "2016-11-01" "2016-10-01"
[5] "2016-09-01" "2016-08-01" "2016-07-01" "2016-06-01"
[9] "2016-05-01" "2016-04-01" "2016-03-01" "2016-02-01"
[13] "2016-01-01"
```

# Sequences

## Generating sequences of dates

```
R Console
> startdate <- as.Date("2016-1-1")
> enddate <- as.Date("2017-1-1")
> out <- seq(enddate, startdate, by = "-1 month")
> out
[1] "2017-01-01" "2016-12-01" "2016-11-01" "2016-10-01"
[5] "2016-09-01" "2016-08-01" "2016-07-01" "2016-06-01"
[9] "2016-05-01" "2016-04-01" "2016-03-01" "2016-02-01"
[13] "2016-01-01"
```

# Sequences

## Generating sequences of letters

**letters** is used to find sequence of lowercase alphabets

```
> letters
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"  
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

R Console

```
> letters
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"  
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

# Sequences

## Generating sequences of letters

`letters[from_index:to_index]` is used to find sequence of lowercase alphabets from a particular index to a specified index.

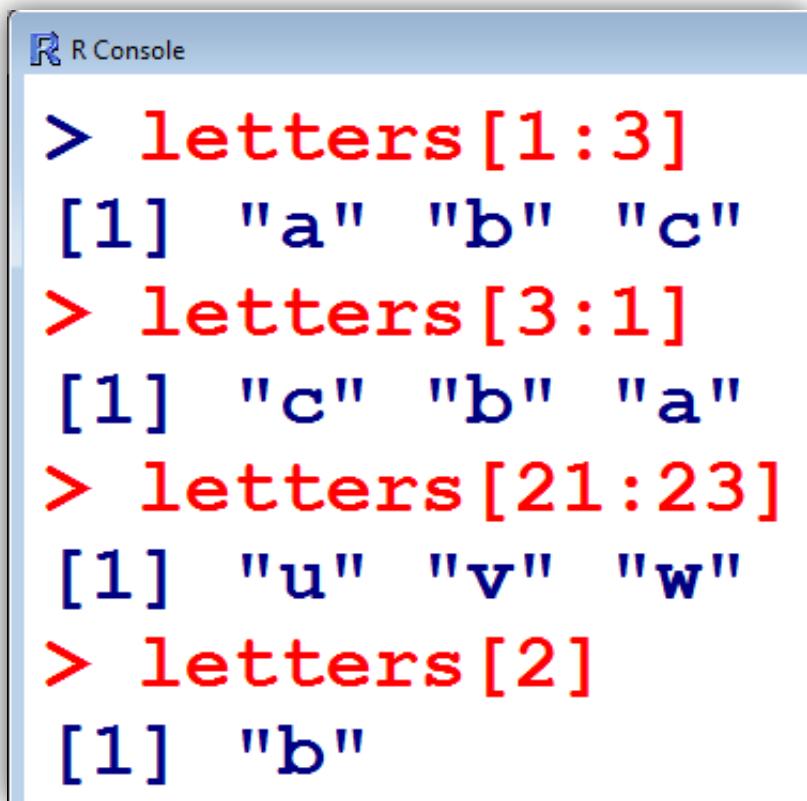
```
> letters[1:3]  
[1] "a" "b" "c"
```

```
> letters[3:1]  
[1] "c" "b" "a"
```

```
> letters[21:23]  
[1] "u" "v" "w"
```

```
> letters[2]  
[1] "b"
```

# Sequences



R Console

```
> letters[1:3]
[1] "a" "b" "c"
> letters[3:1]
[1] "c" "b" "a"
> letters[21:23]
[1] "u" "v" "w"
> letters[2]
[1] "b"
```

# Sequences

## Generating sequences of alphabets

**LETTERS** is used to find sequence of uppercase alphabets

> LETTERS

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"  
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

R Console

> LETTERS

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"  
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

# Sequences

## Generating sequences of alphabets

`LETTERS[from_index:to_index]` is used to find sequence of uppercase alphabets from a particular index to a specified index.

```
> LETTERS[1:3]  
[1] "A" "B" "C"
```

```
> LETTERS[3:1]  
[1] "C" "B" "A"
```

```
> LETTERS[21:23]  
[1] "U" "V" "W"
```

```
> LETTERS[2]  
[1] "B"
```

# Sequences

```
R R Console
> LETTERS[1:3]
[1] "A" "B" "C"
> LETTERS[3:1]
[1] "C" "B" "A"
> LETTERS[21:23]
[1] "U" "V" "W"
> LETTERS[2]
[1] "B"
```

# **Introduction to R Software**

## **Repeats**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Repeats

Command `rep` is used to replicates the values in a vector.

Syntax `rep(x)` replicates the values in a vector `x`.

`rep(x, times=n)` # Repeat `x` as a whole `n` times

`rep(x, each=n)` # Repeat each cell `n` times

# Repeats

## Help for the command `rep`

```
> help("rep")
```

The screenshot shows a web browser displaying the R documentation for the `rep` function. The URL in the address bar is `127.0.0.1:13069/library/base/html/rep.html`. The page title is "R Documentation". The main content area has a header "Replicate Elements of Vectors and Lists". Below it, under "Description", is the text: "rep replicates the values in `x`. It is a generic function, and the (internal) default method is described here." Under "Usage", there are three examples: `rep(x, ...)`, `rep.int(x, times)`, and `rep_len(x, length.out)`. Under "Arguments", there are two entries: "`x` a vector (of any mode including a list) or a factor or (for `rep` only) a `POSIXct` or `POSIXlt` or `Date` object; or an S4 object containing such an object." and "`...` further arguments to be passed to or from other methods. For the internal default method these can include: `times`".

rep {base}

R Documentation

Replicate Elements of Vectors and Lists

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

`rep.int` and `rep_len` are faster simplified versions for two common cases. They are not generic.

Usage

```
rep(x, ...)  
rep.int(x, times)  
rep_len(x, length.out)
```

Arguments

- `x` a vector (of any mode including a list) or a factor or (for `rep` only) a `POSIXct` or `POSIXlt` or `Date` object; or an S4 object containing such an object.
- `...` further arguments to be passed to or from other methods. For the internal default method these can include:
  - `times`

# Repeats

The command `rep`

Repeat an object  $n$ -times:

```
> rep(3.5, times=10)
```

```
[1] 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5
```

```
> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4
```

R Console

```
> rep(3.5, times=10)
[1] 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5
>
> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4
```

# Repeats

Repeat an object  $n$ -times:

```
rep(x, times = n )
```

Repeat each cell  $n$ -times:

```
rep(x, each = n )
```

```
> x <- 1:3
```

```
> x
```

```
[1] 1 2 3
```

```
> rep(x, times = 3 )
```

```
[1] 1 2 3 1 2 3 1 2 3
```

```
> rep(x, each = 3 )
```

```
[1] 1 1 1 2 2 2 3 3 3
```

# Repeats

Every object is repeated several times successively:

```
> rep(1:4, each = 2)
```

```
[1] 1 1 2 2 3 3 4 4
```

```
> rep(1:4, each = 2, times = 3)
```

```
[1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

R R Console

```
> rep(1:4, each = 2)
```

```
[1] 1 1 2 2 3 3 4 4
```

```
>
```

```
> rep(1:4, each = 2, times = 3)
```

```
[1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

# Repeats

Every object is repeated a different number of times:

```
> rep(1:4, 2:5)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4
```

R R Console

```
> rep(1:4, 2:5)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4
```

# Repeats

Every object is repeated a different number of times:

```
> ans <- seq(from=2, to=8, by=2)
> ans
[1] 2 4 6 8

> rep(1:4, ans)
[1] 1 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4
```

# Repeats

R R Console

```
> rep(1:4, 2:5)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4
>
> ans <- seq(from=2, to=8, by=2)
> ans
[1] 2 4 6 8
>
> rep(1:4, ans)
[1] 1 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4
```

# Repeats

```
> x <- matrix(nrow=2, ncol=2, data=1:4, byrow=T)  
  
> x  
      [,1] [,2]  
[1,]   1    2  
[2,]   3    4  
  
> rep(x, 3)  
[1] 1 3 2 4 1 3 2 4 1 3 2 4
```

# Repeats

```
R R Console

> x <- matrix(nrow=2, ncol=2, data=1:4, byrow=T)
> x
     [,1] [,2]
[1,]    1    2
[2,]    3    4
>
> rep(x, 3)
 [1] 1 3 2 4 1 3 2 4 1 3 2 4
> |
```

# Repeats

## Repetition of characters

```
> rep(c("a", "b", "c"), 2)
[1] "a" "b" "c" "a" "b" "c"
```

```
> rep(c("apple", "banana", "cake"), 2)
[1] "apple"   "banana"  "cake"    "apple"   "banana"  "cake"
```

# Repeats

```
R R Console
> rep(c("a", "b", "c"), 2)
[1] "a" "b" "c" "a" "b" "c"
>
> rep(c("apple", "banana", "cake"), 2)
[1] "apple"  "banana" "cake"    "apple"  "banana" "cake"
```

# **Introduction to R Software**

## **Sorting and Ordering**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**



# Sorting

**sort** function sorts the values of a vector in ascending order (by default) or descending order.

## Syntax

**sort(x, decreasing = FALSE, ...)**

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

**x**

Vector of values to be sorted

**decreasing**

Should the sort be increasing or decreasing

**na.last**

for controlling the treatment of NAs.

If TRUE, missing values in the data are put last;  
if FALSE, they are put first;  
if NA, they are removed.

# Sorting

## Example

```
> y <- c(8,5,7,6)
```

```
> y
```

```
[1] 8 5 7 6
```

```
> sort(y)
```

```
[1] 5 6 7 8
```

```
> sort(y, decreasing = TRUE)
```

```
[1] 8 7 6 5
```

# Sorting

```
R Console
> y <- c(8,5,7,6)
> y
[1] 8 5 7 6
> sort(y)
[1] 5 6 7 8
> sort(y, decreasing = TRUE)
[1] 8 7 6 5
```

# Ordering

**order** function sorts a variable according to the order of variable.

## Syntax

```
order(x, decreasing = FALSE, ...)
```

```
order(x, decreasing = FALSE, na.last = TRUE, ...)
```

**x**

Vector of values to be sorted

**decreasing**

Should the sort be increasing or decreasing

**na.last**

for controlling the treatment of NAs.

If TRUE, missing values in the data are put last;  
if FALSE, they are put first;  
if NA, they are removed.

# Ordering

## Example

```
> y <- c(8,5,7,6)
```

```
> y
```

```
[1] 8 5 7 6
```

```
> order(y)
```

```
[1] 2 4 3 1
```

```
> order(y, decreasing = TRUE)
```

```
[1] 1 3 4 2
```

# Ordering

```
R Console
> y <- c(8,5,7,6)
> y
[1] 8 5 7 6
> order(y)
[1] 2 4 3 1
>
> order(y,decreasing = TRUE)
[1] 1 3 4 2
```

# **Introduction to R Software**

## **Lists**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**



# **Lists**

**Vectors, matrices, and arrays** is that each of these types of objects may only contain one type of data.

For example, a vector may contain all numeric data or all character data.

A list is a special type of object that can contain data of multiple types.

Lists are characterized by the fact that their elements do not need to be of the same object type.

# Lists

- ❖ Lists can contain elements of different types so that the list elements may have different modes.
- ❖ Lists can even contain other structured objects, such as lists and data frames which allows to create recursive data structures.
- ❖ Lists can be indexed by position.  
So `x[[5]]` refers to the fifth element of `x`.

# Lists

- ❖ Lists can extract sublists.

So `x[c(2,5)]` is a sublist of `x` that consists of the second and fifth elements.

- ❖ List elements can have names.

Both `x[["Students"]]` and `x$Students` refer to the element named "Students".

- ❖ Difference between a vector and a list :

- In a vector, all elements must have the same mode.
- In a list, the elements can have different modes.

# **Lists**

## **Mode:**

**Every object has a mode.**

**The mode indicates how the object is stored in memory: as a**

- ✓ **number,**
- ✓ **character string,**
- ✓ **list of pointers to other objects,**
- ✓ **function etc.**

# Lists

## Mode:

Object	Example	Mode
Number	<code>1.234</code>	numeric
Vector of numbers	<code>c(5, 6, 7, 8)</code>	numeric
Character string	<code>"India"</code>	character
Vector of character strings	<code>c("India", "USA")</code>	character
Factor	<code>factor(c("UP", "MP"))</code>	numeric
List	<code>list("India", "USA")</code>	list
Data frame	<code>data.frame(x=1:2, y=c("India", "USA"))</code>	list
Function	<code>print</code>	function

# Lists

## Mode:

`mode` function gives us such information.

## Syntax

`mode ()`

# Lists

## Mode:

### Example

```
> mode(1.234)
```

```
[1] "numeric"
```

```
> mode(c(5,6,7,8))
```

```
[1] "numeric"
```

```
> mode("India")
```

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

```
> mode(c("India", "USA"))
```

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

# Lists

## Mode:

```
R R Console
> mode(1.234)
[1] "numeric"
> mode(c(5,6,7,8))
[1] "numeric"
> mode("India")
[1] "character"
> mode(c("India", "USA"))
[1] "character"
> mode(factor(c("UP", "MP")))
[1] "numeric"
> mode(list("India", "USA"))
[1] "list"
```

# Lists

Mode:

Example

```
> mode(factor(c("UP", "MP")))
[1] "numeric"

> mode(list("India", "USA"))
[1] "list"

> mode(data.frame(x=1:2, y=c("India", "USA")))
[1] "list"

> mode(print)
[1] "function"
```

# Lists

```
R R Console
> mode(factor(c("UP", "MP")))
[1] "numeric"
> mode(list("India", "USA"))
[1] "list"
> mode(data.frame(x=1:2, y=c("India", "USA")))
[1] "list"
> mode(print)
[1] "function"
```

# **Introduction to R Software**

## **Lists**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Lists

## Example

```
> x1 <- matrix(nrow=2, ncol=2, data=1:4, byrow=T)
> x2 <- matrix(nrow=2, ncol=2, data=5:8, byrow=T)
> x1
      [,1] [,2]
[1,]    1    2
[2,]    3    4

> x2
      [,1] [,2]
[1,]    5    6
[2,]    7    8

> x1+x2
      [,1] [,2]
[1,]    6    8
[2,]   10   12
```

# Lists

R R Console

```
> x1 <- matrix(nrow=2, ncol=2, data=1:4, byrow=T
> x2 <- matrix(nrow=2, ncol=2, data=5:8, byrow=T
> x1
      [,1] [,2]
[1,]     1     2
[2,]     3     4
> x2
      [,1] [,2]
[1,]     5     6
[2,]     7     8
> x1+x2
      [,1] [,2]
[1,]     6     8
[2,]    10    12
```

# Lists

## Example

```
> x1[2,1] <- "hello"  
  
> x1  
      [,1]      [,2]  
[1,]    "1"      "2"  
[2,]    "hello"   "4"
```

```
> x1 + x2  
Error in x1 + x2 : non-numeric argument to  
binary operator
```

# Lists

## Example

```
R Console
> x2 <- matrix(nrow=2, ncol=2, data=5:8, byrow=T)
> x1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> x1[2,1] <- "hello"
> x1
     [,1]      [,2]
[1,] "1"        "2"
[2,] "hello"    "4"
> x1+x2
Error in x1 + x2 : non-numeric argument to
binary operator
```

# Lists

Lists can contain any kind of objects as well as objects of different types. For example, lists can contain matrices as objects:

## Example

```
> x1 <- matrix(nrow=2, ncol=2, data=1:4, byrow=T)
> x2 <- matrix(nrow=2, ncol=2, data=5:8, byrow=T)

> x1
      [,1] [,2]
[1,]    1    2
[2,]    3    4

> x2
      [,1] [,2]
[1,]    5    6
[2,]    7    8
```

# Lists

## Example

```
> matlist <- list(x1, x2)
```

```
> matlist
```

```
[[1]]  
      [,1] [,2]  
[1,]    1    2  
[2,]    3    4  
  
[[2]]  
      [,1] [,2]  
[1,]    5    6  
[2,]    7    8
```

R Console

```
> matlist <- list(x1, x2)  
> matlist  
[[1]]  
      [,1] [,2]  
[1,]    1    2  
[2,]    3    4  
  
[[2]]  
      [,1] [,2]  
[1,]    5    6  
[2,]    7    8
```

# Lists

## Example

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

> matlist[2]
[[1]]
  [,1] [,2]
[1,]   5   6
[2,]   7   8
```

The screenshot shows the R Console window with the title "R Console". It displays two matrices, matlist[1] and matlist[2], as lists of matrices. Each matrix is a 2x2 array.

```
R Console
> matlist[1]
[[1]]
  [,1] [,2]
[1,]   1   2
[2,]   3   4

> matlist[2]
[[1]]
  [,1] [,2]
[1,]   5   6
[2,]   7   8
```

# Lists

An example of a list that contains different object types:

```
> z1 <- list( c("water", "juice", "lemonade") ,  
rep(1:4, each=2) , matrix(data=5:8, nrow=2,  
ncol=2, byrow=T) )  
  
> z1  
[[1]]  
[1] "water"      "juice"       "lemonade"  
  
[[2]]  
[1] 1 1 2 2 3 3 4 4  
  
[[3]]  
     [,1] [,2]  
[1,]    5    6  
[2,]    7    8
```

# Lists

```
R R Console  
> z1 <- list( c("water", "juice", "lemonade"), rep(1:4, each=2), matrix(data=5:8, nrow=2, ncol=2, byrow=T) )
```

```
R R Console  
> z1  
[[1]]  
[1] "water"      "juice"       "lemonade"  
  
[[2]]  
[1] 1 1 2 2 3 3 4 4  
  
[[3]]  
     [,1] [,2]  
[1,]    5    6  
[2,]    7    8
```

# Lists

Access the elements of a list using the operator `[[]]`

Following commands work.

```
> z1[[1]]  
[1] "water" "juice" "lemonade"
```

Suppose we want to extract "juice". The command

```
> z1[1][2] # Notice the positions of brackets  
[[1]] NULL
```

returns `NULL` instead of "juice", while

```
> z1[[1]][2] # Notice the positions of brackets  
[1] "juice"
```

finally returns the desired result.

# Lists

```
R Console
> z1[[1]]
[1] "water"      "juice"       "lemonade"
>
> z1[1][2]
[[1]]
NULL

> z1[[1]][2]
[1] "juice"
```

# **Introduction to R Software**

## **Vector Indexing**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Vector Indexing

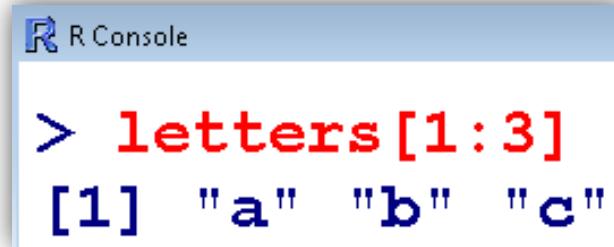
A vector of positive integers (`letters` and `LETTERS` return the 26 lowercase and uppercase letters, respectively).

```
> letters[1:3]
[1] "a" "b" "c"
```

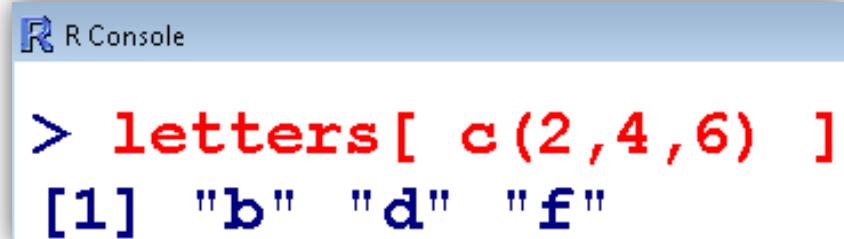
```
> letters[ c(2,4,6) ]
[1] "b" "d" "f"
```

```
> LETTERS[1:3]
[1] "A" "B" "C"
```

```
> LETTERS[ c(2,4,6) ]
[1] "B" "D" "F"
```



R Console  
> letters[1:3]  
[1] "a" "b" "c"



R Console  
> letters[ c(2,4,6) ]
[1] "b" "d" "f"

# Vector Indexing

## □ A logical vector

```
> x <- 1:10
```

```
>x
```

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

```
> x[ (x > 5) ]
```

```
[1] 6 7 8 9 10
```

```
> x[ (x%%2==0) ] #%% indicates x mod y
```

```
[1] 2 4 6 8 10      #values for which x mod 2 is 0
```

```
> x[ (x%%2==1) ]
```

```
[1] 1 3 5 7 9      #values for which x mod 2 is 1
```

## Vector Indexing

R R Console

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> x[ (x>5) ]
[1] 6 7 8 9 10
> x[ (x%%2==0) ]
[1] 2 4 6 8 10
> x[ (x%%2==1) ]
[1] 1 3 5 7 9
```

# Vector Indexing

## □ A logical vector

```
> x[5] <- NA
```

```
> x
```

```
[1] 1 2 3 4 NA 6 7 8 9 10
```

```
> y <- x[ !is.na(x) ] # ! Means negation
```

```
> y
```

```
[1] 1 2 3 4 6 7 8 9 10 # 5 is missing
```

```
> mean(x)
```

```
[1] NA
```

```
> mean(y)
```

```
[1] 5.555556
```

# Vector Indexing

R Console

```
> x[5] <- NA
> x
[1] 1 2 3 4 NA 6 7 8 9 10
>
> y <- x[ !is.na(x) ]
> y
[1] 1 2 3 4 6 7 8 9 10
>
> mean(x)
[1] NA
>
> mean(y)
[1] 5.555556
```

# Vector Indexing

## □ Vector of negative integers

```
> x <- 1:10  
> x  
[1] 1 2 3 4 5 6 7 8 9 10
```

```
> x[-(1:5)]  
[1] 6 7 8 9 10
```

has the same outcome as

```
> x[(6:10)]  
[1] 6 7 8 9 10
```

## Vector Indexing

```
R R Console
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
>
> x[-(1:5)]
[1] 6 7 8 9 10
>
> x[(6:10)]
[1] 6 7 8 9 10
```

# **Introduction to R Software**

## **Vector Indexing**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

## □ String vector

The elements of a vector can be named.

Using these **names**, we can access the vector elements.

**names** is used for functions to get or set the names of an object

```
> z <- list(a1 = 1, a2 = "c", a3 = 1:3)
```

```
> z
```

```
$a1
```

```
[1] 1
```

```
$a2
```

```
[1] "c"
```

```
$a3
```

```
[1] 1 2 3
```

```
> names(z)
```

```
[1] "a1" "a2" "a3"
```

## □ String vector

R Console

```
> z <- list(a1 = 1, a2 = "c", a3 = 1:3)
> z
$a1
[1] 1

$a2
[1] "c"

$a3
[1] 1 2 3

> names(z)
[1] "a1" "a2" "a3"
```

## ❑ String vector

Suppose want to change just the name of the third element.

```
> z <- list(a1 = 1, a2 = "c", a3 = 1:3)  
  
> names(z)[3] <- "c2"  
  
> z  
  
$a1  
[1] 1  
  
$a2  
[1] "c"  
  
$c2  
[1] 1 2 3
```

## □ String vector

R R Console

```
> z <- list(a1 = 1, a2 = "c", a3 = 1:3)
> names(z)[3] <- "c2"
> z
$a1
[1] 1

$a2
[1] "c"

$c2
[1] 1 2 3
```

## □ String vector

### Example

`names` is used for functions to get or set the names of an object

```
> x <- c(water=1, juice=2, lemonade=3 )
```

```
> names(x)
```

```
[1] "water"      "juice"       "lemonade"
```

```
> x["juice"]
```

```
juice
```

```
2
```

## □ String vector

R Console

```
> x <- c(water=1, juice=2, lemonade=3 )
> names(x)
[1] "water"      "juice"       "lemonade"
>
> x["juice"]
juice
2
```

## □ Empty index

```
> x <- 1:10
```

```
>x
```

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

```
> x[]
```

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



R Console

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
>
> x[]
[1] 1 2 3 4 5 6 7 8 9 10
```

## Matrices created from Lists

List can be heterogeneous (mixed modes).

We can start with a heterogeneous list,

give it dimensions, and

thus create a heterogeneous matrix

that is a mixture of numeric and character data:

### Example

```
> ab <- list(1, 2, 3, "X", "Y", "Z")  
> dim(ab) <- c(2,3)  
> print(ab)  
      [,1] [,2] [,3]  
[1,] 1     3     "Y"  
[2,] 2     "X"   "Z"
```

## □ Matrices created from Lists

R R Console

```
> ab <- list(1, 2, 3, "X", "Y", "Z")
> dim(ab) <- c(2,3)
> print(ab)
      [,1] [,2] [,3]
[1,] 1     3     "Y"
[2,] 2     "X"   "Z"
```

# **Introduction to R Software**

## **Factors**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Categorical variables

Quantitative variables

Example:

Height (in meters) – 1.65, 1.76, ...

Qualitative variables

Example:

Gender – Male, Female

Performance – Excellent, Good, Average, Bad ...

# Categorical variables

## Categorical variables

### Example:

X : Gender – Male, Female

X = 0 if a person is male

X = 1 if a person is female

### Example:

Performance	Excellent	Average	Good	Bad	Labels
X	1	2	3	4	Numeric codes

The categories are stored internally as numeric codes, with labels to provide meaningful names for each code.

# Factors

Factors represent categorical variables and are used as grouping indicators.

# Factors

Example:

Suppose we denote the three colours of balls in a basket by following numbers:

**Red = 1, Blue = 2, Green = 3**

Suppose we draw five balls with following colours:

**Red, Green, Green, Blue, Red**

This outcome of colours can be coded by numbers

Performance	Excellent	Average	Good	Bad	Labels
X	1	2	3	4	Numeric codes

# Factors

Each character is mapped to a code.

Factors represent categorical variables and are used as grouping indicators.

The categories are stored internally as numeric codes, with labels to provide meaningful names for each code.

# Factors

The order of the labels is important.

First label is mapped to code 1.

Second label is mapped to code 2 and so on.

The values of the codes are always restricted to  $1, 2, \dots, k$ , to represent  $k$  discrete categories.

Here “**Red**” is mapped to code 1,

“**Blue**” is mapped to code 2 and

“**Green**” is mapped to code 3.

# Factors

We have a vector of character strings or integers.

R's term for a categorical variable is a factor.

In R, each possible value of a categorical variable is called a level.

A vector of levels is called a factor.

A categorical variable is characterized by a (here: finite) number of levels called as factor levels.

# Factors

To define a factor, we start with

- a vector of values,
- a second vector that gives the collection of possible values, and
- a third vector that gives labels to the possible values.

# Factors

The `factor` function encodes the vector of discrete values into a factor:

`factor(x)`

where `x` is a vector of strings or integers.

If the vector contains only a subset of possible values and not the entire values, then include a second argument that gives the possible levels of the factor:

`factor(x, levels)`

# Factors

## Usage

```
factor(x = character(), levels, labels =  
levels, exclude = NA, ...)
```

- levels** : Determines the categories of the factor variable.  
Default is the sorted list of all the distinct values of **x**.
- labels** : (Optional) Vector of values that will be the labels of the categories in the **levels** argument.
- exclude** : (Optional) It defines which levels will be classified as **NA** in any output using the factor variable.

# Factors

Look into **help**

> **help("factor")**

factor {base}

R Documentation

## Factors

### Description

The function `factor` is used to encode a vector as a factor (the terms ‘category’ and ‘enumerated type’ are also used for factors). If argument `ordered` is `TRUE`, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

### Usage

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax = NA)
```

```
ordered(x, ...)
```

```
is.factor(x)
is.ordered(x)
```

# Factors

## Example:

Suppose we roll a die seven times and observe the outcome in the vector **y**.

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



Possible values of upper face of die are 1 to 6 and we store them in a vector **possible.dieface**

```
> possible.dieface <- c(1, 2, 3, 4, 5, 6)
```

# Factors

## Example:

We wish to label the rolls by the words “one”, “two”, ..., “six”.

We put these labels in the vector `labels.diefaces`:

```
> labels.dieface <- c("one", "two", "three",
  "four", "five", "six")
```

Construct the factor variable `facy` using the function `factor`:

```
> facy <- factor(y, levels = possible.dieface,
  labels = labels.dieface)
```

# Factors

## Example:

Observe the difference between a character vector and a factor.

```
> facy
```

```
[1] one four three five four two four
```

```
Levels: one two three four five six
```

Note that

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

# Factors

```
R Console
> y <- c(1, 4, 3, 5, 4, 2, 4)
> y
[1] 1 4 3 5 4 2 4
>
> possible.dieface <- c(1, 2, 3, 4, 5, 6)
> possible.dieface
[1] 1 2 3 4 5 6
>
> labels.dieface <- c("one", "two", "three", "four", "five", "six")
> labels.dieface
[1] "one"    "two"    "three"   "four"    "five"    "six"
>
> facy <- factor(y, levels=possible.dieface, labels=labels.dieface)
> facy
[1] one    four   three  five  four   two    four
Levels: one two three four five six
```

# **Introduction to R Software**

## **Factors**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Factors

The `factor` function encodes the vector of discrete values into a factor:

`factor(x)`

where `x` is a vector of strings or integers.

`factor(x, levels)`

`factor(x = character(), levels, labels = levels, exclude = NA, ...)`

# Factors

## Example

```
> x <- factor( c("juice", "juice", "lemonade",
"juice", "water") )  
  
>x  
[1] juice      juice      lemonade    juice      water  
Levels: juice lemonade water
```

The single levels are ordered alphabetically:

**juice** --- **lemonade** --- **water**

# Factors

## Example

```
R R Console
> x <- factor( c("juice", "juice", "lemonade", "juice", "water") )
> x
[1] juice      juice      lemonade  juice      water
Levels: juice lemonade water
```

# Factors

`unclass` function :

All objects in R have a class and function `class` reports it.

For simple vectors, this is just the mode, e.g. "`numeric`", "`logical`", "`character`", "`list`", "`matrix`", "`array`", "`factor`" and "`data.frame`".

A special attribute `class` of the object is used to allow for an object-oriented style of programming in R.

# Factors

`unclass` function

For example if an object has class "`data.frame`", it will be printed in a certain way, the `plot()` function will display it graphically in a certain way etc.

`unclass()` is used to temporarily remove the effects of class.

Use `help("unclass")` to get more information.

# Factors

The command `unclass` shows, an integer is assigned to every factor level:

```
> x <- factor( c("juice", "juice", "lemonade",
  "juice", "water") )

> unclass(x)
[1] 1 1 2 1 3

attr(,"levels")
[1] "juice" "lemonade" "water"
```

# Factors

R R Console

```
> unclass(x)
[1] 1 1 2 1 3
attr(,"levels")
[1] "juice"      "lemonade"    "water"
```

# Factors

If a different assignment is desired, the parameter `levels` can be used:

```
> x <- factor( c("juice", "juice", "lemonade",
"juice", "water"),
levels=c("water", "juice", "lemonade") )
```

```
> x
[1] juice      juice      lemonade  juice      water
Levels: water juice lemonade
```

# Factors

```
> unclass(x)
[1] 2 2 3 2 1
attr(,"levels")
[1] "water"      "juice"       "lemonade"

> levels(x)
[1] "water"      "juice"       "lemonade"
```

# Factors

```
R Console
> x <- factor( c("juice", "juice", "lemonade", "juice", "water"),
+ levels=c("water", "juice", "lemonade") )
> x
[1] juice      juice      lemonade juice      water
Levels: water juice lemonade
>
> unclass(x)
[1] 2 2 3 2 1
attr(,"levels")
[1] "water"     "juice"      "lemonade"
>
> levels(x)
[1] "water"     "juice"      "lemonade"
```

# Factors

Example for an ordered factor:

```
> income <- ordered(c("high", "high", "low",
"medium", "medium"), levels=c("low", "medium",
"high") )  
  
> income  
[1] high    high    low     medium medium  
Levels: low < medium < high  
  
> unclass(income)  
[1] 3 3 1 2 2  
attr(,"levels")  
[1] "low"      "medium"   "high"
```

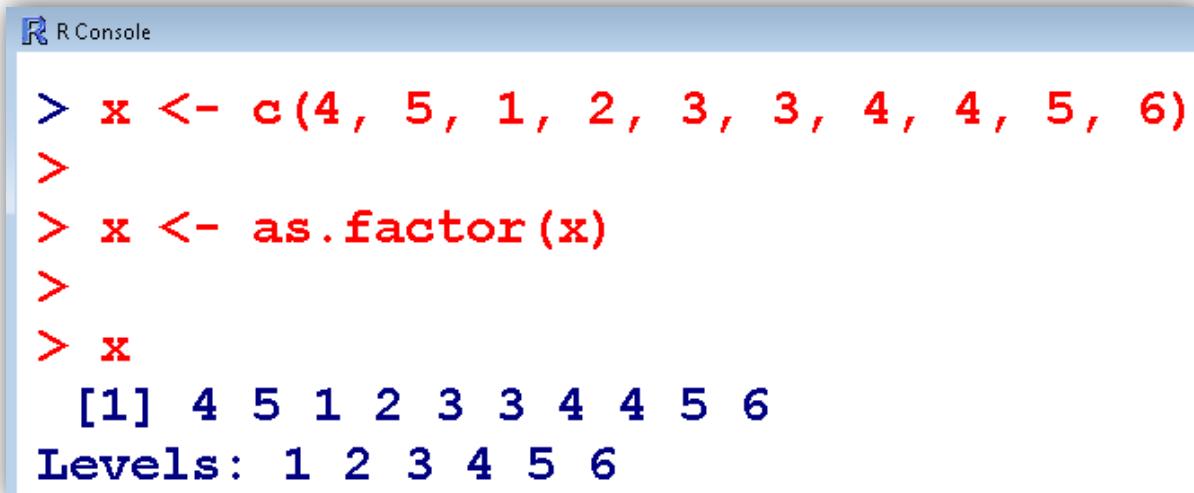
# Factors

```
R R Console
> income <- ordered(c("high", "high", "low", "medium", "medium"),
+ levels=c("low", "medium", "high") )
> income
[1] high   high   low    medium medium
Levels: low < medium < high
>
> unclass(income)
[1] 3 3 1 2 2
attr(,"levels")
[1] "low"   "medium" "high"
```

# Factors

A vector can be turned into a factor with the command `as.factor`:

```
> x <- c(4, 5, 1, 2, 3, 3, 4, 4, 5, 6)  
  
> x <- as.factor(x)  
> x  
[1] 4 5 1 2 3 3 4 4 5 6  
Levels: 1 2 3 4 5 6
```



The screenshot shows the R Console window with the following text:

```
R R Console  
  
> x <- c(4, 5, 1, 2, 3, 3, 4, 4, 5, 6)  
>  
> x <- as.factor(x)  
>  
> x  
[1] 4 5 1 2 3 3 4 4 5 6  
Levels: 1 2 3 4 5 6
```

# **Introduction to R Software**

## **Strings – Display and Formatting**

**:::**

## **Print and Format Function**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# **Strings**

- **Formatting and Display of Strings**
- **Operations with Strings**

**We need formatting and display of strings to obtain the results of specific operations in required format.**

# **Formatting and Display of Strings**

Important commands regarding formatting and display are **print**, **format**, **cat**, and **paste**.

**print** function prints its argument.

Usage

**print()**

**print()** is a generic command that is available for every object class.

# Formatting and Display of Strings

Examples:

```
> print( sqrt(2) )  
[1] 1.414214
```

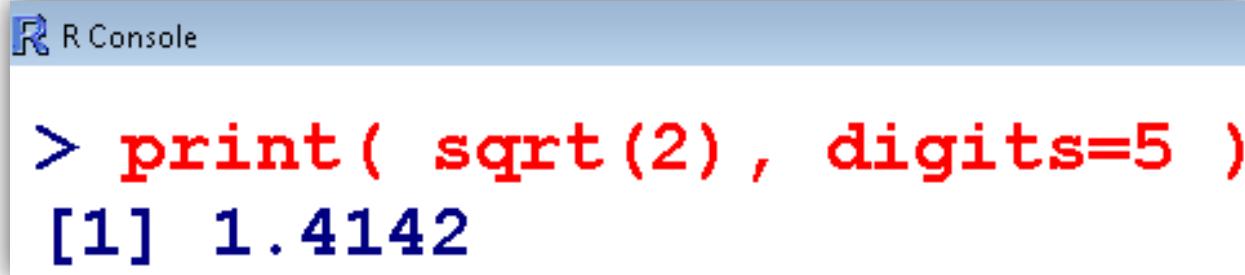
R R Console

```
> print( sqrt(2) )  
[1] 1.414214
```

# Formatting and Display of Strings

Examples:

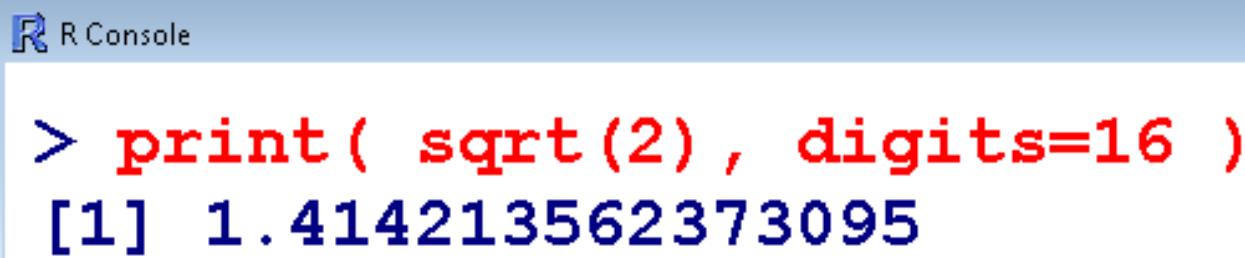
```
> print( sqrt(2), digits=5 )  
[1] 1.4142
```



R Console

```
> print( sqrt(2) , digits=5 )  
[1] 1.4142
```

```
> print( sqrt(2), digits=16 )  
[1] 1.414213562373095
```



R Console

```
> print( sqrt(2) , digits=16 )  
[1] 1.414213562373095
```

# Formatting and Display of Strings

Format an R object for pretty printing.

Usage

```
format(x, ...)
```

**x** is any R object; typically numeric.

# Formatting and Display of Strings

## Usage

```
format(x, trim = FALSE, digits = NULL, nsmall  
= 0L, justify = c("left", "right", "centre",  
"none"), width = NULL,     ...)
```

- digits** shows how many significant digits are to be used
- nsmall** shows the minimum number of digits to the right of the decimal point
- justify** provides left-justified (the default), right-justified, or centred.

# Formatting and Display of Strings

```
> print( format( 0.5, digits=10, nsmall=15 ) )  
[1] "0.500000000000000"
```

R R Console

```
> print( format( 0.5, digits=10, nsmall=15 ) )  
[1] "0.500000000000000"
```

# Formatting and Display of Strings

**Example:**

```
> x <- matrix(nrow=3, ncol=2, data=1:6, byrow=T)

> print(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Here, a matrix is displayed in the R command window.

One can specify the desired number of digits with the option **digits**.

# Formatting and Display of Strings

R R Console

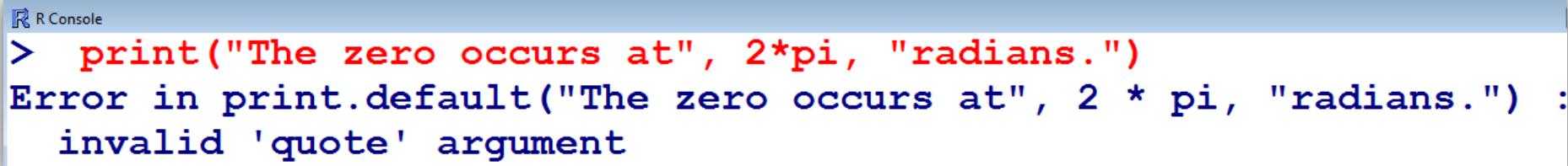
```
> x <- matrix(nrow=3, ncol=2, data=1:6, byrow=T)
>
> print(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

# Formatting and Display of Strings

The `print` function has a significant limitation that it prints only one object at a time.

Trying to print multiple items gives error message:

```
> print("The zero occurs at", 2*pi, "radians.")  
Error in print.default("The zero occurs at", 2 *  
pi, "radians.") :  
invalid 'quote' argument
```



A screenshot of an R console window titled "R Console". The window shows the following text:  
> print("The zero occurs at", 2\*pi, "radians.")  
Error in print.default("The zero occurs at", 2 \* pi, "radians.") :  
invalid 'quote' argument

# Formatting and Display of Strings

The only way to print multiple items is to print them one at a time

```
> print("The zero occurs at"); print(2*pi);  
print("radians")  
[1] "The zero occurs at"  
[1] 6.283185  
[1] "radians"
```

The `cat` function is an alternative to `print` that lets you combine multiple items into a continuous output.

# **Introduction to R Software**

## **Strings – Display and Formatting**

**:::**

## **Print and Format with Concatenate**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Formatting and Display of Strings

## cat function

- The function `cat()` concatenate (link in the same sequence) and print the arguments in strings, concatenates them and prints the entire string in the command window.
- `cat` is useful for producing output in user-defined functions.
- It converts its arguments to character vectors, link them together in the same sequence to a single character vector, appends the given `sep = string(s)` to each element and then outputs them.

# Formatting and Display of Strings

## cat function

### Usage

```
cat(... , file = "", sep = " ", ...)
```

**cat** puts a space between each item by default.

One must provide a newline character (`\n`) (newline) to terminate the line.

# Formatting and Display of Strings

The only way to print multiple items is to print them one at a time

```
> print("The zero occurs at"); print(2*pi);
print("radians")
[1] "The zero occurs at"
[1] 6.283185
[1] "radians"
```

The `cat` function is an alternative to `print` that lets you combine multiple items into a continuous output:

```
> cat("The zero occurs at", 2*pi, "radians.", 
"\n")
The zero occurs at 6.283185 radians.
```

# Formatting and Display of Strings

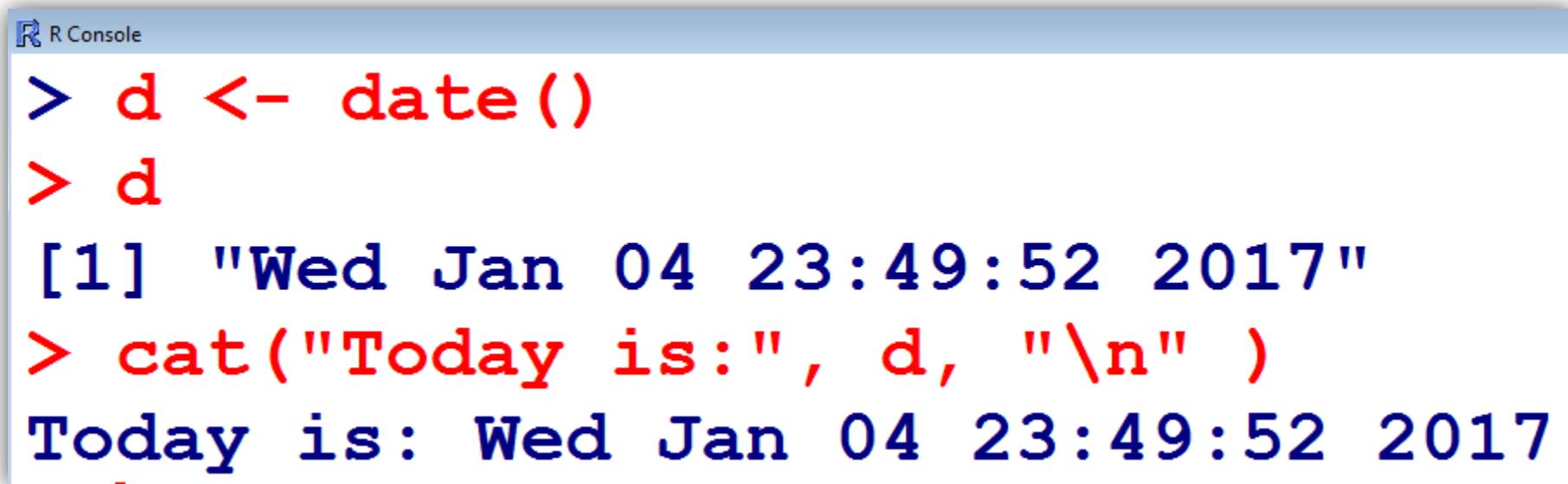
R Console

```
> print("The zero occurs at"); print(2*pi); print("radians")
[1] "The zero occurs at"
[1] 6.283185
[1] "radians"
>
> cat("The zero occurs at", 2*pi, "radians.", "\n")
The zero occurs at 6.283185 radians.
```

# Formatting and Display of Strings

```
> d <- date()  
> cat("Today is:", d, "\n" )
```

Today is: Wed Jan 04 23:49:52 2017



A screenshot of the R Console window. The title bar says "R Console". The console area contains the following text:

```
> d <- date()  
> d  
[1] "Wed Jan 04 23:49:52 2017"  
> cat("Today is:", d, "\n" )  
Today is: Wed Jan 04 23:49:52 2017
```

# Formatting and Display of Strings

```
> x <- 7  
> cat("The square of", x, "is", x^2, "!\\n")
```

The square of 7 is 49 !

R R Console

```
> x <- 7  
> cat("The square of", x, "is", x^2, "!\\n")  
The square of 7 is 49 !
```

# Formatting and Display of Strings

```
> cat("The square root of",x,"is  
approximately", format(sqrt(x),digits=3),"\n")
```

The square root of 7 is approximately 2.65

R Console

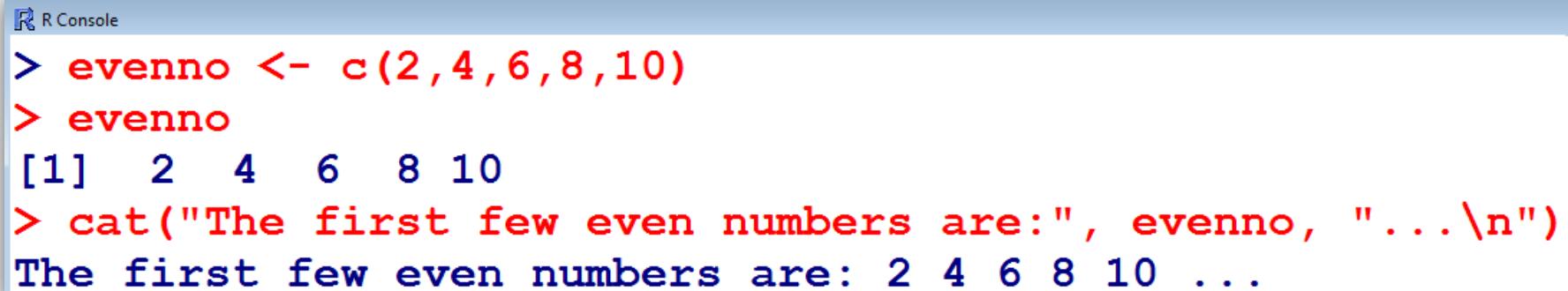
```
> cat("The square root of",x,"is approximately", format(sqrt(x),digits=3),"\n")  
The square root of 7 is approximately 2.65
```

# Formatting and Display of Strings

The `cat` function can also print simple vectors

```
> evenno <- c(2,4,6,8,10)
> evenno
[1] 2 4 6 8 10
```

```
> cat("The first few even numbers are:",
evenno, "...\\n")
The first few even numbers are: 2 4 6 8 10 ...
```



R Console

```
> evenno <- c(2,4,6,8,10)
> evenno
[1] 2 4 6 8 10
> cat("The first few even numbers are:", evenno, "...\\n")
The first few even numbers are: 2 4 6 8 10 ...
```

# **Introduction to R Software**

## **Strings – Display and Formatting**

**:::**

### **Paste Function**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Formatting and Display of Strings

## paste function

- The `paste()` function concatenates several strings together.
- It creates a new string by joining the given strings end to end.
- The result of `paste()` can be assigned to a variable  
(in contrast to the function `cat()`).

# Formatting and Display of Strings

## Usage

```
paste (..., sep = " ", collapse = NULL).
```

**collapse** is an optional character string to separate the results.

- The parameter **sep** is a string that serves as a separation between the strings that are given as input.
- **paste** inserts a single space between pairs of strings.
- A desired line break can be achieved with "**\n**" (newline).

# Formatting and Display of Strings

```
> paste("Everybody", "loves", "R Programming.")  
[1] "Everybody loves R Programming."
```

```
> paste("Everybody", "loves", "R Programming.",  
sep="*")  
[1] "Everybody*loves*R Programming."
```

```
> paste("Everybody", "loves", "R Programming.",  
sep="====")  
[1] "Everybody====loves====R Programming."
```

# Formatting and Display of Strings

R R Console

```
> paste("Everybody", "loves", "R Programming.")
[1] "Everybody loves R Programming."
>
> paste("Everybody", "loves", "R Programming.", sep="*")
[1] "Everybody*loves*R Programming."
>
> paste("Everybody", "loves", "R Programming.", sep="====")
[1] "Everybody====loves====R Programming."
```

# Formatting and Display of Strings

If one or more arguments are vectors of strings, **paste** will generate all combinations of the arguments:

```
> names <- c("Prof. Singh", "Mr. Venkat", "Dr. Jha")  
  
> paste(names, "is", "a good", "person.")  
[1] "Prof. Singh is a good person."  
[2] "Mr. Venkat is a good person."  
[3] "Dr. Jha is a good person."
```

# Formatting and Display of Strings

When we want to join even those combinations into one, big string.

The `collapse` parameter defines a top-level separator and instructs `paste` to concatenate the generated strings using that separator:

```
> names <- c("Prof. Singh", "Mr. Venkat", "Dr. Jha")  
  
> paste(names, "is", "a good", "person.",  
collapse=", and ")  
[1] "Prof. Singh is a good person., and Mr.  
Venkat is a good person., and Dr. Jha is a good  
person."
```

# Formatting and Display of Strings

```
RGui (64-bit)

> paste(names, "is", "a good", "person.")
[1] "Prof. Singh is a good person."
[2] "Mr. Venkat is a good person."
[3] "Dr. Jha is a good person."
> paste(names, "is", "a good", "person.", collapse=", and $"
[1] "Prof. Singh is a good person., and Mr. Venkat is a g$
```

# Operations with Strings

Example:

```
> x <- paste("Ex", 1:5, sep="_")  
>x  
[1] "Ex_1" "Ex_2" "Ex_3" "Ex_4" "Ex_5"  
  
> x[1]  
[1] "Ex_1"  
  
> x[2]  
[1] "Ex_2"  
  
> x[3]  
[1] "Ex_3"  
  
> x[5]  
[1] "Ex_5"
```

# Operations with Strings

```
R R Console
> x <- paste("Ex", 1:5, sep="_")
> x
[1] "Ex_1" "Ex_2" "Ex_3" "Ex_4" "Ex_5"
>
> x[1]
[1] "Ex_1"
> x[2]
[1] "Ex_2"
> x[3]
[1] "Ex_3"
> x[5]
[1] "Ex_5"
```

# Operations with Strings

**x** is a vector of strings.

If we use the parameter **collapse**, a single string, instead of a vector of strings, is created:

```
> x <- paste("Ex", 1:5, sep="_", collapse="")  
> x[1]  
[1] "Ex_1Ex_2Ex_3Ex_4Ex_5"
```

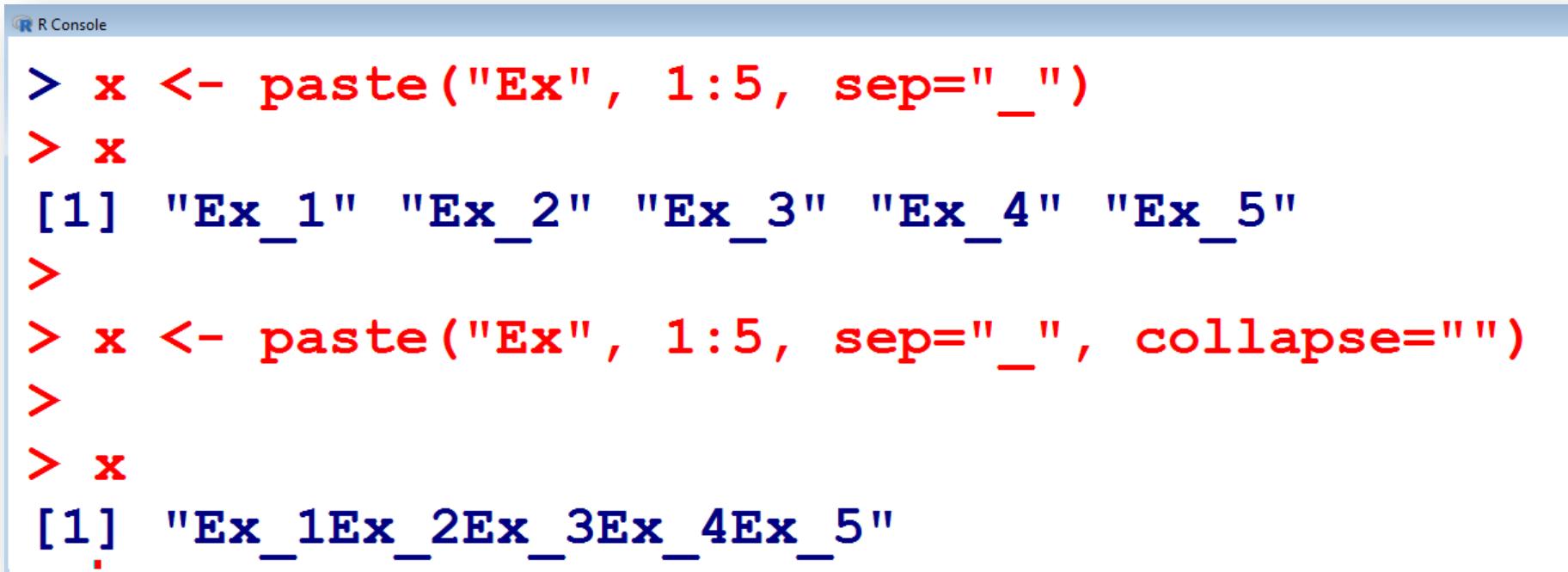
# Operations with Strings

Note the difference between

```
x <- paste("Ex", 1:5, sep="_")
```

and

```
x <- paste("Ex", 1:5, sep="_", collapse="")
```



The screenshot shows the R console interface with a light blue header bar containing the R logo and the text "R Console". Below the header, there are two sets of R code and their corresponding outputs.

```
> x <- paste("Ex", 1:5, sep="_")
> x
[1] "Ex_1" "Ex_2" "Ex_3" "Ex_4" "Ex_5"
>
> x <- paste("Ex", 1:5, sep="_", collapse="")
>
> x
[1] "Ex_1Ex_2Ex_3Ex_4Ex_5"
```

# **Introduction to R Software**

## **Strings – Display and Formatting**

**:::**

### **Splitting**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Operations with Strings

Command **strsplit**, split the elements of a character vector.

"Split" can be a single character, or a character string:

Usage

```
strsplit(x, split, fixed = FALSE, ...)
```

Arguments

**x** character vector, each element of which is to be split.

**split** character vector containing regular expression(s)  
(unless `fixed = TRUE`) to use for splitting.

# Operations with Strings

- With a command `strsplit`, we can split a string in pieces.

```
> x <- "The&!syntax&!of&!paste&!is!&available!
&inthe online-help"
```

```
> x
[1] "The&!syntax&!of&!paste&!is!&available!
&inthe online-help"

> strsplit(x, "!")
[[1]]
[1] "The&"           "syntax&"        "of&"
[4] "paste&"          "is"            "&available"
[7] "&inthe online-help"
```

# Operations with Strings

```
R Console
> x <- "The&!syntax&!of&!paste&!is!&available! &inthe online-help"
> x
[1] "The&!syntax&!of&!paste&!is!&available! &inthe online-help"
>
> strsplit(x, "!")
[[1]]
[1] "The&"                      "syntax&"                  "of&"
[4] "paste&"                     "is"                      "&available"
[7] "&inthe online-help"
```

# Operations with Strings

```
> x <- "The&!syntax&!of&!paste&!is!&available!
&inthe online-help"

> strsplit(x,"&!")
[[1]]
[1] "The"           "syntax&"
[3] "of"            "paste"
[5] "is!&available!&in the online-help"
```

# Operations with Strings

```
R Console
> strsplit(x, "&!")
[[1]]
[1] "The"
[2] "syntax"
[3] "of"
[4] "paste"
[5] "is! &available! &inthe online-help"
```

# Operations with Strings

```
> x <- "The&!syntax&!of&!paste&!is!&available!
&inthe online-help"

> x
[1] "The&!syntax&!of&!paste&!is!&available!
&inthe online-help"

> l1 <- strsplit(x,"!&")

> l1
[[1]]
[1] "The&!syntax&!of&!paste&!is"      "available!
&inthe online-help"
```

# Operations with Strings

```
R Console
> x <- "The syntax of paste is available! &inthe online-help"
> x
[1] "The syntax of paste is available! &inthe online-help"
>
> ll <- strsplit(x,"!&")
> ll
[[1]]
[1] "The syntax of paste"      "available! &inthe online-help"
```

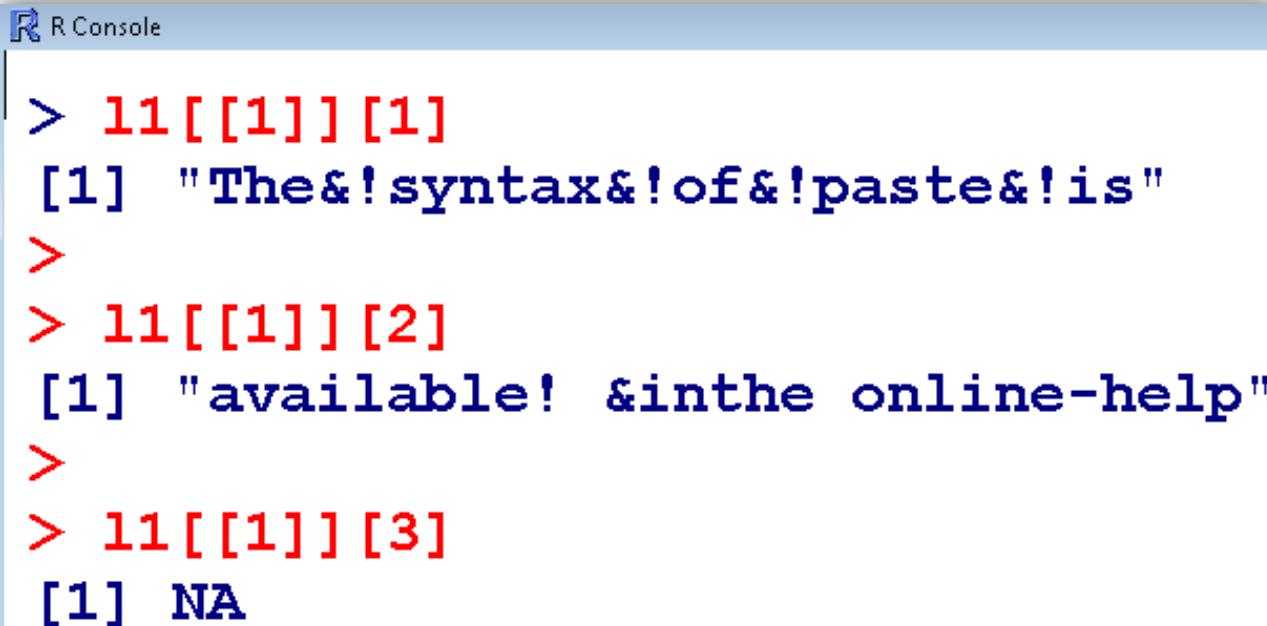
# Operations with Strings

Notice the access to single components:

```
> l1[[1]][1]
[1] "The&!syntax&!of&!paste&!is"

> l1[[1]][2]
[1] "available! &inthe online-help"

> l1[[1]][3]
[1] NA
```



A screenshot of the R console window titled "R Console". The window shows the same R code and output as the previous text block, demonstrating how to access individual components of a character vector.

```
> l1[[1]][1]
[1] "The&!syntax&!of&!paste&!is"
>
> l1[[1]][2]
[1] "available! &inthe online-help"
>
> l1[[1]][3]
[1] NA
```

# **Introduction to R Software**

## **Strings – Display and Formatting**

⋮

## **Replacement and Manipulations with Alphabets**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Operations with Strings

There are a variety of commands that can be used for strings.

Examples:

Count of number of characters:

```
> x <- "R course 24.07.2017"
```

```
> y <- "Number of participants: 25"
```

```
> nchar(x) #Count the Number of Characters in x  
[1] 19
```

```
> nchar(y) #Count the Number of Characters in y  
[1] 26
```

# Operations with Strings

```
R Console
> x <- "R course 24.07.2017"
> y <- "Number of participants: 25"
> x
[1] "R course 24.07.2017"
> y
[1] "Number of participants: 25"
> nchar(x)
[1] 19
>
> nchar(y)
[1] 26
```

# Operations with Strings

**sub** and **gsub** Functions:

Within a string, we want to replace one substring with another.

Use **sub** and **gsub** to replace the first instance of a substring:

`sub(old, new, string)`

The **sub** function finds the first instance of the old substring within string and replaces it with the new substring.

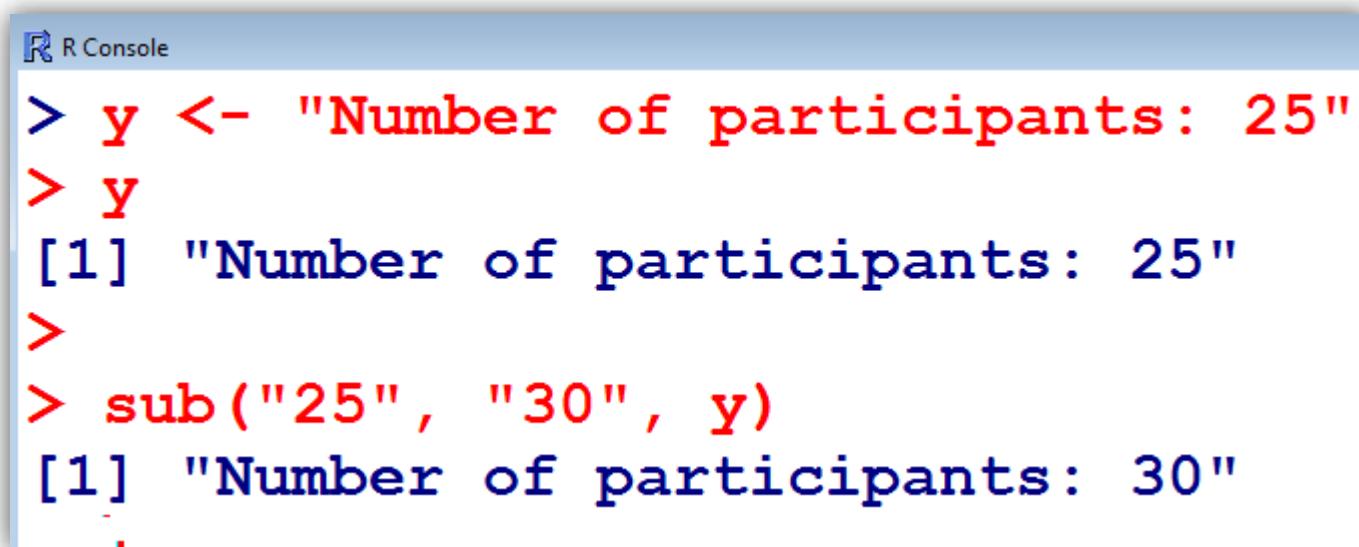
**gsub** does the same thing, but it replaces all instances of the substring (a global replace), not just the first.

`gsub(old, new, string)`

# Operations with Strings

Examples:

```
> y <- "Number of participants: 25"  
  
> sub("25", "30", y)  
[1] "Number of participants: 30"
```



A screenshot of an R console window titled "R Console". The window contains the following R code and its output:

```
> y <- "Number of participants: 25"  
> y  
[1] "Number of participants: 25"  
>  
> sub("25", "30", y)  
[1] "Number of participants: 30"
```

# Operations with Strings

## Examples:

```
> y <- "Mr. Singh is the smart one. Mr. Singh  
is funny, too."
```

```
> y  
[1] "Mr. Singh is the smart one. Mr. Singh is  
funny, too."
```

```
> sub("Mr. Singh", "Professor Jha", y)  
[1] "Professor Jha is the smart one. Mr. Singh  
is funny, too."
```

# Operations with Strings

```
R Console
> y <- "Mr. Singh is the smart one. Mr. Singh is funny, too."
> y
[1] "Mr. Singh is the smart one. Mr. Singh is funny, too."
>
> sub("Mr. Singh","Professor Jha", y)
[1] "Professor Jha is the smart one. Mr. Singh is funny, too."
```

# Operations with Strings

Examples:

```
> y <- "Mr. Singh is the smart one. Mr. Singh  
is funny, too."  
  
> gsub("Mr. Singh","Professor Jha", y)  
[1] "Professor Jha is the smart one. Professor  
Jha is funny, too."
```

Recall

```
> sub("Mr. Singh","Professor Jha", y)  
[1] "Professor Jha is the smart one. Mr. Singh  
is funny, too."
```

# Operations with Strings

```
R Console
> y <- "Mr. Singh is the smart one. Mr. Singh is funny, too."
> y
[1] "Mr. Singh is the smart one. Mr. Singh is funny, too."
> gsub("Mr. Singh","Professor Jha", y)
[1] "Professor Jha is the smart one. Professor Jha is funny, too."
>
> sub("Mr. Singh","Professor Jha", y)
[1] "Professor Jha is the smart one. Mr. Singh is funny, too."
.
```

# Operations with Strings

**tolower(x) and toupper(x) Functions:**

**tolower(x) and toupper(x) convert upper-case  
characters in a character vector to lower-case, or vice versa.**

**Non-alphabetic characters are left unchanged.**

# Operations with Strings

`tolower(x)` and `toupper(x)` Functions:

Examples:

```
> x <- "R course will start from 24.07.2017"  
  
> toupper(x)  
[1] "R COURSE WILL START FROM 24.07.2017"  
  
> z<- "R COURSE WILL START FROM 24.07.2017"  
  
> tolower(z)  
[1] "r course will start from 24.07.2017"
```

# Operations with Strings

```
R R Console
> x <- "R course will start from 24.07.2017"
> x
[1] "R course will start from 24.07.2017"
>
> toupper(x)
[1] "R COURSE WILL START FROM 24.07.2017"
```

```
R R Console
> z<-"R COURSE WILL START FROM 24.07.2017"
> z
[1] "R COURSE WILL START FROM 24.07.2017"
>
> tolower(z)
[1] "r course will start from 24.07.2017"
```

# **Introduction to R Software**

## **Strings – Display and Formatting**

**:::**

## **Replacement and Evaluation of Strings**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Operations with Strings

R has various functions for regular expression based match and replaces.

Some functions (e.g., `grep`, `grepl`, etc.) are used for searching for matches and functions whereas `sub` and `gsub` are used for performing replacement.

# Operations with Strings

**grep function:**

The **grep** function is used for searching the matches.

( **sub** and **gsub** are used for performing replacement. )

**grep :** Globally search regular expression and print it

**grep(pattern, x)** search for matches to argument

**pattern** within each element of a character vector **x**.

It returns an integer vector of the indices of the elements of **x** that yielded a match

# Operations with Strings

`grep(pattern, x, value = TRUE)` returns a character vector containing the selected elements of x.

```
> str <- c("R Course", "exercises", "include  
examples of R language")  
  
> grep("ex", str, value=T)  
[1] "exercises" "include examples of R language"
```

# Operations with Strings

`grep(pattern, x, value = FALSE)` returns an integer vector of the indices of the elements of `x` that yielded a match

`value = FALSE` is default.

```
> str <- c("R Course", "exercises", "include  
examples of R language")  
  
> grep("ex", str, value=F)  
[1] 2 3
```

# Operations with Strings

```
R R Console
> str <- c("R Course", "exercises", "include examples of R language")
>
> grep("ex",str,value=T)
[1] "exercises"                      "include examples of R language"
>
> grep("ex",str,value=F)
[1] 2 3
```

# Operations with Strings

Example:

```
> x <- "R course 24.07.2017"  
  
> y <- "Number of participants: 25"  
  
> c(x,y) # Combine the two strings  
[1] "R course 24.07.2017"  "Number of  
participants: 25"  
  
> grep("our", c(x,y) )  
[1] 1
```

"our" is in the 1st element (in the word "course"), therefore in x.  
There is no "our" in y.

# Operations with Strings

Example:

```
x <- "R course 24.07.2017"
```

```
y <- "Number of participants: 25"
```

```
> c(x,y) # Combine the two strings
```

```
[1] "R course 24.07.2017" "Number of  
participants: 25"
```

```
> grep("Num", c(x,y) )
```

```
[1] 2
```

**"Num" is in the 2nd element (in the word "Number"), therefore in y.**

There is no "Num" in x.

# Operations with Strings

grep function:

```
R Console
> x <- "R course 24.07.2017"
> x
[1] "R course 24.07.2017"
> y <- "Number of participants: 25"
> y
[1] "Number of participants: 25"
> c(x,y)
[1] "R course 24.07.2017"
[2] "Number of participants: 25"
> grep("our", c(x,y) )
[1] 1
```

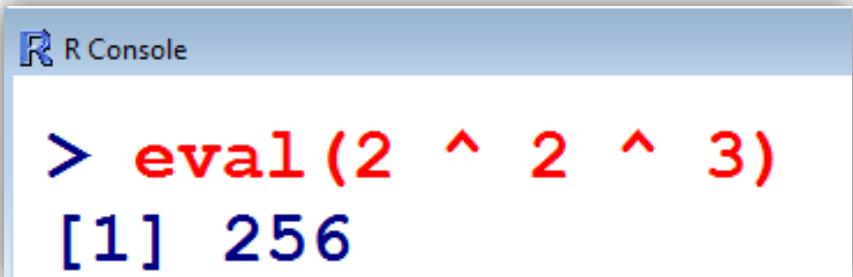
# Operations with Strings

**eval** function:

**eval** function evaluates an (Unevaluated) R expression in a specified environment.

**Example:**

```
> eval(2 ^ 2 ^ 3)  
[1] 256
```



R Console

```
> eval(2 ^ 2 ^ 3)  
[1] 256
```

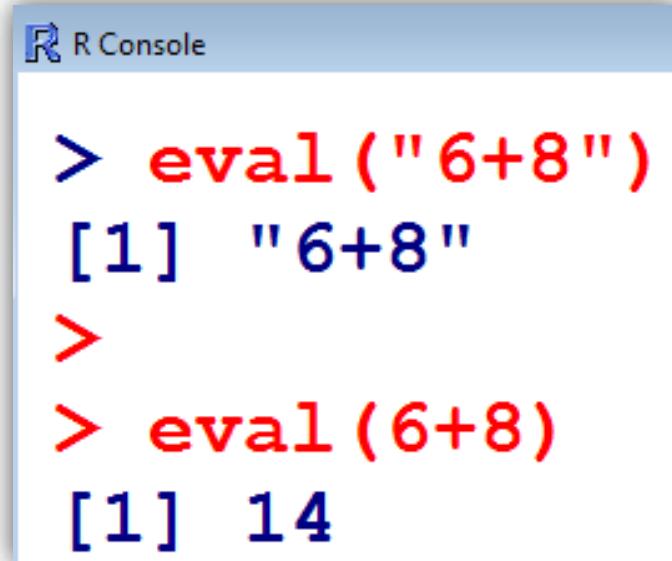
# Operations with Strings

**eval** function:

**Example:**

```
> eval("6+8")
[1] "6+8"

> eval(6+8)
[1] 14
```



A screenshot of the R Console window. The title bar says "R Console". The console area contains the following text:

```
> eval("6+8")
[1] "6+8"
>
> eval(6+8)
[1] 14
```

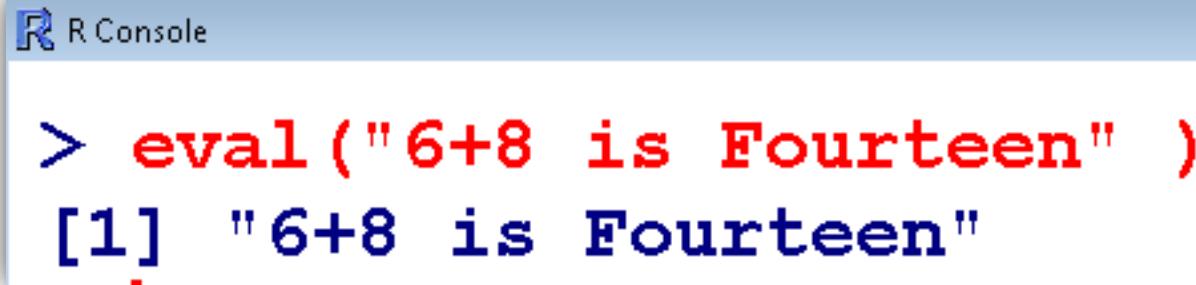
The **eval()** function evaluates an expression, but "**6+8**" is a string, not an expression whereas **6+8** is not an expression.

# Operations with Strings

**eval** function:

Example:

```
> eval("6+8 is Fourteen" )  
[1] "6+8 is Fourteen"
```



A screenshot of an R console window titled "R Console". The window shows the command `> eval("6+8 is Fourteen" )` and its output `[1] "6+8 is Fourteen"`. The text is displayed in red and blue colors.

```
R Console  
> eval("6+8 is Fourteen" )  
[1] "6+8 is Fourteen"
```

# Operations with Strings

`parse` function:

`parse()` with `text=string` is used to change the string into an expression.

**Example:**

```
> eval("6+8")
[1] "6+8"

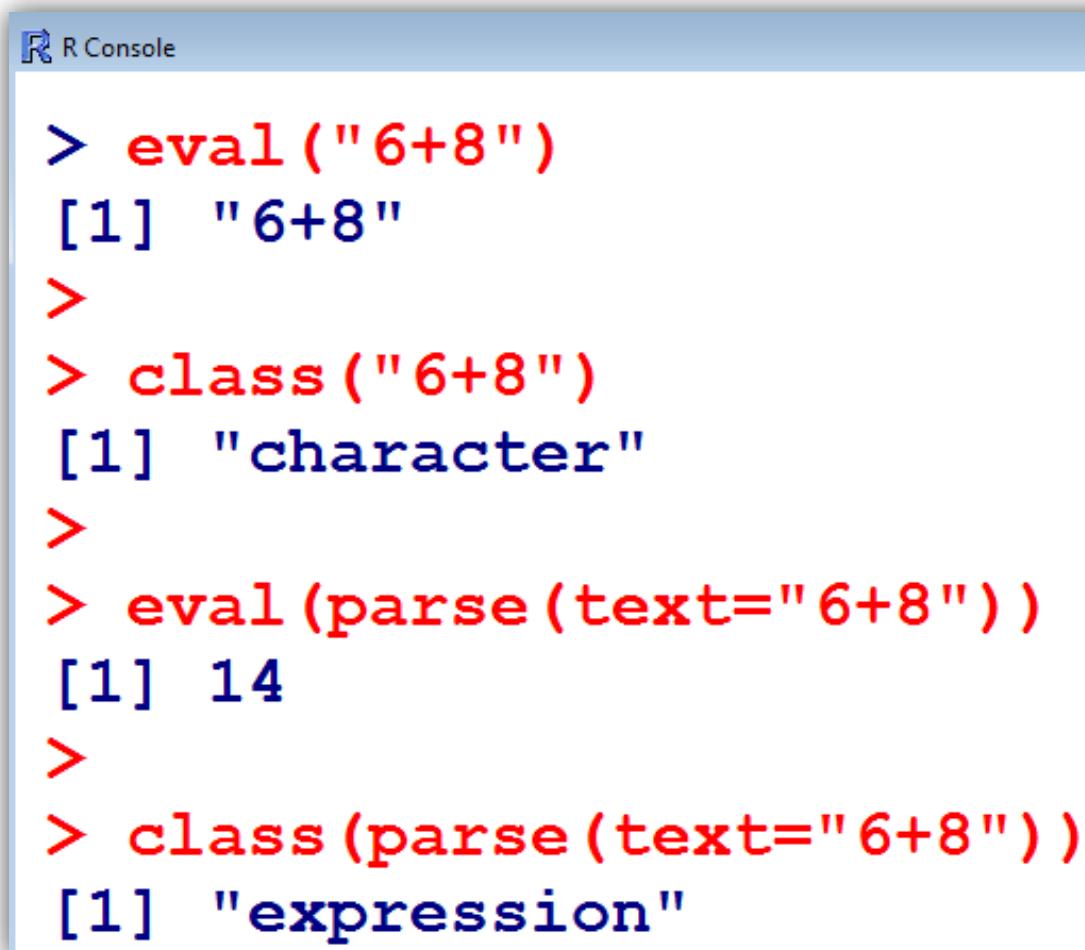
> class("6+8")
[1] "character"

> eval(parse(text="6+8"))
[1] 14

> class(parse(text="6+8"))
[1] "expression"
```

# Operations with Strings

`parse` function:



The image shows a screenshot of an R console window titled "R Console". The console displays the following R session:

```
> eval("6+8")
[1] "6+8"
>
> class("6+8")
[1] "character"
>
> eval(parse(text="6+8"))
[1] 14
>
> class(parse(text="6+8"))
[1] "expression"
```

# **Introduction to R Software**

## **Data Frames**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Data Frames

The commands `c`, `cbind`, `vector` and `matrix` functions combine data.

Another option is the data frame.

In a data frame, we can combine variables of equal length, with each row in the data frame containing observations on the same unit.

Hence, it is similar to the `matrix` or `cbind` functions.

Advantage is that one can make changes to the data without affecting the original data.

# Data Frames

One can also combine numerical variables, character strings as well as factors in data frame.

For example, `cbind` and `matrix` functions can not be used to combine different types of data

Data frames are special types of objects in R designed for data sets.

The data frame format is similar to a spreadsheet, where columns contain variables and observations are contained in rows.

# Data Frames

**Data frames contain complete data sets that are mostly created with other programs (spreadsheet-files, software SPSS-files, Excel-files etc.).**

**Variables in a data frame may be numeric (numbers) or categorical (characters or factors).**

# Data Frames

## Example:

Package “**MASS**” describes functions and datasets to support Venables and Ripley, ``Modern Applied Statistics with S'' (4<sup>th</sup> edition 2002)

# Data Frames

An example data frame `painters` is available in the library.

MASS (here only an excerpt of a data set):

```
> library(MASS)  
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.

Here, the names of the painters serve as row identifications, i.e., every row is assigned to the name of the corresponding painter.

# Data Frames

R Console

```
> library(MASS)
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A
	.	*	*	*	*
	.	*	*	*	*
	*	*	*	*	*
	*	*	*	*	*
Rubens	18	13	17	17	G
Teniers	15	12	13	6	G
Van Dyck	15	10	17	13	G
Bourdon	10	8	8	4	H
Le Brun	16	16	8	16	H

# Data Frames

However, these names are not variables of the data set. Here a subset of these names:

```
> rownames(painters)
[1] "Da Udine"          "Da Vinci"           "Del Piombo"
[4] "Del Sarto"         "Fr. Penni"          "Guilio Romano"
[7] "Michelangelo"      "Perino del Vaga"   "Perugino"
[10] "Raphael"          "F. Zuccaro"        "Fr. Salviata"
[13] "Parmigiano"       "Primaticcio"      "T. Zuccaro"
[16] "Volterra"         "Barocci"           "Cortona"
[19] "Josepin"          "L. Jordaeans"     "Testa"
[22] "Vanius"            "Bassano"           "Bellini"
[25] "Giorgione"        "Murillo"          "Palma Giovane"
[28] "Palma Vecchio"    "Pordenone"        "Tintoretto"
[31] "Titian"            "Veronese"          "Albani"
[34] "Caravaggio"        "Correggio"         "Domenichino"
[37] "Guercino"          "Lanfranco"        "The Carraci"
[40] "Durer"             "Holbein"           "Pourbus"
[43] "Van Leyden"        "Diepenbeck"        "J. Jordaeans"
[46] "Otho Venius"       "Rembrandt"         "Rubens"
[49] "Teniers"           "Van Dyck"          "Bourdon"
```

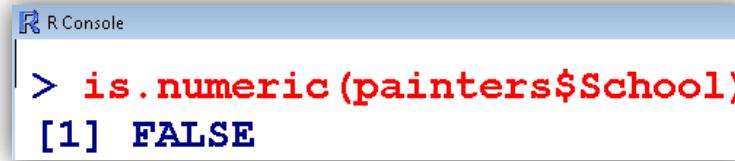
# Data Frames

```
R Console
> rownames(painters)
[1] "Da Udine"           "Da Vinci"          "Del Piombo"
[4] "Del Sarto"          "Fr. Penni"          "Guilio Romano"
[7] "Michelangelo"       "Perino del Vaga"   "Perugino"
[10] "Raphael"           "F. Zuccaro"        "Fr. Salviata"
[13] "Parmigiano"         "Primaticcio"      "T. Zuccaro"
[16] "Volterra"          "Barocci"           "Cortona"
[19] "Josepin"            "L. Jordaens"       "Testa"
[22] "Vanius"             "Bassano"           "Bellini"
[25] "Giorgione"          "Murillo"           "Palma Giovane"
[28] "Palma Vecchio"     "Pordenone"         "Tintoretto"
[31] "Titian"              "Veronese"          "Albani"
[34] "Caravaggio"          "Corregio"          "Domenichino"
[37] "Guercino"            "Lanfranco"         "The Carraci"
[40] "Durer"               "Holbein"           "Pourbus"
[43] "Van Leyden"          "Diepenbeck"        "J. Jordaens"
[46] "Otho Venius"         "Rembrandt"         "Rubens"
[49] "Teniers"             "Van Dyck"          "Bourdon"
```

# Data Frames

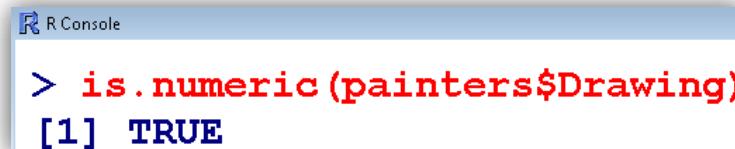
- The data set contains four numerical variables (Composition, Drawing, Colour and Expression), as well as one factor variable (School).

```
> is.numeric(painters$School)  
[1] FALSE
```



Notice how we extract a variable (column) from data set.

```
> is.numeric(painters$Drawing)  
[1] TRUE
```



# Data Frames

- The data set contains four numerical variables (Composition, Drawing, Colour and Expression), as well as one factor variable (School).

```
> is.factor(painters$School)
```

```
[1] TRUE
```

R R Console

```
> is.factor(painters$School)
[1] TRUE
```

```
> is.factor(painters$Drawing)
```

```
[1] FALSE
```

R R Console

```
> is.factor(painters$Drawing)
[1] FALSE
```

# Data Frames

```
> colnames(painters)
[1] "Composition" "Drawing" "Colour"
"Expression" "School"
```

R Console

```
> colnames(painters)
[1] "Composition" "Drawing"      "Colour"       "Expression"   "School"
```

# Data Frames

Using the **summary** function, we can get a quick overview of descriptive measures for each variable: (*We will learn later*).

```
> summary(painters)
```

Composition	Drawing	Colour	Expression	School
Min. : 0.00	Min. : 6.00	Min. : 0.00	Min. : 0.000	A : 10
1st Qu.: 8.25	1st Qu.: 10.00	1st Qu.: 7.25	1st Qu.: 4.000	D : 10
Median : 12.50	Median : 13.50	Median : 10.00	Median : 6.000	E : 7
Mean : 11.56	Mean : 12.46	Mean : 10.94	Mean : 7.667	G : 7
3rd Qu.: 15.00	3rd Qu.: 15.00	3rd Qu.: 16.00	3rd Qu.: 11.500	B : 6
Max. : 18.00	Max. : 18.00	Max. : 18.00	Max. : 18.000	C : 6
				(Other) : 8

The categories F and H, each present 4 times in the variable "School", are summed under the category Other as 8 with the corresponding frequency. i.e., only the 6 most frequent values are displayed.

# Data Frames

```
R Console
> summary(painters)
   Composition      Drawing       Colour      Expression     School
  Min.    : 0.00  Min.    : 6.00  Min.    : 0.00  Min.    : 0.000  A      :10
  1st Qu.: 8.25  1st Qu.:10.00  1st Qu.: 7.25  1st Qu.: 4.000  D      :10
  Median  :12.50  Median  :13.50  Median  :10.00  Median  : 6.000  E      : 7
  Mean    :11.56  Mean    :12.46  Mean    :10.94  Mean    : 7.667  G      : 7
  3rd Qu.:15.00  3rd Qu.:15.00  3rd Qu.:16.00  3rd Qu.:11.500  B      : 6
  Max.    :18.00  Max.    :18.00  Max.    :18.00  Max.    :18.000  C      : 6
                                         (Other) : 8
```

# **Introduction to R Software**

## **Data Frames**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Data Frames

An example data frame `painters` is available in the library MASS (here only an excerpt of a data set):

```
> library(MASS)
```

```
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.

Here, the names of the painters serve as row identifications, i.e., every row is assigned to the name of the corresponding painter.

# Data Frames

R Console

```
> library(MASS)
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A
	.	*	*	*	*
	.	*	*	*	*
	*	*	*	*	*
	*	*	*	*	*
Rubens	18	13	17	17	G
Teniers	15	12	13	6	G
Van Dyck	15	10	17	13	G
Bourdon	10	8	8	4	H
Le Brun	16	16	8	16	H

# Data Frames

- Test if we are dealing with a data frame:

```
> is.data.frame(painters)
[1] TRUE
```



A screenshot of an R console window titled "R Console". The window shows the command `> is.data.frame(painters)` in red text, followed by the output `[1] TRUE` in blue text.

```
R Console
> is.data.frame(painters)
[1] TRUE
```

# Data Frames

## □ Creating Data Frames

Use the `data.frame` function to create a data frame by adding column vectors to the data frame.

### Example:

```
> x <- 1:16 # Vector  
> y <- matrix(x, nrow=4, ncol=4) # 4 X 4 matrix  
> z <- letters[1:16] # lowercase alphabets  
  
> x  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
> y  
      [,1] [,2] [,3] [,4]  
[1,]    1     5     9    13  
[2,]    2     6    10    14  
[3,]    3     7    11    15  
[4,]    4     8    12    16  
> z  
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"  
[n] "o" "p"
```

# Data Frames

```
> datafr <- data.frame(x, y, z)
```

```
> datafr
```

	x	X1	X2	X3	X4	z
1	1	1	5	9	13	a
2	2	2	6	10	14	b
3	3	3	7	11	15	c
4	4	4	8	12	16	d
5	5	1	5	9	13	e
6	6	2	6	10	14	f
7	7	3	7	11	15	g
8	8	4	8	12	16	h
9	9	1	5	9	13	i
10	10	2	6	10	14	j
11	11	3	7	11	15	k
12	12	4	8	12	16	l
13	13	1	5	9	13	m
14	14	2	6	10	14	n
15	15	3	7	11	15	o
16	16	4	8	12	16	p

# Data Frames

```
R Console
> x <- 1:16
> y <- matrix(x, nrow=4, ncol=4)
> z <- letters[1:16]
> x
 [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
>
> y
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
>
> z
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[14] "n" "o" "p"
```

# Data Frames

```
R Console
> datafr <- data.frame(x, y, z)
> datafr
   x X1 X2 X3 X4 z
1 1  1  5  9 13 a
2 2  2  6 10 14 b
3 3  3  7 11 15 c
4 4  4  8 12 16 d
5 5  1  5  9 13 e
6 6  2  6 10 14 f
7 7  3  7 11 15 g
8 8  4  8 12 16 h
9 9  1  5  9 13 i
10 10 2  6 10 14 j
11 11 3  7 11 15 k
12 12 4  8 12 16 l
13 13 1  5  9 13 m
14 14 2  6 10 14 n
15 15 3  7 11 15 o
16 16 4  8 12 16 p
```

# Data Frames

## □ Structure of the data:

Display information about the structure of the data frame (`str`).

The result of `str` gives the dimension as well as the name and type of each variable.

```
> str(painters)

'data.frame' : 54 obs. of 5 variables:

$ Composition: int 10 15 8 12 0 15 8 15 4 17 ...
$ Drawing     : int 8 16 13 16 15 16 17 16 12 18 ...
$ Colour      : int 16 4 16 9 8 4 4 7 10 12 ...
$ Expression  : int 3 14 7 8 0 14 8 6 4 18 ...
$ School      : Factor w/ 8 levels "A","B","C","D",...: 1
                                         1 1 1 1 1 1 1 1 1 ...
```

`int` means integer.

# Data Frames

```
R Console
> str(painters)
'data.frame': 54 obs. of 5 variables:
 $ Composition: int 10 15 8 12 0 15 8 15 4 17 ...
 $ Drawing    : int 8 16 13 16 15 16 17 16 12 18 ...
 $ Colour     : int 16 4 16 9 8 4 4 7 10 12 ...
 $ Expression : int 3 14 7 8 0 14 8 6 4 18 ...
 $ School     : Factor w/ 8 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1 ...
```

# Data Frames

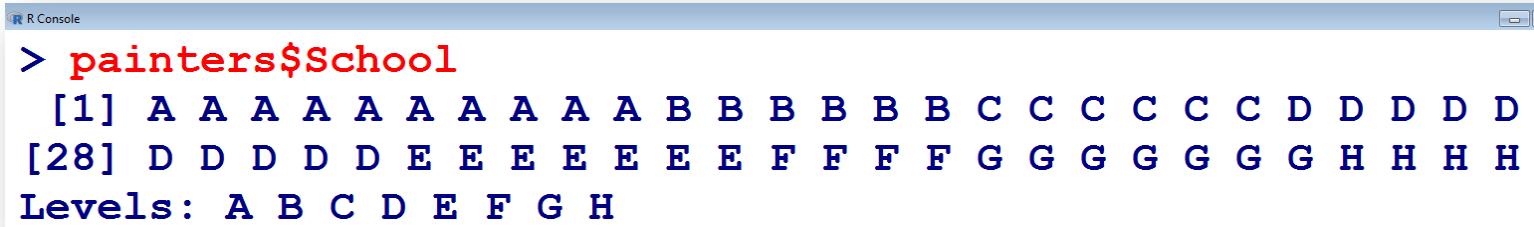
- Extract a variable from data frame using \$

Variables can be extracted using the \$ operator followed by the name of the variable.

**Example:** Suppose we want to extract information on variable School from the data set painters.

painters\$School

```
[1] A A A A A A A A A A B B B B B C C C C C C D D D D D  
[28] D D D D D E E E E E F F F F G G G G G G G H H H H  
Levels: A B C D E F G H
```



R Console

```
> painters$School
[1] A A A A A A A A A A B B B B B C C C C C C D D D D D
[28] D D D D D E E E E E F F F F G G G G G G G H H H H
Levels: A B C D E F G H
```

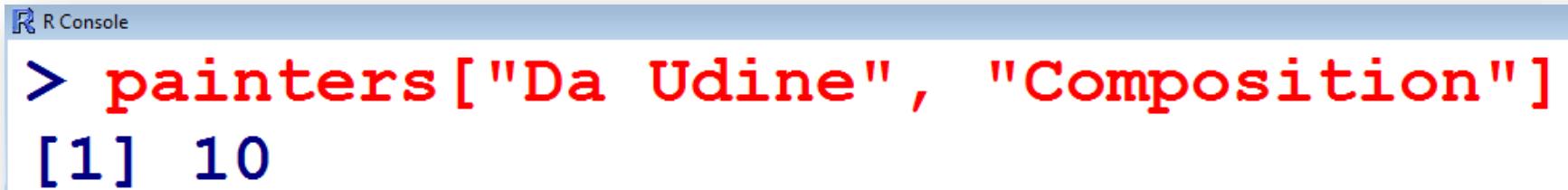
# Data Frames

## □ Extract data from a data frame

The data from a data frame can be extracted by using the matrix-style `[row, column]` indexing.

Example: Suppose we want to extract information on the first painter `Da Udine` on the variable `Composition` from the data set `painters`.

```
> painters["Da Udine", "Composition"]  
[1] 10
```



A screenshot of an R console window titled "R Console". The window shows the command `> painters["Da Udine", "Composition"]` in red, indicating it is the current input line. Below the command, the output is shown in blue: `[1] 10`. The background of the console is light blue.

# Data Frames

The **summary** function for a categorical variable returns a detailed frequency table:

```
> summary(painters$School)
 A   B   C   D   E   F   G   H
10  6   6  10  7   4   7   4
```

R Console

```
> summary(painters$School)
 A   B   C   D   E   F   G   H
10  6   6  10  7   4   7   4
```

*We will learn later:*

**summary** is a generic function used to produce result summaries of the results of various model fitting functions.

# Data Frames

The **summary** function for a numeric variable returns an overview of descriptive measures for each variable: (*We will learn later*).

```
> summary(painters)
```

Composition	Drawing	Colour	Expression	School
Min. : 0.00	Min. : 6.00	Min. : 0.00	Min. : 0.000	A : 10
1st Qu.: 8.25	1st Qu.: 10.00	1st Qu.: 7.25	1st Qu.: 4.000	D : 10
Median : 12.50	Median : 13.50	Median : 10.00	Median : 6.000	E : 7
Mean : 11.56	Mean : 12.46	Mean : 10.94	Mean : 7.667	G : 7
3rd Qu.: 15.00	3rd Qu.: 15.00	3rd Qu.: 16.00	3rd Qu.: 11.500	B : 6
Max. : 18.00	Max. : 18.00	Max. : 18.00	Max. : 18.000	C : 6
				(Other) : 8

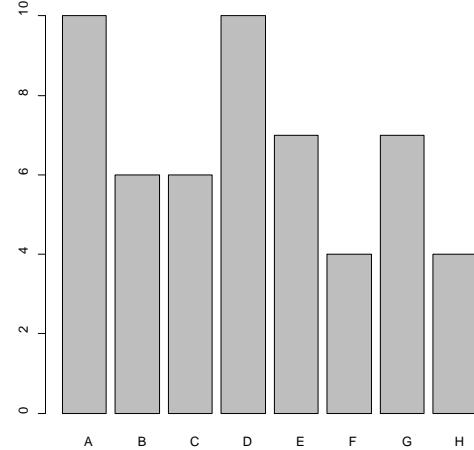
# Data Frames

```
R Console
> summary(painters)
   Composition      Drawing       Colour      Expression     School
  Min.    : 0.00  Min.    : 6.00  Min.    : 0.00  Min.    : 0.000  A      :10
  1st Qu.: 8.25  1st Qu.:10.00  1st Qu.: 7.25  1st Qu.: 4.000  D      :10
  Median  :12.50  Median  :13.50  Median  :10.00  Median  : 6.000  E      : 7
  Mean    :11.56  Mean    :12.46  Mean    :10.94  Mean    : 7.667  G      : 7
  3rd Qu.:15.00  3rd Qu.:15.00  3rd Qu.:16.00  3rd Qu.:11.500  B      : 6
  Max.    :18.00  Max.    :18.00  Max.    :18.00  Max.    :18.000  C      : 6
                                         (Other) : 8
```

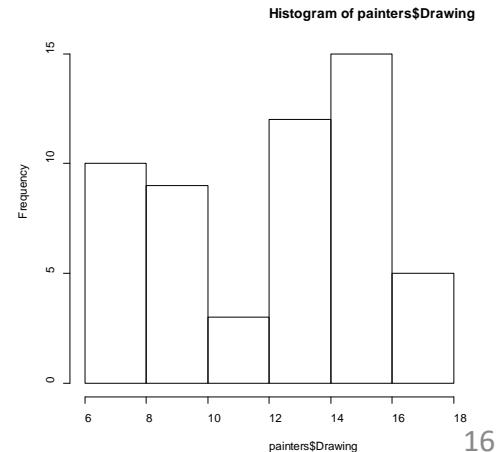
# Data Frames

□ Plot and graphics of the data

> `plot(painters$School) #factor variable`



> `hist(painters$Drawing) #numeric variable`



# **Introduction to R Software**

## **Data Frames**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Data Frames

An example data frame `painters` is available in the library MASS (here only an excerpt of a data set):

```
> library(MASS)
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.

Here, the names of the painters serve as row identifications, i.e., every row is assigned to the name of the corresponding painter.

# Data Frames

R Console

```
> library(MASS)
> painters
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A
	.	*	*	*	*
	.	*	*	*	*
	*	*	*	*	*
	*	*	*	*	*
Rubens	18	13	17	17	G
Teniers	15	12	13	6	G
Van Dyck	15	10	17	13	G
Bourdon	10	8	8	4	H
Le Brun	16	16	8	16	H

# Data Frames

## □ Attaching a data frame

With a command `attach( )` over the data frame, the variables can be referenced directly by name.

It can address the names of a data frame directly, without the prefix dollar sign operator, e.g. `painters$`.

### Example

```
> attach(painters)
```

Variable names are

- `Composition`,
- `Drawing`,
- `Colour`,
- `Expression`,
- `School`

# Data Frames

```
> summary(School) # Character variable
```

	A	B	C	D	E	F	G	H
10	6	6	10	7	4	7	4	7

R Console

```
> attach(painters)  
> summary(School)
```

	A	B	C	D	E	F	G	H
10	6	6	10	7	4	7	4	7

```
> summary(Composition) # Numeric variable
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.00	8.25	12.50	11.56	15.00	18.00

R Console

```
> summary(Composition)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.00	8.25	12.50	11.56	15.00	18.00

# Data Frames

- The command `detach()` recovers the default setting and then we have to use `painters$` again.

```
> detach(painters)
```

```
> summary(School)
```

```
Error in summary(School) : Object "School" not found
```

R Console

```
> detach(painters)
```

```
> summary(School)
```

```
Error in summary(School) : object 'School' not found
```

# Data Frames

Subsets of a data frame can be obtained with `subset()` or with the second equivalent command:

```
> subset(painters, School=='F')
```

(# == means logical equal sign )

	Composition	Drawing	Colour	Expression	School
Durer	8	10	10	8	F
Holbein	9	10	16	13	F
Pourbus	4	15	6	6	F
VanLeyden	8	6	6	4	F

# Data Frames

Similar outcome can be also obtained from

```
> painters[ painters[["School"]] == "F", ]
```

	Composition	Drawing	Colour	Expression	School
Durer	8	10	10	8	F
Holbein	9	10	16	13	F
Pourbus	4	15	6	6	F
VanLeyden	8	6	6	4	F

R R Console

```
> painters[ painters[["School"]] == "F", ]
```

	Composition	Drawing	Colour	Expression	School
Durer	8	10	10	8	F
Holbein	9	10	16	13	F
Pourbus	4	15	6	6	F
Van Leyden	8	6	6	4	F

# Data Frames

Subsets of a data frame can be obtained with `subset()` or with the second equivalent command:

```
> subset(painters, Composition <= 6)
```

```
R Console

> subset(painters, Composition <= 6)
   Composition Drawing Colour Expression School
Fr. Penni          0      15      8          0     A
Perugino          4      12     10          4     A
Bassano           6       8     17          0     D
Bellini           4       6     14          0     D
Murillo           6       8     15          4     D
Palma Vecchio    5       6     16          0     D
Caravaggio        6       6     16          0     E
Pourbus           4      15      6          6     F
> |
```

# Data Frames

- Uninteresting columns can be eliminated.

```
> subset(painters, School=="F", select=c(-3,-5))
```

	Composition	Drawing	Expression
Durer	8	10	8
Holbein	9	10	13
Pourbus	4	15	6
Van Leyden	8	6	4

The third and the fifth column (Colour and School) are not shown.

# Data Frames

R R Console

```
> subset(painters, School=="F", select=c(-3,-5))
```

	Composition	Drawing	Expression
Durer	8	10	8
Holbein	9	10	13
Pourbus	4	15	6
Van Leyden	8	6	4

```
> |
```

# Data Frames

- The command `split` partitions the data set by values of a specific variable. This should preferably be a factor variable.

**Example:** Following command splits `painters` with respect to `School` (A,B,C,... categories)

```
> splitted <- split(painters, painters$School)
```

# Data Frames

> splitted

\$A

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A
Michelangelo	8	17	4	8	A
Perino del Vaga	15	16	7	6	A
Perugino	4	12	10	4	A
Raphael	17	18	12	18	A

R Console

Contd...

```
> splitted <- split(painters, painters$School)
> splitted
$A
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A
Michelangelo	8	17	4	8	A

# Data Frames

\$B

	Composition	Drawing	Colour	Expression	School
F. Zuccaro	10	13	8	8	B
Fr. Salviata	13	15	8	8	B
Parmigiano	10	15	6	6	B
Primaticcio	15	14	7	10	B
T. Zuccaro	13	14	10	9	B
Volterra	12	15	5	8	B

Contd...

R Console

```
> splitted $B
```

	Composition	Drawing	Colour	Expression	School
F. Zuccaro	10	13	8	8	B
Fr. Salviata	13	15	8	8	B
Parmigiano	10	15	6	6	B
Primaticcio	15	14	7	10	B
T. Zuccaro	13	14	10	9	B
Volterra	12	15	5	8	B

# Data Frames

\$C

	Composition	Drawing	Colour	Expression	School
Barocci	14	15	6	10	C
Cortona	16	14	12	6	C
Josepin	10	10	6	2	C
L. Jordaens	13	12	9	6	C
Testa	11	15	0	6	C
Vanius	15	15	12	13	C

Contd...

R Console

```
> splitted $C
```

	Composition	Drawing	Colour	Expression	School
Barocci	14	15	6	10	C
Cortona	16	14	12	6	C
Josepin	10	10	6	2	C
L. Jordaens	13	12	9	6	C
Testa	11	15	0	6	C
Vanius	15	15	12	13	C

# Data Frames

Contd...

...

\$H

	Composition	Drawing	Colour	Expression	School
Bourdon	10	8	8	4	H
Le Brun	16	16	8	16	H
Le Sueur	15	15	4	15	H
Poussin	15	17	6	15	H

```
R Console
> splitted $H
      Composition Drawing Colour Expression School
Bourdon          10       8     8        4      H
Le Brun           16      16     8       16      H
Le Sueur          15      15     4       15      H
Poussin          15      17     6       15      H
```

Remark: If the data set is not attached, we have to use  
`painters$School.`

# Data Frames

The objects  `splitted$A` to  `splitted$H` are themselves data frames:

```
> is.data.frame(splitted$A)
[1] TRUE
```



R Console

```
> is.data.frame(splitted$A)
[1] TRUE
```

# **Introduction to R Software**

## **Data Handling**

**::::**

## **Importing CSV and Tabular Data Files**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# Setting up directories

- We can change the current working directory as follows:

```
> setwd( "<location of the dataset>" )
```

## Example:

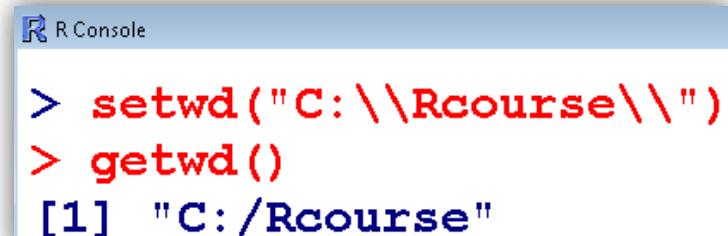
```
> setwd( "C:/Rcourse/" )
```

or

```
> setwd( "C:\\Rcourse\\\\" )
```

- The following command returns the current working directory:

```
> getwd()  
[1] "C:/Rcourse/"
```



A screenshot of an R console window titled "R Console". The window shows the following R session:

```
R Console  
> setwd("C:\\Rcourse\\\\")  
> getwd()  
[1] "C:/Rcourse"
```

# **Importing Data Files**

Suppose we have some data on our computer and we want to import it in R.

Different formats of files can be read in R

- comma-separated values (CSV) data file,
- table file (TXT),
- Spreadsheet (e.g., MS Excel) file,
- files from other software like SPSS, Minitab etc.

## Importing Data Files

One can also read or upload the file from Internet site.

We can read the file containing rent index data from website:

<http://home.iitk.ac.in/~shalab/Rcourse/munichdata.asc>

as follows

```
> datamunich <- read.table(file="http://home.iitk.ac.in/~shalab/Rcourse/munichdata.asc", header=TRUE)
```

File name is **munichdata.asc**

# Importing Data Files

## Comma-separated values (CSV) files

First set the working directory where the CSV file is located.

```
setwd("location of your dataset")
```

```
> setwd("C:/RCourse/")
```



To read a CSV file

Syntax: `read.csv("filename.csv")`

Example:

```
> data <- read.csv("example1.csv")
```

# Importing Data Files

## Comma-separated values (CSV) files

Example:

```
> data <- read.csv("example1.csv")
> data
  X1  X10  X100
1  2    20   200
2  3    30   300
3  4    40   400
4  5    50   500
```

	A	B	C	D
1	1	10	100	
2	2	20	200	
3	3	30	300	
4	4	40	400	
5	5	50	500	
6				

R Console

```
> setwd("C:/RCourse")
> data <- read.csv("example1.csv")
> data
  X1  X10  X100
1  2    20   200
2  3    30   300
3  4    40   400
4  5    50   500
```

Notice the difference in the first rows of excel file and output

# Importing Data Files

## Comma-separated values (CSV) files

Data files have many formats and accordingly we have options for loading them.

If the data file does not have headers in the first row, then use

```
data <- read.csv("datafile.csv", header=FALSE)
```

# Importing Data Files

Comma-separated values (CSV) data

Example:

```
> data <- read.csv("example1.csv", header=FALSE)
```

```
> data  
  v1  v2  v3  
1  1  10 100  
2  2  20 200  
3  3  30 300  
4  4  40 400  
5  5  50 500
```

	A	B	C	D
1	1	10	100	
2	2	20	200	
3	3	30	300	
4	4	40	400	
5	5	50	500	
6				
7				

```
R Console  
> data <- read.csv("example1.csv", header=FALSE)  
> data  
  v1  v2  v3  
1  1  10 100  
2  2  20 200  
3  3  30 300  
4  4  40 400  
5  5  50 500
```

# Importing Data Files

Comma-separated values (CSV) files

The resulting data frame will have columns named V1, V2, ...

We can rename the header names manually:

```
> names(data) <-c("Column1", "Column2", "Column3")
```

```
> data
   Column1 Column2 Column3
1      1      10     100
2      2      20     200
3      3      30     300
4      4      40     400
5      5      50     500
```

# Importing Data Files

## Comma-separated values (CSV) files

```
R Console
> data <- read.csv("example1.csv", header=FALSE)
> data
  V1  V2  V3
1  1  10 100
2  2  20 200
3  3  30 300
4  4  40 400
5  5  50 500
> names(data) <-c("Column1","Column2","Column3")
> data
  Column1 Column2 Column3
1        1      10     100
2        2      20     200
3        3      30     300
4        4      40     400
5        5      50     500
```

# Importing Data Files

## Comma-separated values (CSV) files

We can set the delimiter with `sep`.

If it is tab delimited, use `sep="\t"`.

```
data <- read.csv("datafile.csv", sep="\t")
```

If it is space-delimited, use `sep=" "`.

```
data <- read.csv("datafile.csv", sep=" ")
```

# Importing Data Files

## Reading Tabular Data Files

Tabular data files are text files with a simple format:

- Each line contains one record.
- Within each record, fields (items) are separated by a one-character delimiter, such as a space, tab, colon, or comma.
- Each record contains the same number of fields.

We want to read a text file that contains a table of data.

**read.table** function is used and it returns a data frame.

```
read.table("FileName")
```

# Importing Data Files

## Reading Tabular Data Files

Data:

1 10 100

2 20 200

3 30 300

4 40 400

5 50 500

Saved in example3.txt

```
> data <- read.table("example3.txt", sep=" ")  
> data
```

	v1	v2	v3
1	1	10	100
2	2	20	200
3	3	30	300
4	4	40	400
5	5	50	500

# Importing Data Files

## Reading Tabular Data Files

R RGui (32-bit)

```
> data <- read.table("example3.txt", sep=" ")  
> data  
   V1  V2  V3  
1  1  10 100  
2  2  20 200  
3  3  30 300  
4  4  40 400  
5  5  50 500
```

# **Introduction to R Software**

## **Data Handling**

::::

### **Importing Data Files of Other Software and Redirecting Output**

**Shalabh**

**Department of Mathematics and Statistics  
Indian Institute of Technology Kanpur**

# Importing Data Files

## Spreadsheet (Excel) file data

The `xlsx` package has the function `read.xlsx()` for reading Excel files.

This will read the first sheet of an Excel spreadsheet.

To read Excel files, we first need to install the package

```
install.packages("xlsx")
```

```
library(xlsx)
```

```
data <- read.xlsx("datafile.xlsx", Sheet Index  
or Sheet Name )
```

# Importing Data Files

## Spreadsheet (Excel) file data

To load other sheets with `read.xlsx()`, we specify a number for `sheetIndex` or a name for `sheetName`:

```
data <- read.xlsx("datafile.xlsx", sheetIndex=2)
```

```
data <- read.xlsx("datafile.xlsx",  
sheetName="marks")
```

# **Importing Data Files**

## **Spreadsheet (Excel) file data**

**For reading older Excel files in .xls format, use `gdata` package and function `read.xls()`**

**This will read the first sheet of an Excel spreadsheet.**

**To read Excel files, we first need to install the package**

```
install.packages("gdata")
```

```
library(gdata)
```

```
data <- read.xls("datafile.xls", Sheet Index or  
Sheet Name ))
```

# Importing Data Files

SPSS data file

For reading SPSS data files, use **foreign** package and function

**read.spss( )**

To read SPSS files, we first need to install the package

```
install.packages(" foreign ")
```

```
library(foreign)
```

```
data <- read.spss("datafile.sav")
```

# Importing Data Files

## Other data files

The `foreign` package also includes functions to load from other formats, including:

- `read.octave("<Path to file>")`: Octave and MATLAB
- `read.systat("<Path to file>")`: SYSTAT
- `read.xport("<Path to file>")`: SAS XPORT
- `read.dta("<Path to file>")`: Stata

## **Importing Data Files**

More description of data import and export can be found in the respective R manual at

<http://cran.r-project.org/doc/manuals/r-release/R-data.pdf>

# Contents of working directory

The `list.files` function shows the contents of your working directory:

```
> list.files()  
  
> setwd("C:/RCourse/")  
  
> list.files()  
  
[1] "~$example3.xlsx"      "example1.csv"        "example2.txt"  
"example3.xlsx"  
  
[5] "marks.csv"           "munichdata.asc"  
  
"pizza_delivery.csv"
```



R Console

```
> list.files()  
[1] "~$example3.xlsx"      "example1.csv"        "example2.txt"        "example3.xlsx"  
[5] "marks.csv"           "munichdata.asc"     "pizza_delivery.csv"
```

# Redirecting Output to a File

## Issue:

We want to redirect the output from R into a file instead of your console.

## Solution:

Redirect the output of the `cat` function by using its `file` argument:

```
> ans <- 6 + 8  
> cat("The answer of 6 + 8 is", ans, "\n",  
file="filename")
```

The output will be saved in the working directory with given filename

## Redirecting Output to a File

Use the `sink` function to redirect all the output from both `print` and `cat`.

Call `sink` with a filename argument to begin redirecting console output to that file.

When we are done, `sink` with no argument to close the file and resume output to the console:

```
> sink("filename") #Begin writing output to file  
.  
.  
. other session work .  
> sink()
```

## Redirecting Output to a File

The `print` and `cat` functions normally write the output to console.

The `cat` function writes to a file if we supply a file argument.

The `print` function cannot redirect its output.

The `sink` function can force all output to a file.

# Redirecting Output to a File: Three steps

1.

```
> sink("output.txt") # Redirect output to file
```

2.

```
> source("script.R") # Run the script, capture  
its output
```

3.

```
> sink() # Resume writing output to console
```

Other options like `append=TRUE/FALSE`, `split=TRUE/FALSE` are available.

## Example:

Find the mean of all the three variables in the data set

`example1.csv`

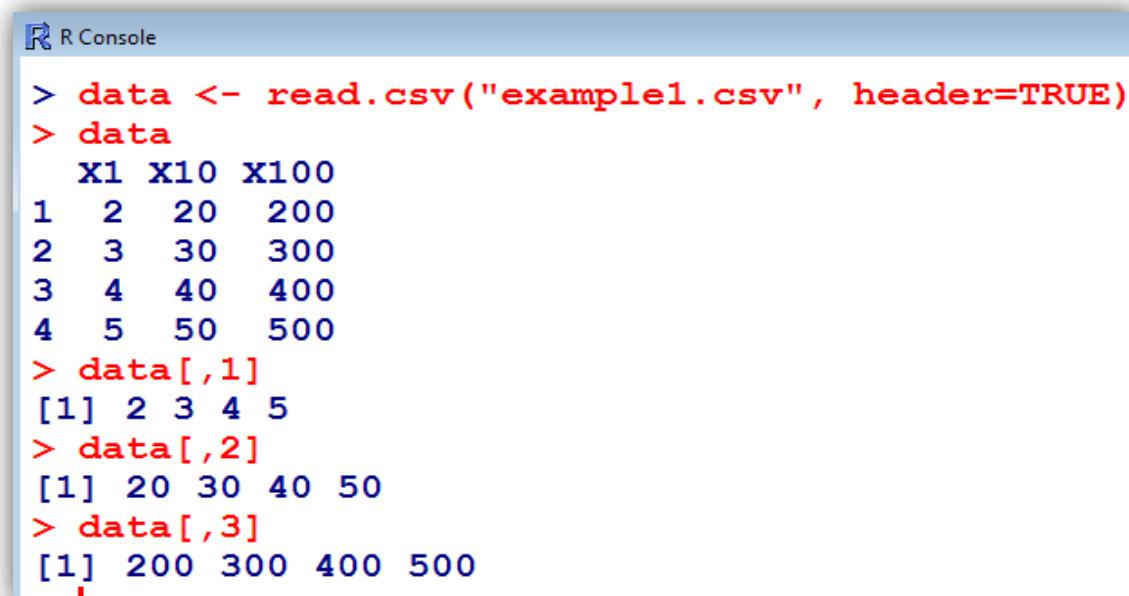
```
setwd("C:/RCourse/")
```

```
data <- read.csv("example1.csv", header=TRUE)
```

```
> data[,1]  
[1] 2 3 4 5
```

```
> data[,2]  
[1] 20 30 40 50
```

```
> data[,3]  
[1] 200 300 400 500
```



The screenshot shows the R Console window with the title "R Console". It displays the following R session:

```
R Console  
> data <- read.csv("example1.csv", header=TRUE)  
> data  
  X1  X10 X100  
1   2   20  200  
2   3   30  300  
3   4   40  400  
4   5   50  500  
> data[,1]  
[1] 2 3 4 5  
> data[,2]  
[1] 20 30 40 50  
> data[,3]  
[1] 200 300 400 500  
>
```

The data frame has four rows and three columns. The first column is labeled "X1" and contains values 2, 3, 4, 5. The second column is labeled "X10" and contains values 20, 30, 40, 50. The third column is labeled "X100" and contains values 200, 300, 400, 500.

# Example:

Programme:

```
meanxyz <- function(data)
{
  meanofdata <- 0
  for (i in 1:3)
  {
    meanofdata[i] <- mean(data[,i])
    cat("The mean of X", i, "is", meanofdata[i], ".", "\n")
  }
}
meanxyz(data)
```

Save it as script, say `meanxyz.R`

## Example:

```
> sink("output_meanxyz.txt") # Creates a blank file
```

(Open the file and check it – a blank file will be there)

```
> source("meanxyz.R") # Writes output inside the file
```

Or run the programme as

```
> meanxyz(data)
```

(Open the file and check it – a file with the output will be there)

```
> sink() # Resume writing output to console
```

## **Output:**

**Open the directory "C:/RCourse/" .**

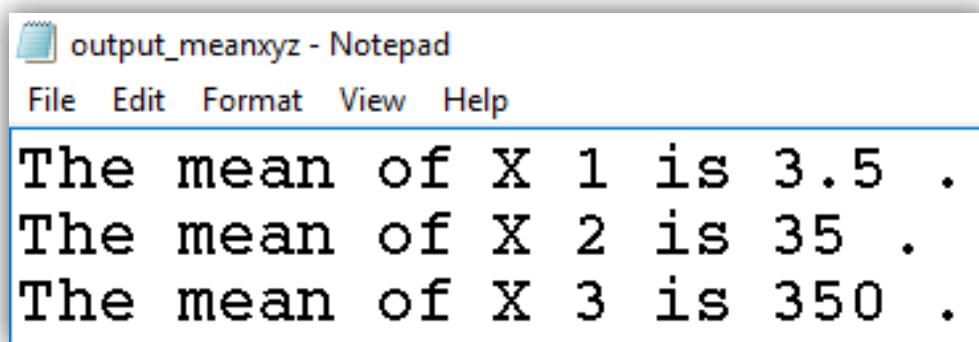
**Find a file `output_meanxyz.txt`**

**Open it and we find the following output**

**The mean of X 1 is 3.5 .**

**The mean of X 2 is 35 .**

**The mean of X 3 is 350 .**



A screenshot of a Windows Notepad window titled "output\_meanxyz - Notepad". The window contains the following text:

```
The mean of X 1 is 3.5 .
The mean of X 2 is 35 .
The mean of X 3 is 350 .
```

```
> meanxyz <- function(data)
+ {
+   meanofdata <- 0
+   for (i in 1:3)
+   {
+     meanofdata[i]<-mean(data[,i])
+     cat("The mean of X",i, "is", meanofdata[i], ".", "\n")
+   }
+ }
>
> meanxyz
function(data)
{
  meanofdata <- 0
  for (i in 1:3)
  {
    meanofdata[i]<-mean(data[,i])
    cat("The mean of X",i, "is", meanofdata[i], ".", "\n")
  }
}
>
> meanxyz(data)
The mean of X 1 is 3.5 .
The mean of X 2 is 35 .
The mean of X 3 is 350 .
>
> sink("output_meanxyz.txt")
> source("meanxyz.R")
```

## Writing to CSV files

Suppose we want to save a matrix or data frame in a file using the comma-separated values format.

The `write.csv` function writes tabular data to an ASCII file in CSV format.

Each row of data creates one line in the file, with data items separated by commas (,):

```
> write.csv(x, file="filename", row.names=FALSE)
```

Example:

```
> write.csv( meanxyz(data),  
file="output_meanxyz.csv", row.names=FALSE )
```

Check working directory, file `output_meanxyz.csv` is created.

# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

## **Introduction, Frequencies and Partition Values**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Descriptive statistics:**

**First hand tools which gives first hand information.**

- **Central tendency of data**
- **Variation in data**
- **Structure and shape of data tendency**
- **Relationship study**

**Graphical as well as analytical tools are used.**

# **Graphical tools:**

## **Graphical tools- various type of plots**

- 2D & 3D plots,**
- scatter diagram**
- Pie diagram**
- Histogram**
- Bar plot**
- Stem and leaf plot**
- Box plot ...**

## Absolute and relative frequencies:

Suppose there are 10 persons coded into two categories as male (M) and female (F).

M, F, M, F, M, M, M, F, M, M.

Use  $a_1$  and  $a_2$  to refer to male and female categories.

There are 7 male and 3 female persons,

denoted as  $n_1 = 7$  and  $n_2 = 3$

The number of observations in a particular category is called the absolute frequency.

## Absolute and relative frequencies:

The relative frequencies of  $a_1$  and  $a_2$  are

$$f_1 = \frac{n_1}{n_1 + n_2} = \frac{7}{10} = 0.7 = 70\%$$

$$f_2 = \frac{n_2}{n_1 + n_2} = \frac{3}{10} = 0.3 = 30\%$$

This gives us information about the proportions of male and female persons.

## Absolute and relative frequencies:

`table(variable)` creates the absolute frequency of the `variable` of the data file.

Enter data as `x`

`table(x) # absolute frequencies`

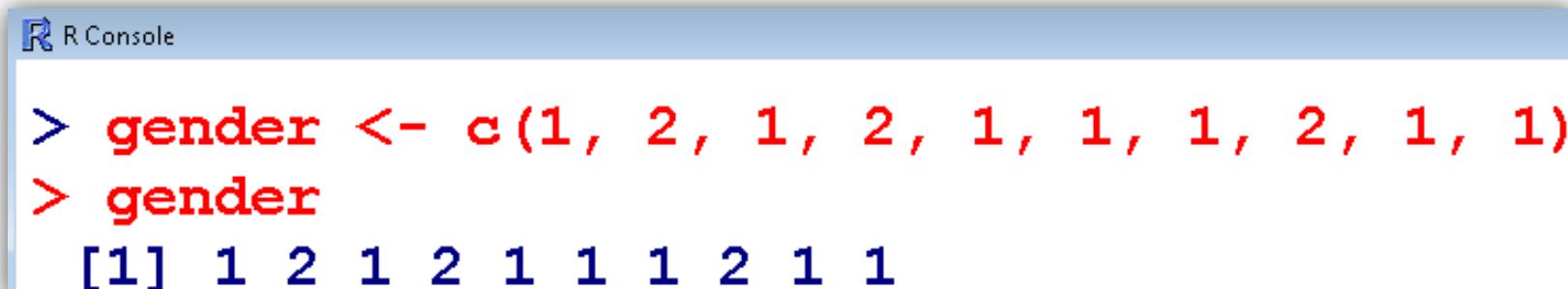
`table(x)/length(x) # relative frequencies`

## Absolute and relative frequencies:

Example: Code the 10 persons by using, say 1 for male (M) and 2 for female (F).

M,	F,	M,	F,	M,	M,	M,	F,	M,	M
1,	2,	1,	2,	1,	1,	1,	2,	1,	1

```
> gender <- c(1, 2, 1, 2, 1, 1, 1, 2, 1, 1)  
  
> gender  
[1] 1 2 1 2 1 1 1 2 1 1
```



R Console

```
> gender <- c(1, 2, 1, 2, 1, 1, 1, 2, 1, 1)  
> gender  
[1] 1 2 1 2 1 1 1 2 1 1
```

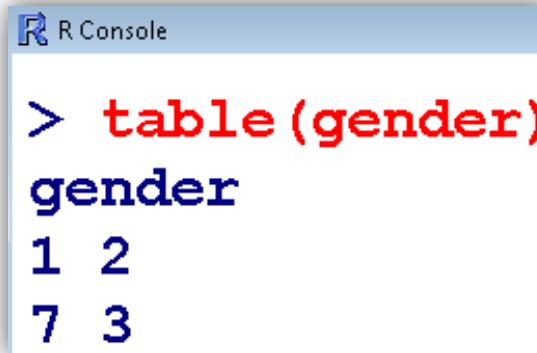
## Absolute and relative frequencies:

```
> table(gender) # Absolute frequencies
```

```
gender
```

```
1 2
```

```
7 3
```



R Console window showing the output of the command `> table(gender)`. The output is a table with two rows, labeled "gender". The first row contains "1" and "2". The second row contains "7" and "3".

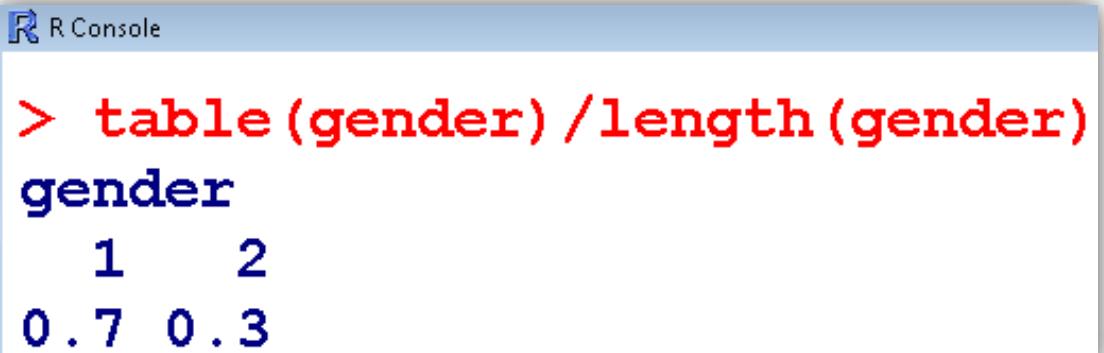
gender	1	2
7	3	

```
> table(gender)/length(gender) #Relative freq.
```

```
gender
```

```
1 2
```

```
0.7 0.3
```



R Console window showing the output of the command `> table(gender)/length(gender)`. The output is a table with two rows, labeled "gender". The first row contains "1" and "2". The second row contains "0.7" and "0.3".

gender	1	2
	0.7	0.3

## Example:

'**pizza\_delivery.csv**' contains the simulated data on pizza home delivery.

- There are three branches (East, West, Central) of the restaurant.
- The pizza delivery is centrally managed over phone and delivered by one of the five drivers.
- The data set captures the number of pizzas ordered and the final bill

```
> setwd( "C:/Rcourse" )  
> pizza <- read.csv('pizza_delivery.csv' )
```

## Example:

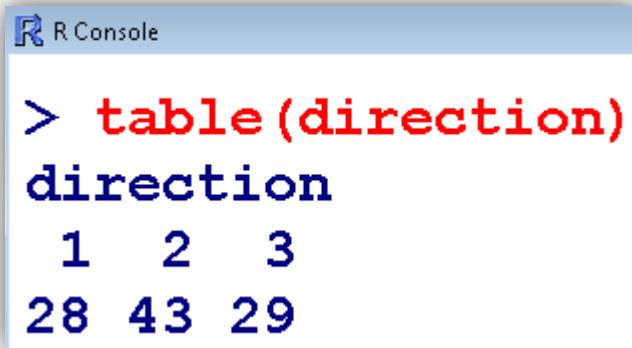
Consider data from Pizza. Take first 100 values from Direction and code Directions as

- ❖ East: 1
- ❖ West: 2
- ❖ Centre: 3

```
direction <-c(1,1,2,1,2,3,2,2,3,3,3,1,2,3,2,2,3,1,  
1,3,3,1,2,1,3,3,2,2,2,2,1,2,2,1,1,1,3,2,2,1,2,3,2  
,2,1,2,3,3,2,1,2,2,3,1,1,2,1,2,3,2,3,2,2,3,1,2,3,3,  
3,2,1,1,1,2,1,1,2,1,2,3,3,1,2,3,3,2,1,2,3,2,1,3,2,2  
,2,2,3,2,2)
```

## Example:

```
> table(direction)
direction
 1 2 3
28 43 29
```



A screenshot of an R console window titled "R R Console". The window contains the following text:

```
> table(direction)
direction
 1 2 3
28 43 29
```

## Example:

```
> table(direction)/length(direction)
direction
  1    2    3
0.28 0.43 0.29
```

R R Console

```
> table(direction)/length(direction)
direction
  1    2    3
0.28 0.43 0.29
```

## **Partition values:**

Such values divides the total frequency given data into required number of partitions.

**Quartile:** Divides the data into 4 equal parts.

**Decile:** Divides the data into 10 equal parts.

**Percentile:** Divides the data into 100 equal parts.

## Partition values:

**quantile** function computes quantiles corresponding to the given probabilities.

The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

**quantile(x, ...)**

**quantile(x, probs = seq(0, 1, 0.25), ...)**

### Arguments

- x** numeric vector whose sample quantiles are wanted,
- probs** numeric vector of probabilities with values in [0, 1].

## Partition values:

Example: Marks of 15 students are

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
29, 51, 75, 77, 56, 59, 42)
```

```
> quantile(marks)  
 0% 25% 50% 75% 100%  
29.0 46.5 63.0 79.5 96.0
```

```
R Console  
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89, 29, 51, 75, 77, 56, 59, 42)  
> marks  
[1] 68 82 63 86 34 96 41 89 29 51 75 77 56 59 42  
> quantile(marks)  
 0% 25% 50% 75% 100%  
29.0 46.5 63.0 79.5 96.0
```

## Partition values:

Example: Marks of 15 students are

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
29, 51, 75, 77, 56, 59, 42)  
  
> quantile(marks, probs=c(0,0.25,0.5,0.75,1))  
0% 25% 50% 75% 100%  
29.0 46.5 63.0 79.5 96.0
```

## Default values

```
R R Console  
  
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89, 29, 51, 75, 77, 56, 59, 42)  
> marks  
[1] 68 82 63 86 34 96 41 89 29 51 75 77 56 59 42  
>  
> quantile(marks, probs=c(0,0.25,0.5,0.75,1))  
0% 25% 50% 75% 100%  
29.0 46.5 63.0 79.5 96.0  
>
```

## Partition values:

Example: Marks of 15 students are

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89, 29,  
 51, 75, 77, 56, 59, 42)
```

Defining probabilities

```
> quantile(marks, probs=c(0,0.20,0.4,0.6,0.8,1))  
 0%   20%   40%   60%   80% 100%  
29.0  41.8  57.8  70.8  82.8 96.0
```

```
R Console  
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89, 29, 51, 75, 77, 56, 59, 42)  
> marks  
[1] 68 82 63 86 34 96 41 89 29 51 75 77 56 59 42  
> quantile(marks, probs=c(0,0.20,0.4,0.6,0.8,1))  
 0%   20%   40%   60%   80% 100%  
29.0  41.8  57.8  70.8  82.8 96.0
```

# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

## **Graphics and Plots**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

# **Graphical tools:**

## **Graphical tools- various type of plots**

- 2D & 3D plots,
- scatter diagram
- Pie diagram
- Histogram
- Bar plot
- Stem and leaf plot
- Box plot ...

**Appropriate number and choice of plots in analysis provides better inferences.**

## **Graphical tools:**

In R, Such graphics can be easily created and saved in various formats.

- **Bar plot**
- **Pie chart**
- **Box plot**
- **Grouped box plot**
- **Scatter plot**
- **Coplots**
- **Histogram**
- **Normal QQ plot ...**

## Bar plots:

Visualize the relative or absolute frequencies of observed values of a variable.

It consists of one bar for each category.

The height of each bar is determined by either the absolute frequency or the relative frequency of the respective category and is shown on the y-axis.

```
barplot(x, width = 1, space = NULL,...)
```

```
> barplot(table(x))
```

```
> barplot(table(x)/length(x))
```

## Bar plots:

```
> help("barplot")
```

```
barplot(height, width = 1, space = NULL,  
names.arg = NULL, legend.text = NULL, beside  
= FALSE, horiz = FALSE, density = NULL, angle  
= 45, col = NULL, border = par("fg"), main =  
NULL, sub = NULL, xlab = NULL, ylab = NULL,  
xlim = NULL, ylim = NULL, xpd = TRUE, log =  
"", axes = TRUE, axisnames = TRUE, cex.axis =  
par("cex.axis"), cex.names = par("cex.axis"),  
inside = TRUE, plot = TRUE, axis.lty = 0,  
offset = 0, add = FALSE, args.legend = NULL,  
...)
```

## Example:

Code the 10 persons by using, say 1 for male (M) and 2 for female (F).

M, F, M, F, M, M, M, F, M, M  
1, 2, 1, 2, 1, 1, 1, 2, 1, 1

```
> gender <- c(1, 2, 1, 2, 1, 1, 1, 2, 1, 1)  
  
> gender  
[1] 1 2 1 2 1 1 1 2 1 1
```



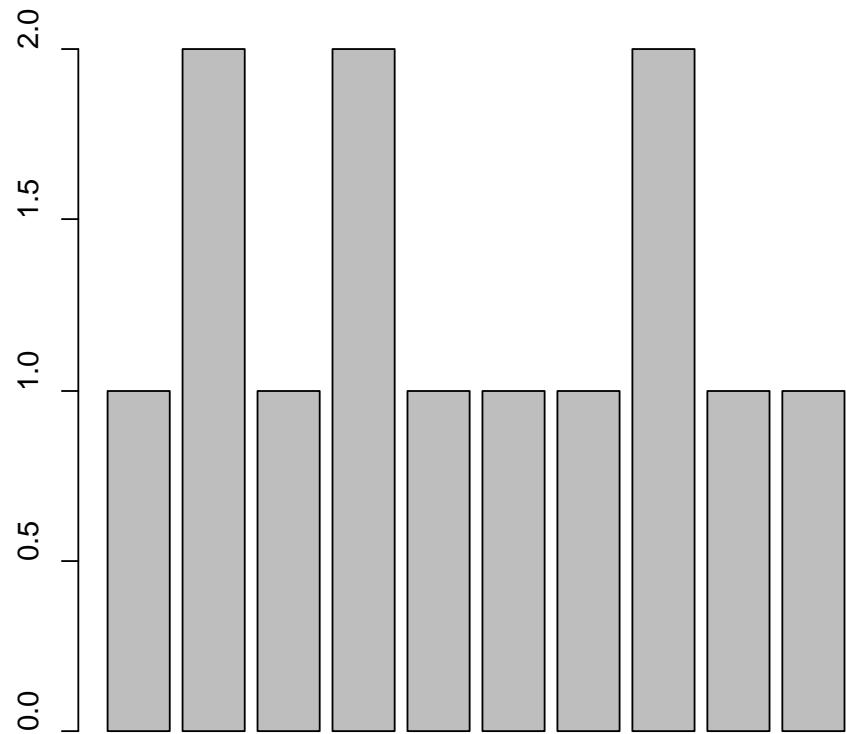
R Console

```
> gender <- c(1, 2, 1, 2, 1, 1, 1, 2, 1, 1)  
> gender  
[1] 1 2 1 2 1 1 1 2 1 1
```

# Bar plots: Example

> `barplot(gender)`

Do you want this?



## Bar plots: Example

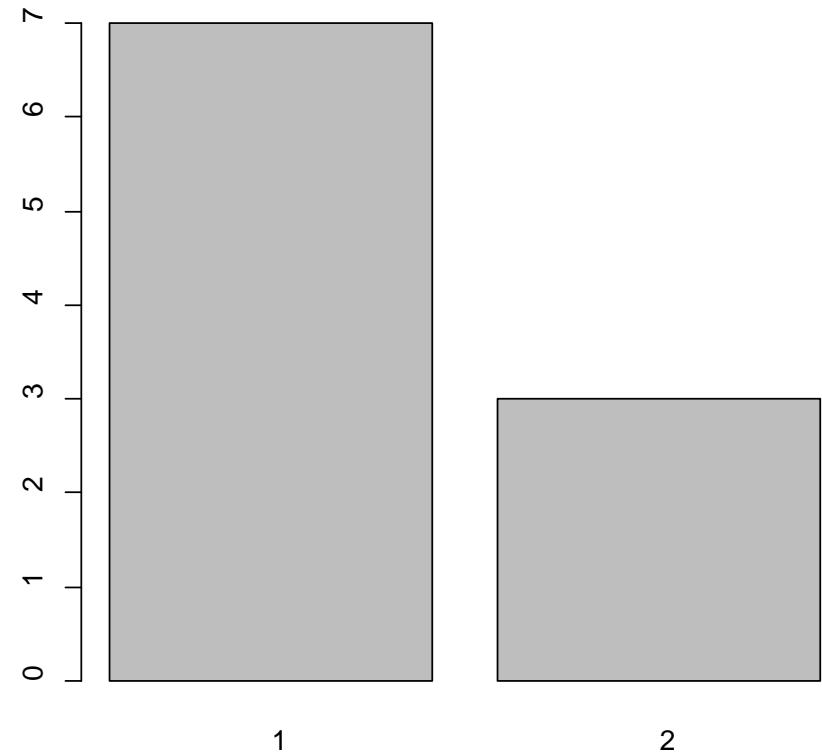
```
> table(gender)
```

```
gender
```

```
1 2
```

```
7 3
```

```
> barplot(table(gender))
```

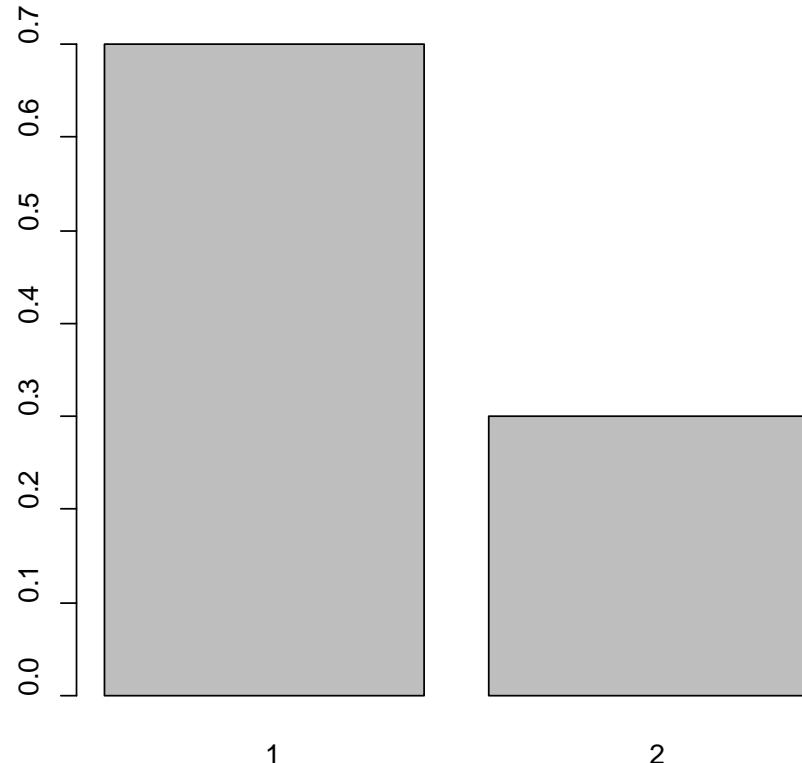


## Bar plots: Example

```
> table(gender)/length(gender)
```

gender

1	2
0.7	0.3



```
> barplot(table(gender)/length(gender))
```

## Example

'`pizza_delivery.csv`' contains the simulated data on pizza home delivery.

- There are three branches (East, West, Central) of the restaurant.
- The pizza delivery is centrally managed over phone and delivered by one of the five drivers.
- The data set captures the number of pizzas ordered and the final bill

```
> setwd("C:/Rcourse")  
> pizza <- read.csv('pizza_delivery.csv')
```

## Example

Consider data from Pizza. Take first 100 values from Direction and code Directions as

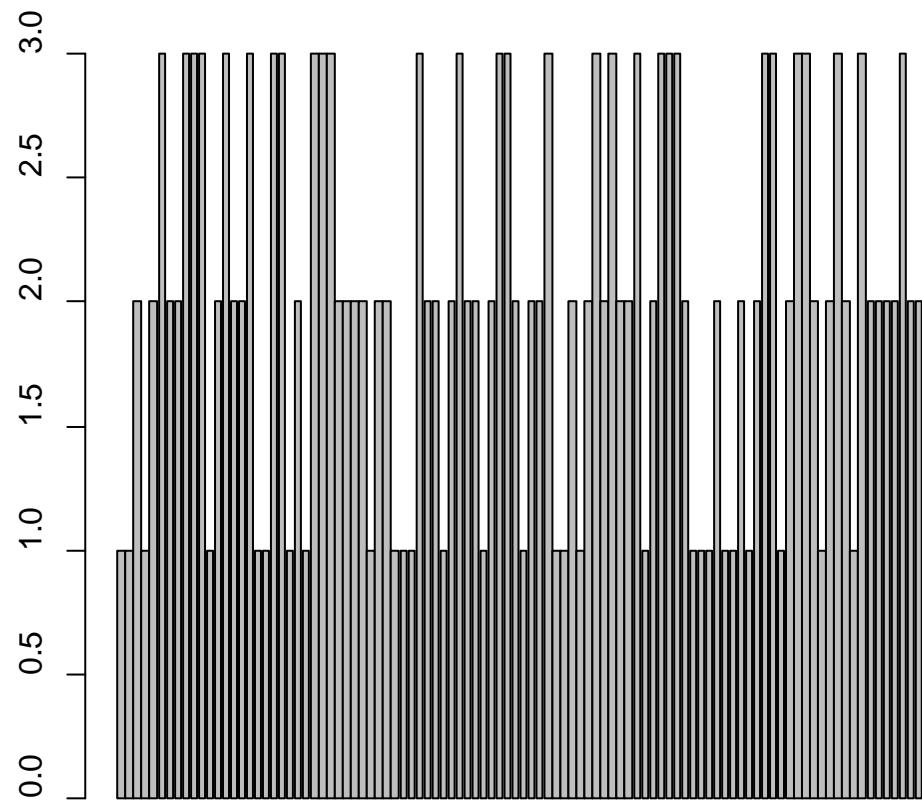
- ❖ East: 1
- ❖ West: 2
- ❖ Centre: 3

```
direction <-c(1,1,2,1,2,3,2,2,3,3,3,1,2,3,2,2,3,1,  
1,3,3,1,2,1,3,3,2,2,2,2,1,2,2,1,1,1,3,2,2,1,2,3,2  
,2,1,2,3,3,2,1,2,2,3,1,1,2,1,2,3,2,3,2,2,3,1,2,3,3,  
3,2,1,1,1,2,1,1,2,1,2,3,3,1,2,3,3,2,1,2,3,2,1,3,2,2  
,2,2,3,2,2)
```

# Bar plots: Example

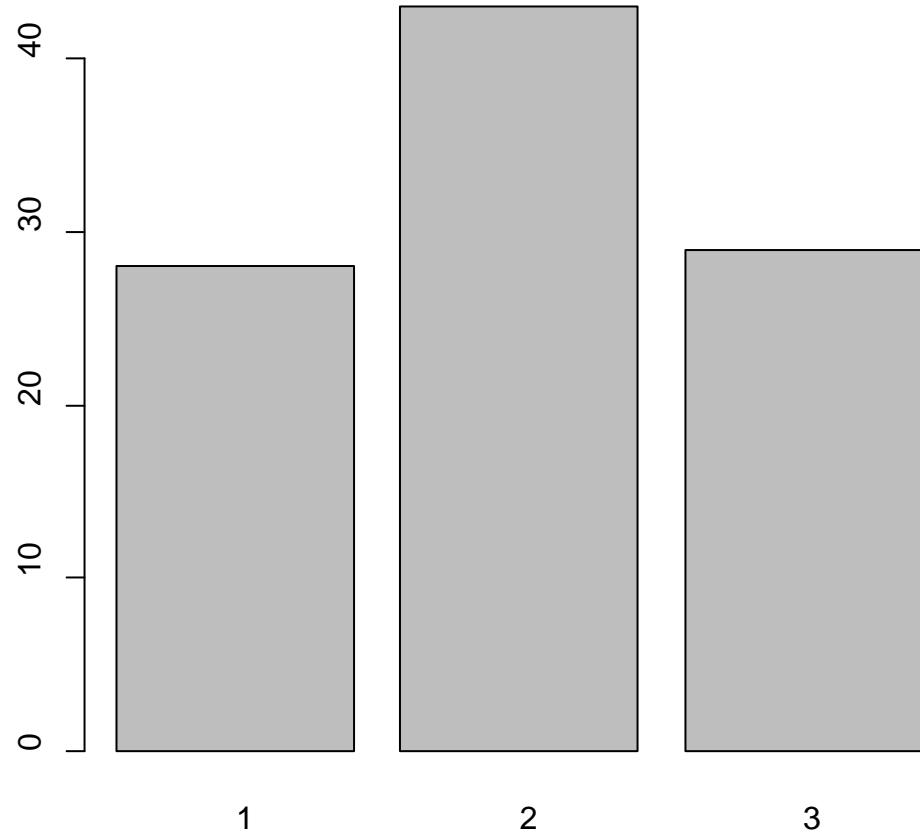
> `barplot(direction)`

Do you want this?



## Bar plots: Example

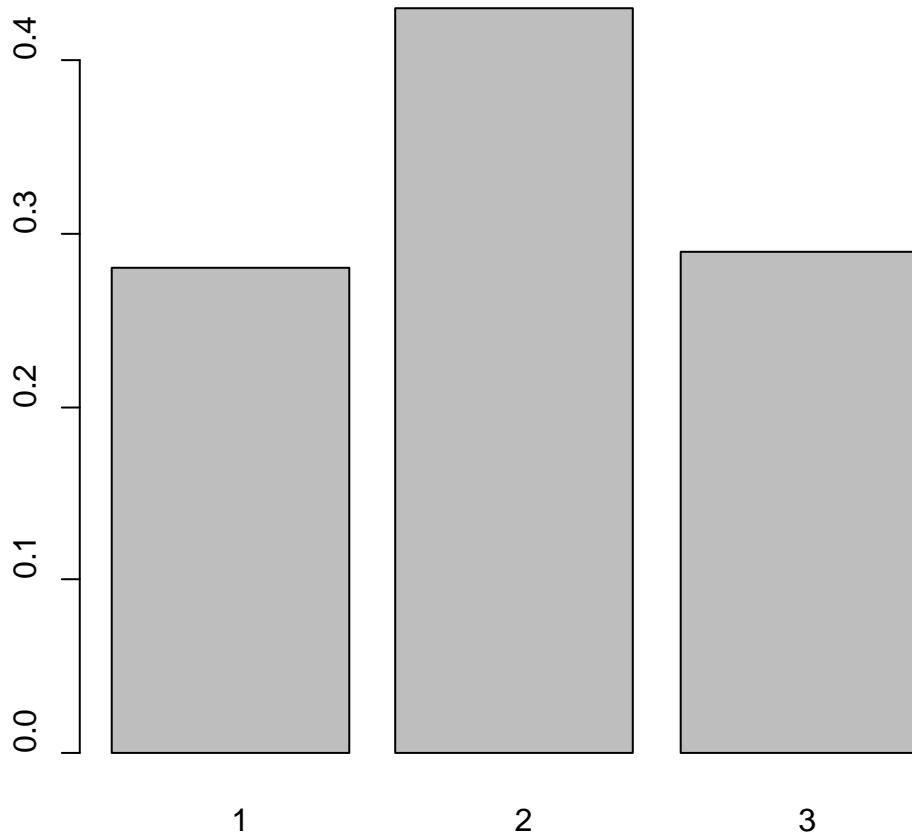
```
> barplot(table(direction))
```



## Bar plots:

### Example

```
> barplot(table(direction)/length(direction))
```



## Pie diagram:

Pie charts visualize the absolute and relative frequencies.

A pie chart is a circle partitioned into segments where each of the segments represents a category.

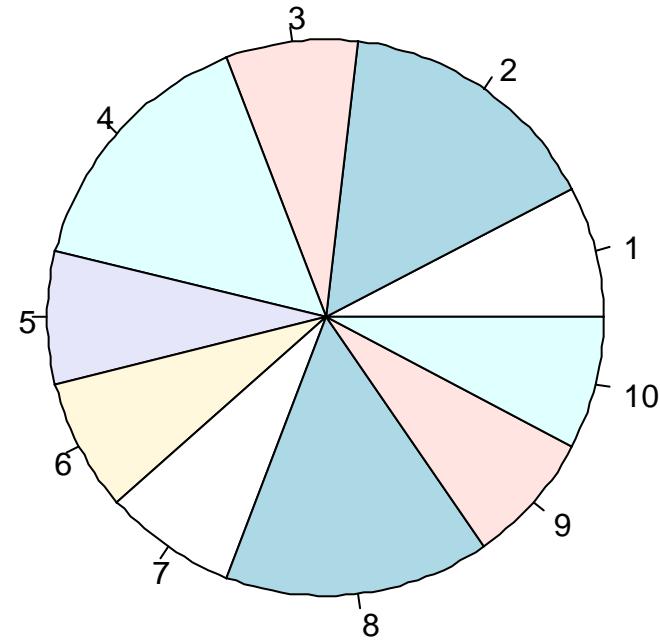
The size of each segment depends upon the relative frequency and is determined by the angle (frequency  $\times$   $360^\circ$ ).

```
pie(x, labels = names(x), ...)
```

# Pie diagram: Example

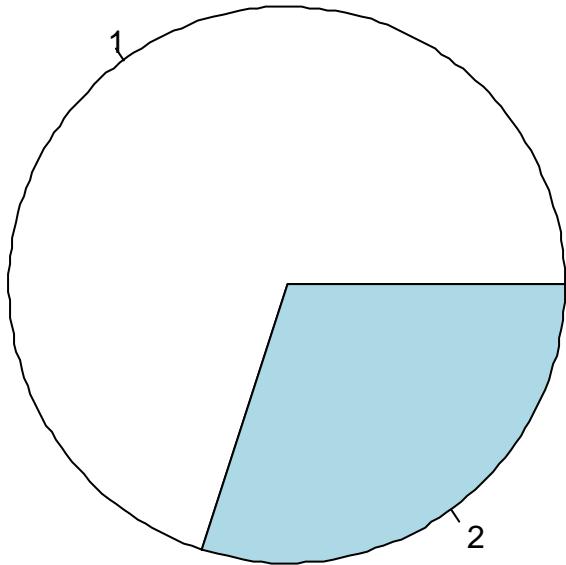
```
> pie(gender)
```

Do you want this?

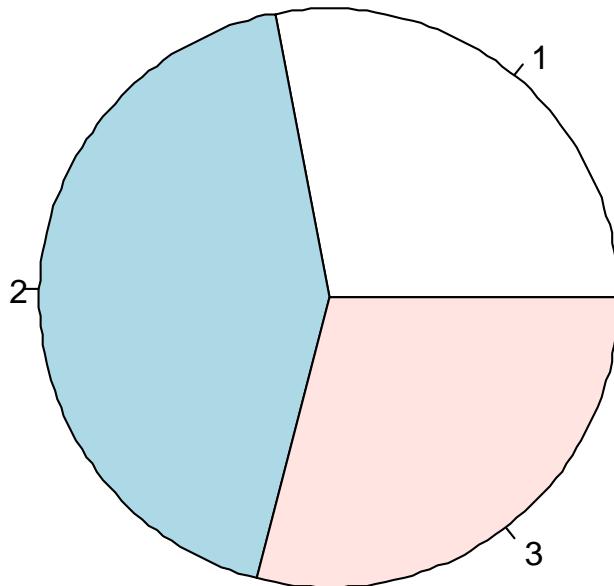


## Pie diagram: Example

```
> pie(table(gender))
```



```
> pie(table(direction))
```



## Histogram:

Histogram is based on the idea to categorize the data into different groups and plot the bars for each category with height.

The area of the bars (= height X width) is proportional to the relative frequency.

So the widths of the bars need not necessarily to be the same

```
hist(x) # show absolute frequencies
```

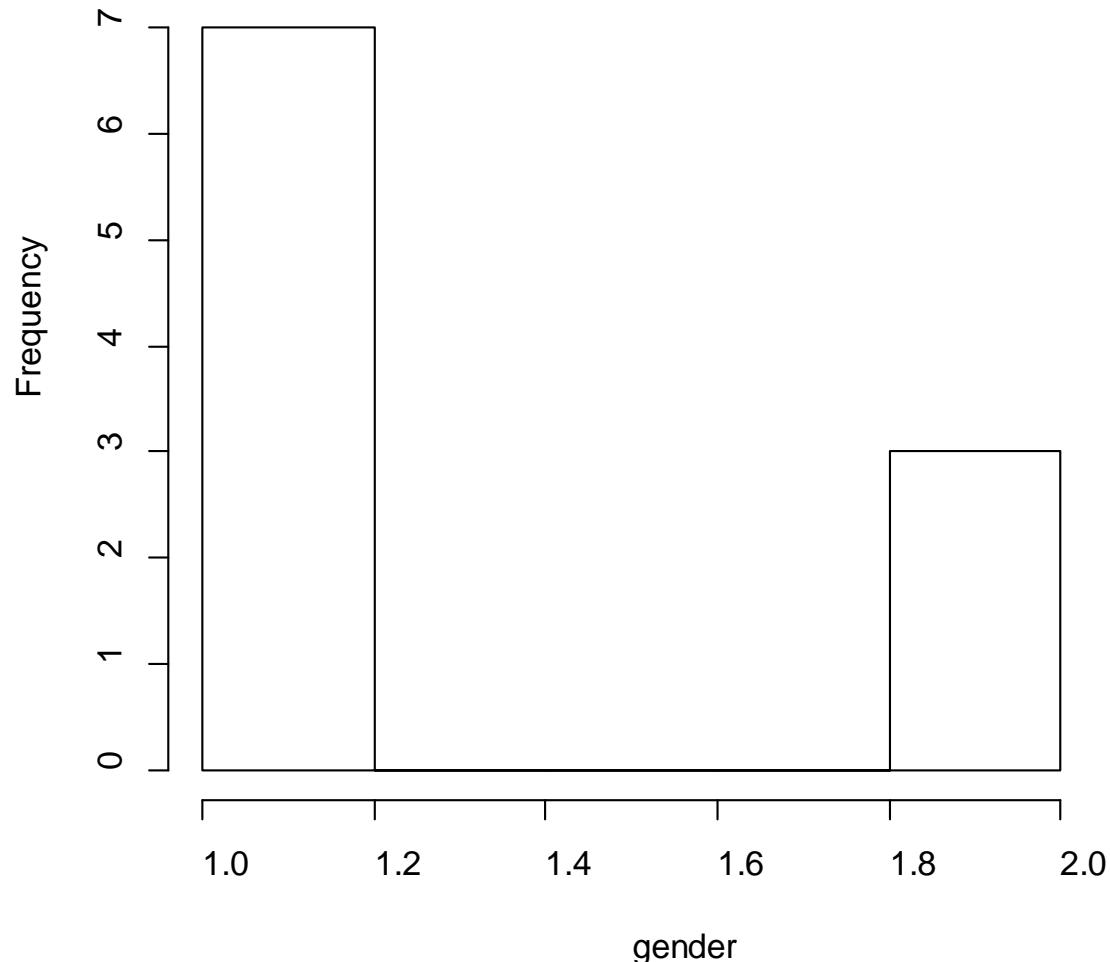
```
hist(x, freq=F) # show relative frequencies
```

See `help("hist")` for more details

# Histogram: Example

Histogram of gender

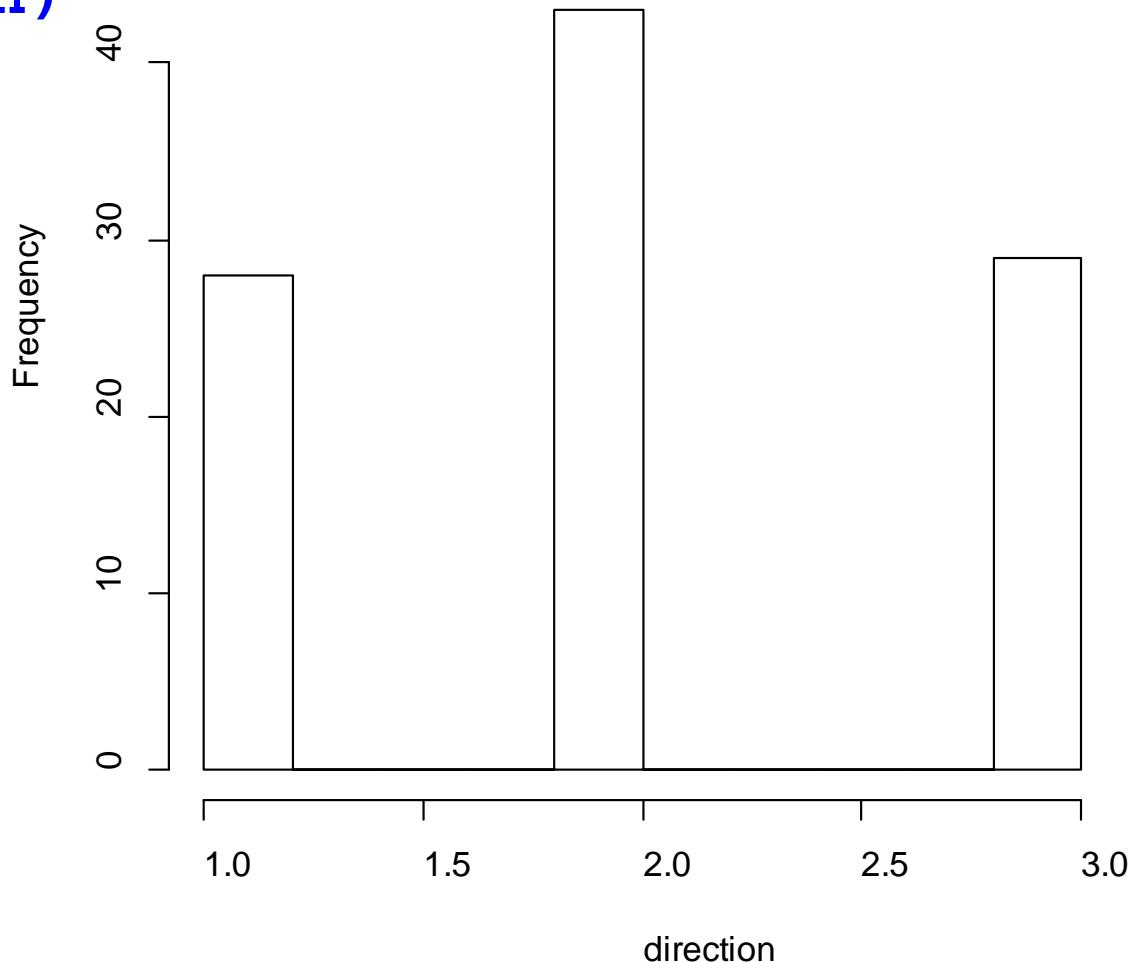
```
> hist(gender)
```



# Histogram: Example

```
> hist(direction)
```

Histogram of direction



# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

## **Central Tendency and Variation**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Descriptive statistics:**

**First hand tools which gives first hand information.**

- **Central tendency of data (Mean, median, mode, geometric mean, harmonic mean etc.)**
- **Variation in data (variance, standard deviation, standard error, mean deviation etc.)**

# Central tendency of the data

Gives an idea about the mean value of the data

The data is clustered around what value?

**Data:**  $x_1, x_2, \dots, x_n$

**x** : Data vector

**Arithmetic mean (mean)**

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

**mean (x)**

## Central tendency of the data

Geometric mean

$$\bar{x}_{GM} = \left( \prod_{i=1}^n x_i \right)^{\frac{1}{n}}$$

`prod(x)^(1/length(x))`

(`length(x)` is equal to the number of elements in x)

Harmonic mean

$$\bar{x}_{HM} = \frac{n}{\frac{1}{n} \sum_{i=1}^n \frac{1}{x_i}}$$

`1/mean(1/x)`

# **Central tendency of the data**

## **Median:**

Value such that the number of observation above it is equal to the number of observation below it.

**median(x)**

# Example

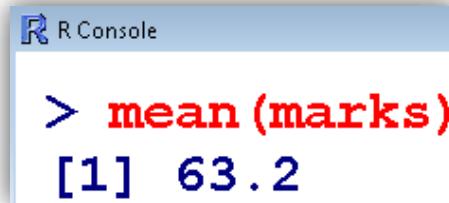
```
> marks<- c(68, 82, 63, 86, 34, 96, 41, 89,  
29, 51, 75, 77, 56, 59, 42)
```



A screenshot of an R console window titled "R Console". The window shows the command `> marks<- c(68, 82, 63, 86, 34, 96, 41, 89, 29, 51, 75, 77, 56, 59, 42)` in red, indicating it has been typed by the user.

## Arithmetic mean:

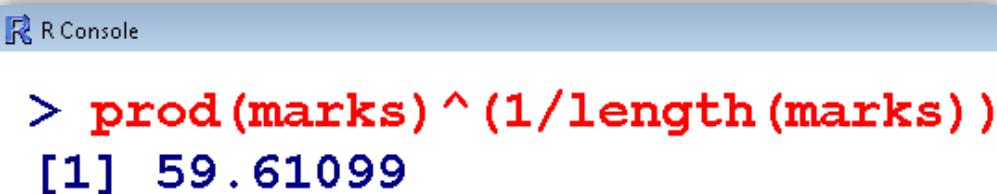
```
> mean(marks)  
[1] 63.2
```



A screenshot of an R console window titled "R Console". The window shows the command `> mean(marks)` in red and its output `[1] 63.2` in black, indicating the arithmetic mean has been calculated.

## Geometric mean:

```
> prod(marks)^(1/length(marks))  
[1] 59.61099
```

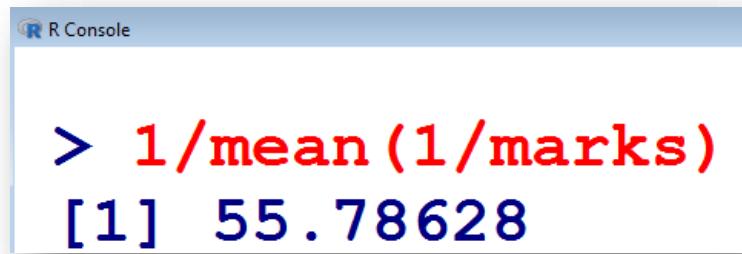


A screenshot of an R console window titled "R Console". The window shows the command `> prod(marks)^(1/length(marks))` in red and its output `[1] 59.61099` in black, indicating the geometric mean has been calculated.

## Example

Harmonic mean:

```
> 1/mean(1/marks)  
[1] 55.78628
```



A screenshot of an R console window titled "R Console". The window shows the command `> 1/mean(1/marks)` in red, indicating it is being typed or has just been entered. Below it, the output `[1] 55.78628` is displayed in blue, representing the calculated harmonic mean.

Median:

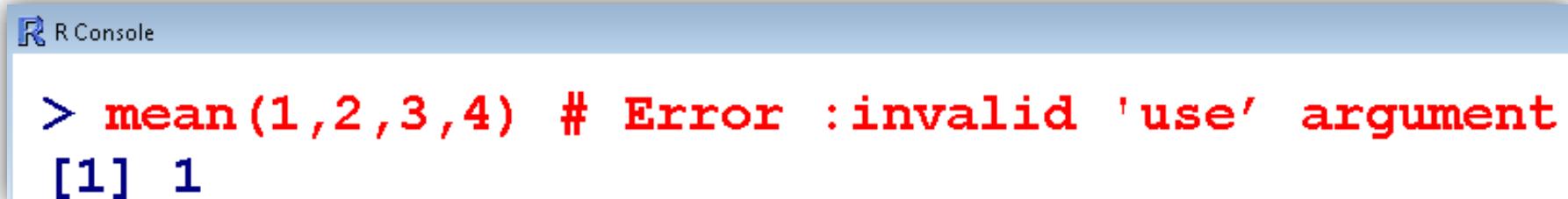
```
> median(marks)  
[1] 63
```



A screenshot of an R console window titled "R Console". The window shows the command `> median(marks)` in red. Below it, the output `[1] 63` is displayed in blue, representing the calculated median.

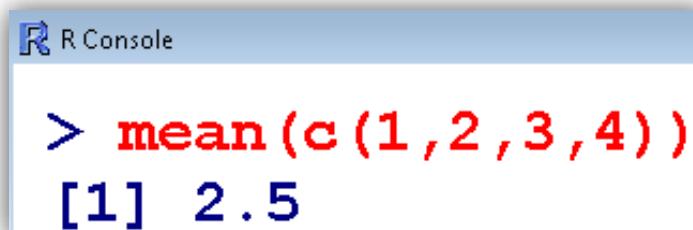
## *Doesn't do what you would expect:*

```
> mean(1,2,3,4) # Error :invalid 'use' argument  
[1] 1
```



A screenshot of an R console window titled "R Console". The window contains the following text:  
> mean(1,2,3,4) # Error :invalid 'use' argument  
[1] 1

```
> mean(c(1,2,3,4))  
[1] 2.5
```



A screenshot of an R console window titled "R Console". The window contains the following text:  
> mean(c(1,2,3,4))  
[1] 2.5



# Variability

Spread and scatterdness of data around any point, preferably the mean value.

Data set 1: 360, 370, 380

$$\text{mean} = (360 + 370 + 380)/3 = 370$$

Data set 2: 10, 100, 1000

$$\text{mean} = (10 + 100 + 1000)/3 = 370$$

How to differentiate between the two data sets?

# Variability

**Variance**       $\text{var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$

**x:** data vector

**var(x)**

**Positive square root of variance : standard deviation**

**sqrt(var(x))**

# Variability

## Variance

Another variant,

$$\text{var}(x) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

If we want divisor to be n, then use

`((n - 1)/n)*var(x)`

where `n = length(x)`

# Variability

Range:

$$\text{maximum}(x_1, x_2, \dots, x_n) - \text{minimum}(x_1, x_2, \dots, x_n)$$

$$\max(x) - \min(x)$$

Interquartile range:

$$\text{Third quartile}(x_1, x_2, \dots, x_n) - \text{First quartile}(x_1, x_2, \dots, x_n)$$

$$\text{IQR}(x)$$

# Variability

**Quartile deviation:**

$$\begin{aligned} & [\text{Third quartile } (x_1, x_2, \dots, x_n) - \text{First quartile } (x_1, x_2, \dots, x_n)]/2 \\ & = \text{Interquartile range}/2 \end{aligned}$$

**IQR(x)/2**

**Mean deviation:**

$$MD(x) = \frac{1}{n} \sum_{i=1}^n |x_i - \bar{x}|$$

**sum(abs(x-mean(x)))/length(x)**

## Example

x: data vector

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
29, 51, 75, 77, 56, 59, 42)
```

Variance:

```
> var(marks)  
[1] 439.3143
```

R R Console  
> var(marks)  
[1] 439.3143

Standard deviation:

```
> sqrt(var(marks))  
[1] 20.95983
```

R R Console  
> sqrt(var(marks))  
[1] 20.95983

## Example

Interquartile Range:

```
> IQR(marks)  
[1] 33
```

```
R R Console  
> IQR(marks)  
[1] 33
```

Quartile deviation :

```
> IQR(marks)/2  
[1] 16.5
```

```
R R Console  
> IQR(marks)/2  
[1] 16.5
```

Mean deviation:

```
> sum(abs(marks-mean(marks)))/length(marks)  
[1] 17.41333
```

```
R R Console  
> sum(abs(marks-mean(marks)))/length(marks)  
[1] 17.41333
```

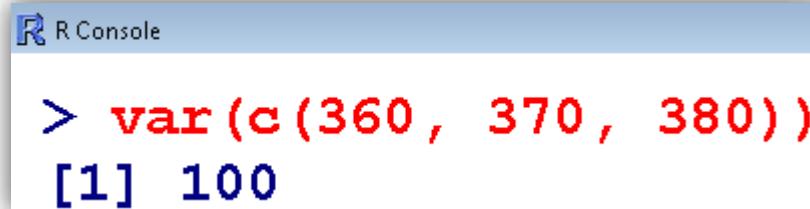
## Example

Data set 1: 360, 370, 380

$$\text{mean} = (360 + 370 + 380)/3 = 370$$

```
> var(c(360, 370, 380))
```

```
[1] 100
```



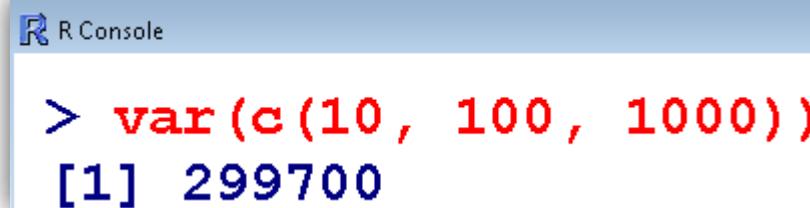
```
R Console
> var(c(360, 370, 380))
[1] 100
```

Data set 2: 10, 100, 1000

$$\text{mean} = (10 + 100 + 1000)/3 = 370 \text{ Same as of Data set 1}$$

```
> var(c(10, 100, 1000))
```

```
[1] 299700
```



```
R Console
> var(c(10, 100, 1000))
[1] 299700
```

## *Doesn't do what we would expect:*

```
> var(1,2,3,4)  
Error in var(1, 2, 3, 4) : invalid 'use' argument
```

R R Console

```
> var(1,2,3,4)  
Error in var(1, 2, 3, 4) : invalid 'use' argument
```

```
> var( c(1,2,3,4) )  
[1] 1.666667
```

R R Console

```
> var( c(1,2,3,4) )  
[1] 1.666667
```

# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

## **Boxplots, Skewness and Kurtosis**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Summary of observations**

In R, quartiles, minimum and maximum values can be easily obtained by the **summary** command

**summary(x)    x: data vector**

It gives information on

- ❖ minimum,
- ❖ maximum
- ❖ first quartile
- ❖ second quartile (median) and
- ❖ third quartile.

# Summary of observations

## Example:

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
 29, 51, 75, 77, 56, 59, 42)  
  
> summary(marks)  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
29.0 46.5 63.0 63.2 79.5 96.0
```

R R Console

```
> summary(marks)  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
29.0 46.5 63.0 63.2 79.5 96.0
```

>

# Summary of observations

## Example:

```
> marks1 <- c(628, 812, 613, 186, 34, 986, 41,  
 89, 29, 51, 795, 77, 56, 509, 420)  
  
> summary(marks1)  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
29.0 53.5 186.0 355.1 620.5 986.0
```

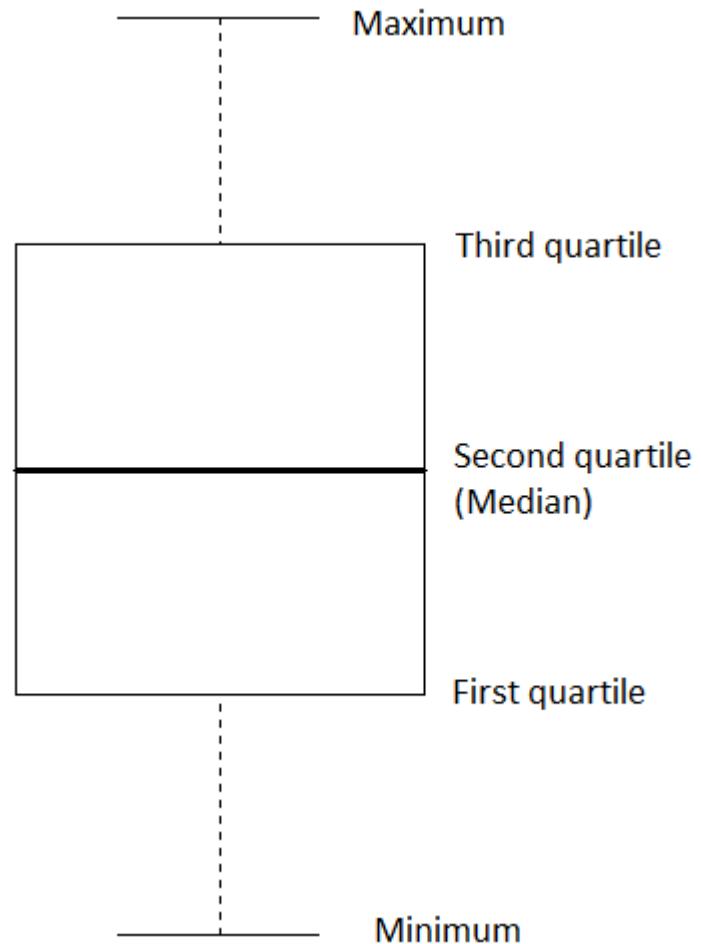
Earlier, we had

```
> summary(marks)  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
29.0 46.5 63.0 63.2 79.5 96.0
```

# Boxplot

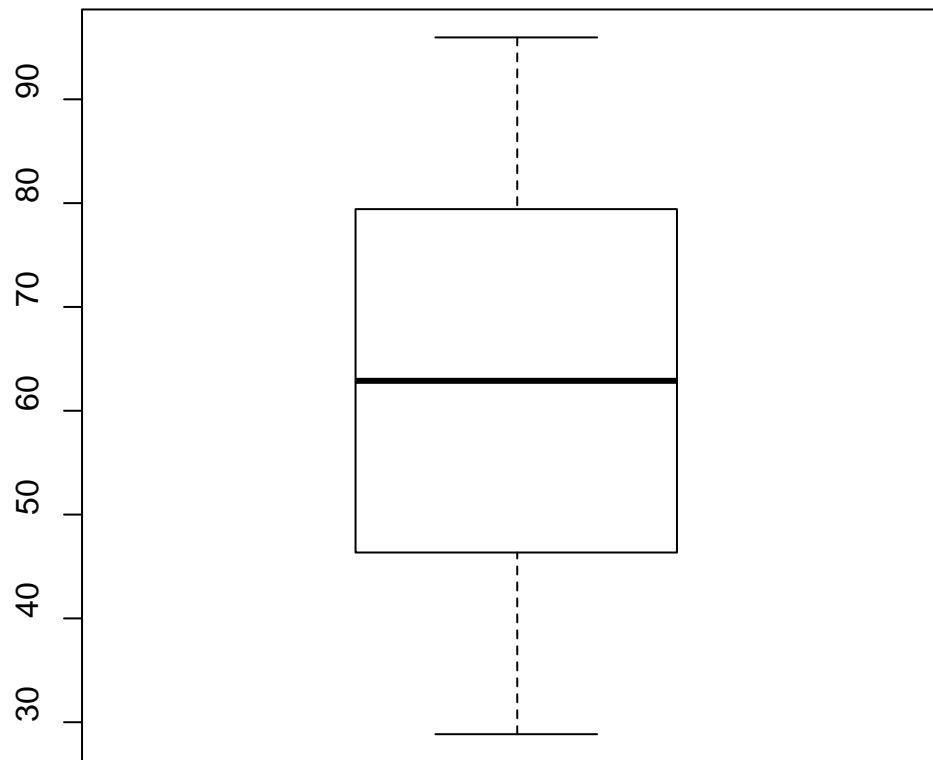
Box plot is a graph which summarizes the distribution of a variable by using its median, quartiles, minimum and maximum values.

`boxplot( )` draws a box plot.



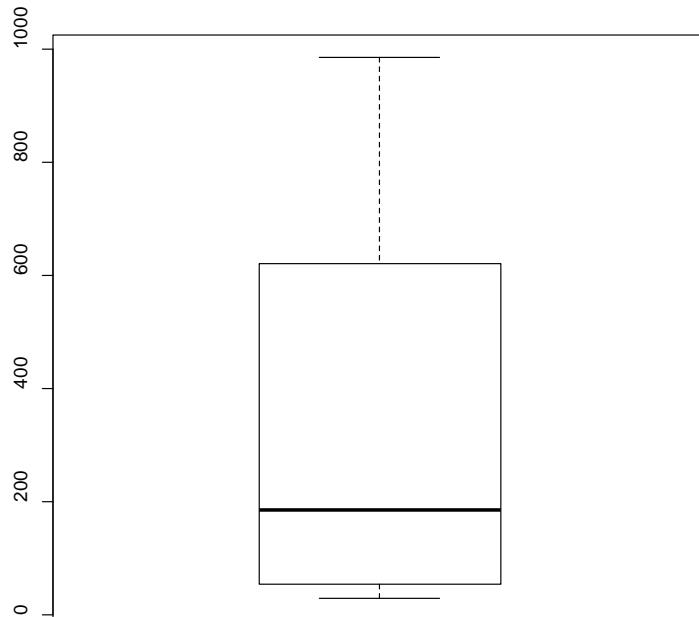
## Example:

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
 29, 51, 75, 77, 56, 59, 42)  
  
> boxplot(marks)
```

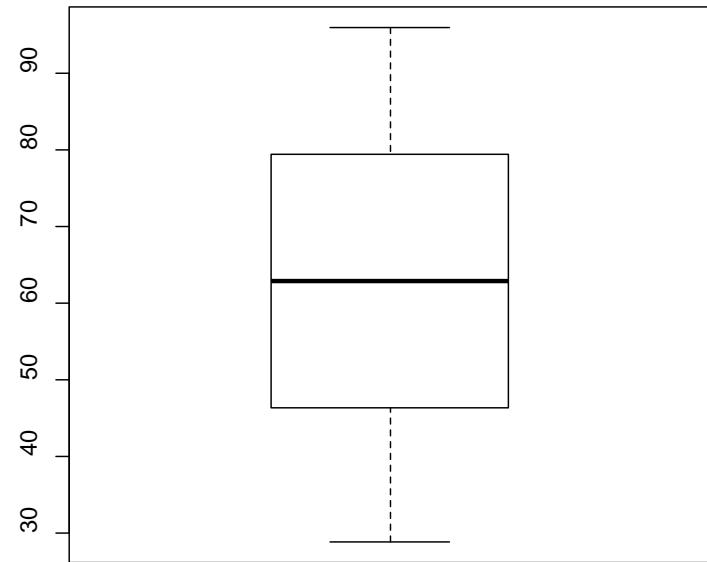


## Example:

```
> marks1 <- c(628, 812, 613, 186, 34, 986, 41,  
 89, 29, 51, 795, 77, 56, 509, 420)  
  
> boxplot(marks1)
```



Boxplot(marks1)



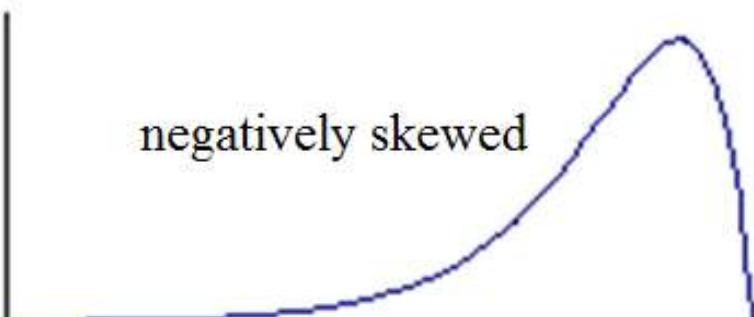
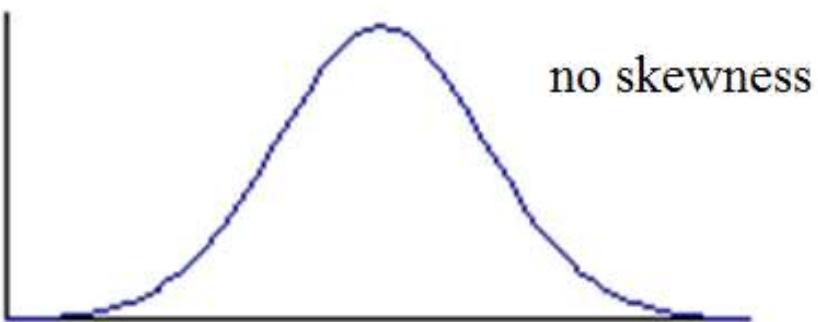
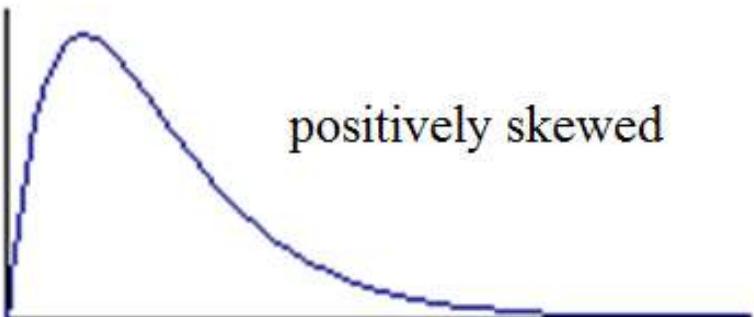
Boxplot(marks)

## **Descriptive statistics:**

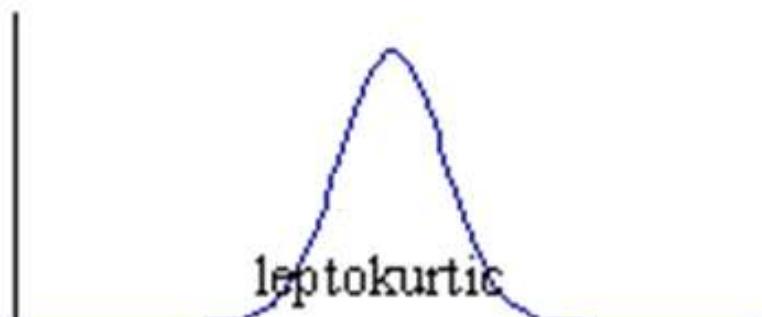
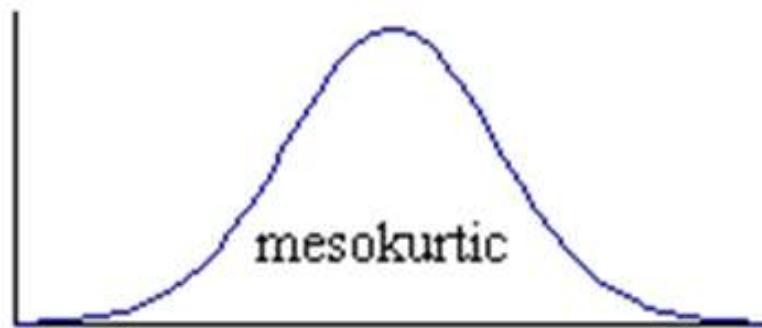
**First hand tools which gives first hand information.**

- **Structure and shape of data tendency (symmetry, skewness, kurtosis etc.)**
- **Relationship study (correlation coefficient, rank correlation, correlation ratio, regression etc.)**

## Skewness



## Kurtosis



## Skewness

Measures the shift of the hump of frequency curve .

Coefficient of skewness based on values  $x_1, x_2, \dots, x_n$ .

$$\gamma_1 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^3}{\left( \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{3/2}}$$

Mean :

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

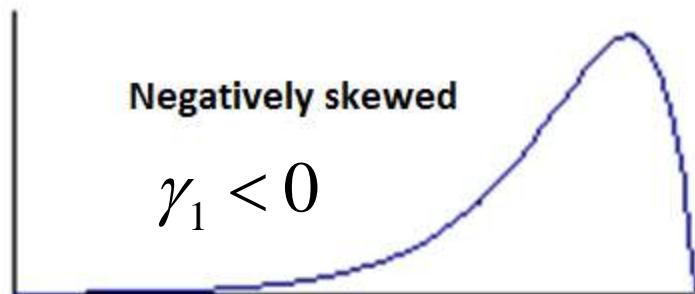
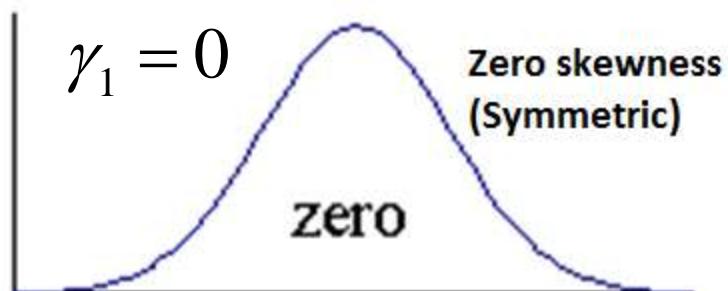
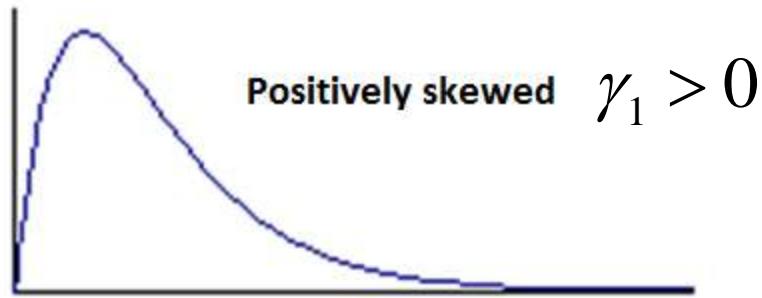
## Kurtosis

Measures the peakedness of the frequency curve.

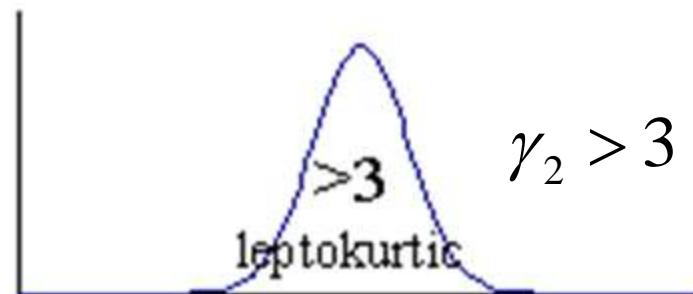
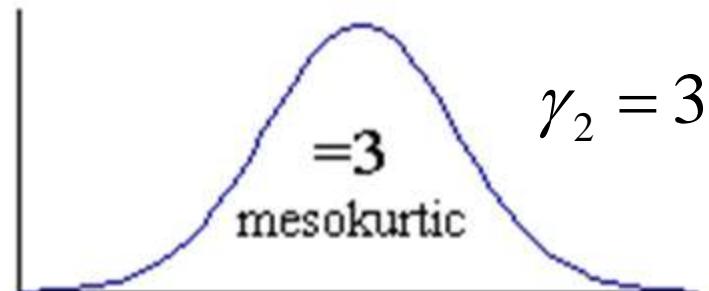
Coefficient of kurtosis based on values  $x_1, x_2, \dots, x_n$ .

$$\gamma_2 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^4}{\left( \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2}, \quad -3 < \gamma_2 < 3$$

## Skewness



## Kurtosis



## Skewness and kurtosis

First we need to install a package 'moments'

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

```
> library(moments)
```

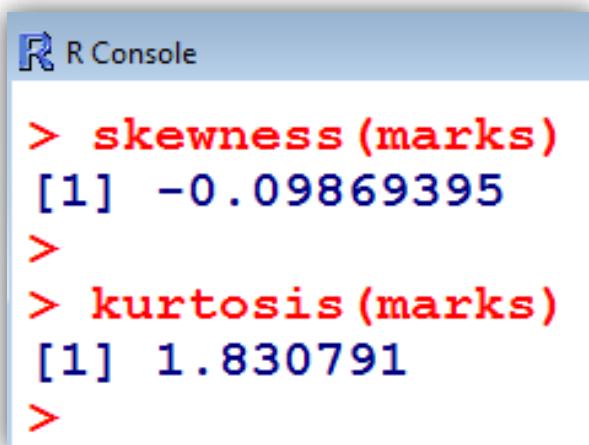
**skewness () : computes coefficient of skewness**

**kurtosis () : computes coefficient of kurtosis**

# Skewness and kurtosis

## Example

```
> marks <- c(68, 82, 63, 86, 34, 96, 41, 89,  
29, 51, 75, 77, 56, 59, 42)  
  
> skewness(marks)  
[1] -0.09869395  
  
> kurtosis(marks)  
[1] 1.830791
```



The screenshot shows the R console interface. The title bar says "R Console". The main area contains the following R code and its output:

```
> skewness(marks)
[1] -0.09869395
>
> kurtosis(marks)
[1] 1.830791
>
```

# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

## **Bivariate and Three Dimensional Plots**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Bivariate plots:**

**Provide first hand visual information about the nature and degree of relationship between two variables.**

**Relationship can be linear or nonlinear.**

**We discuss several types of plots through examples.**

# Scatter plot

Plot command:

**x, y:** Two data vectors

**plot(x, y)**

**plot(x, y, type)**

type	
"p" for points	"l" for lines
"b" for both	"c" for the lines part alone of "b"
"o" for both 'overplotted'	"s" for stair steps.
<b>"h"</b> for 'histogram' like (or 'high-density') vertical lines	

# Scatter plot

Plot command:

**x, y:** Two data vectors

**plot(x, y)**

**plot(x, y, type)**

Get more details from help: **help("type")**

Other options:

**main** an overall title for the plot.

**suba** sub title for the plot.

**xlab** title for the x axis.

**ylab** title for the y axis.

**asp** the y/x aspect ratio.

## Example:

Daily water demand in a city depends upon weather temperature.

We know from experience that water consumption increases as weather temperature increases.

Data on 27 days is collected as follows:

Daily water demand (in million litres)

```
water <- c(33710,31666,33495,32758,34067,36069,  
37497,33044,35216, 35383,37066,38037,38495,  
39895,41311,42849,43038,43873,43923, 45078,  
46935,47951,46085,48003,45050,42924,46061)
```

Temperature (in centigrade)

```
temp <- c(23,25,25,26,27,28,30,26,29,32,33,34,  
35,38,39,42,43,44, 45,45.5,45,46,44,44,41,37,40)
```

# Scatter plot

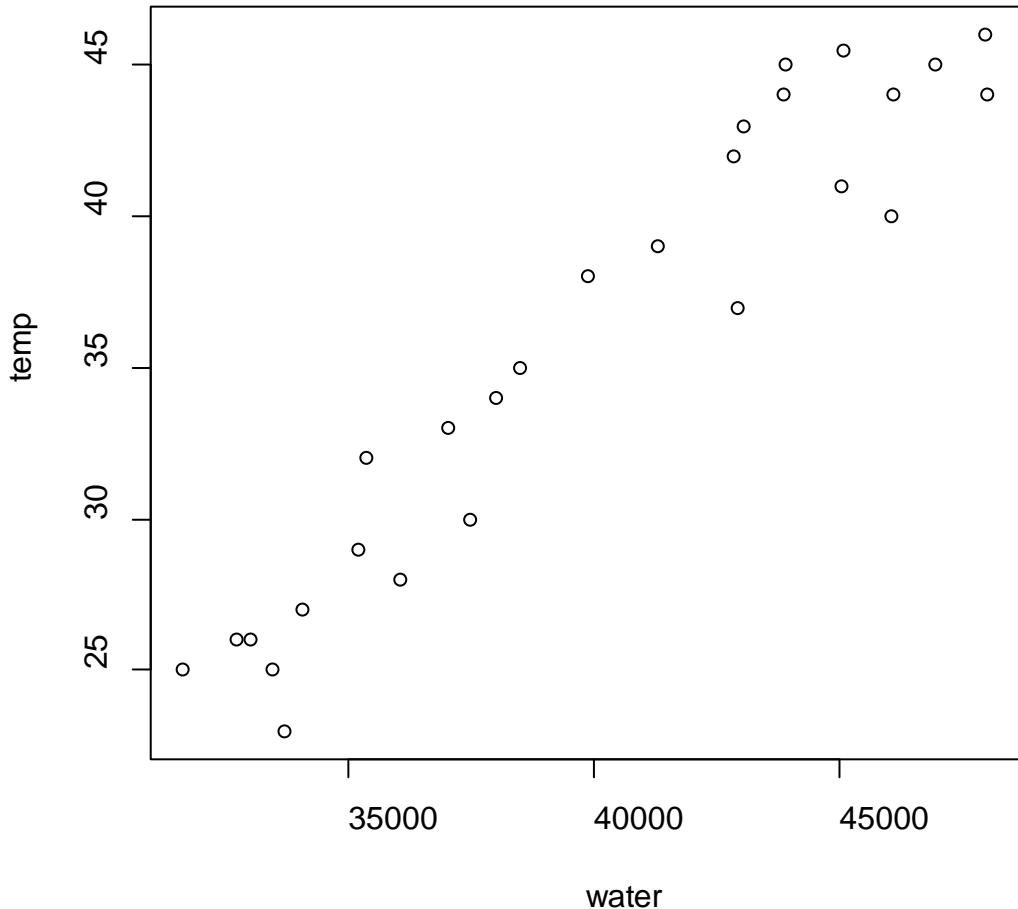
Plot command:

**x, y:** Two data vectors

Various type of plots are possible to draw.

`plot(x, y)`

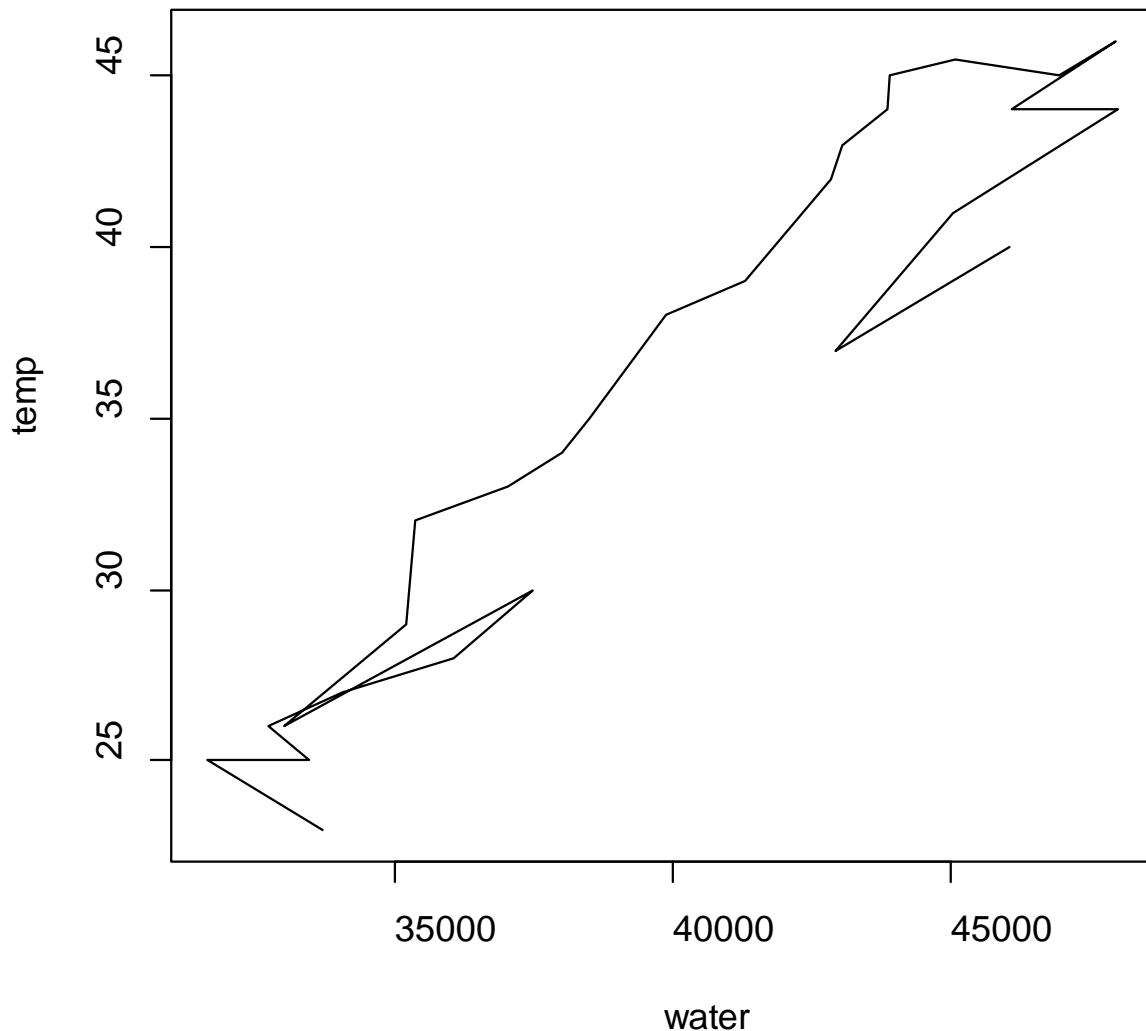
`plot(water, temp)`



## Scatter plot

```
plot(water, temp, "l")
```

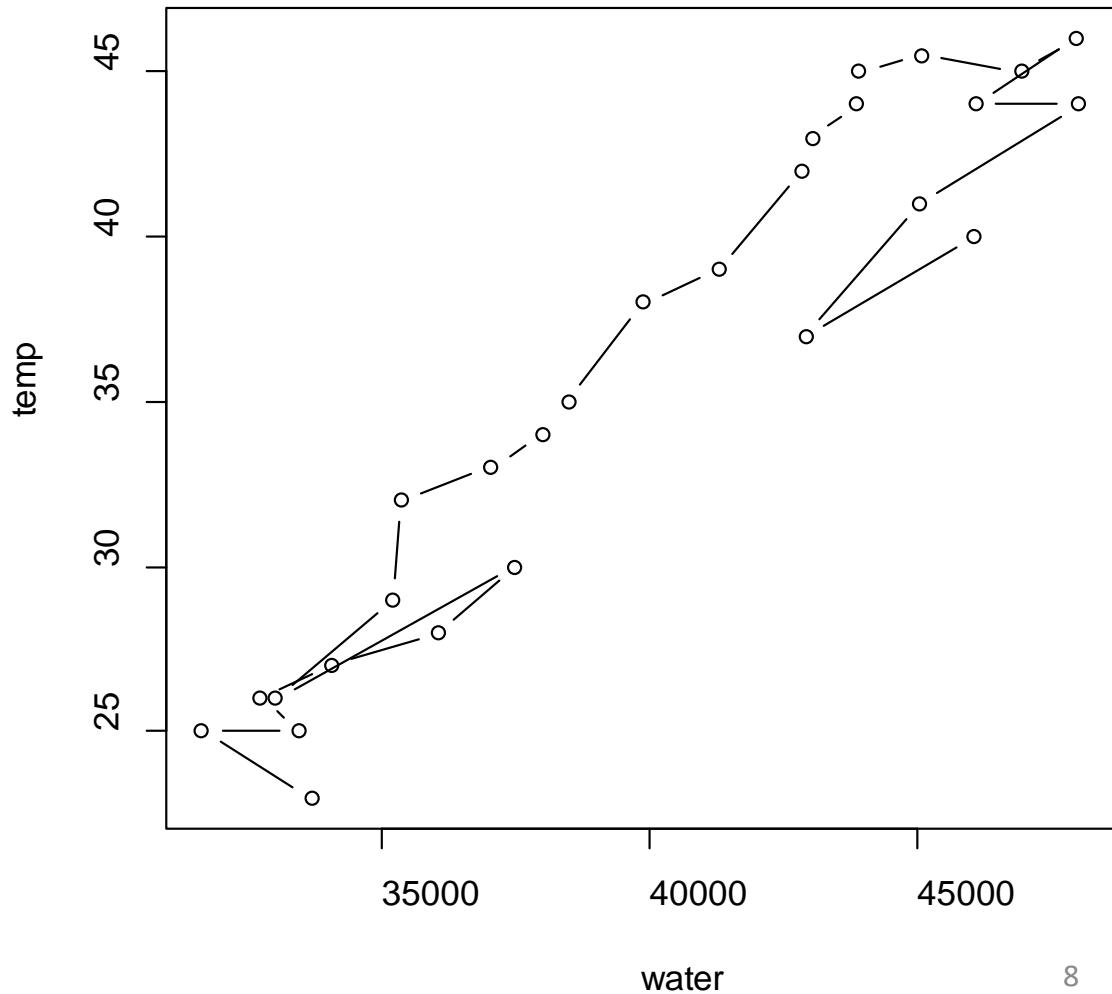
"l" for lines,



# Scatter plot

```
plot(water, temp, "b")
```

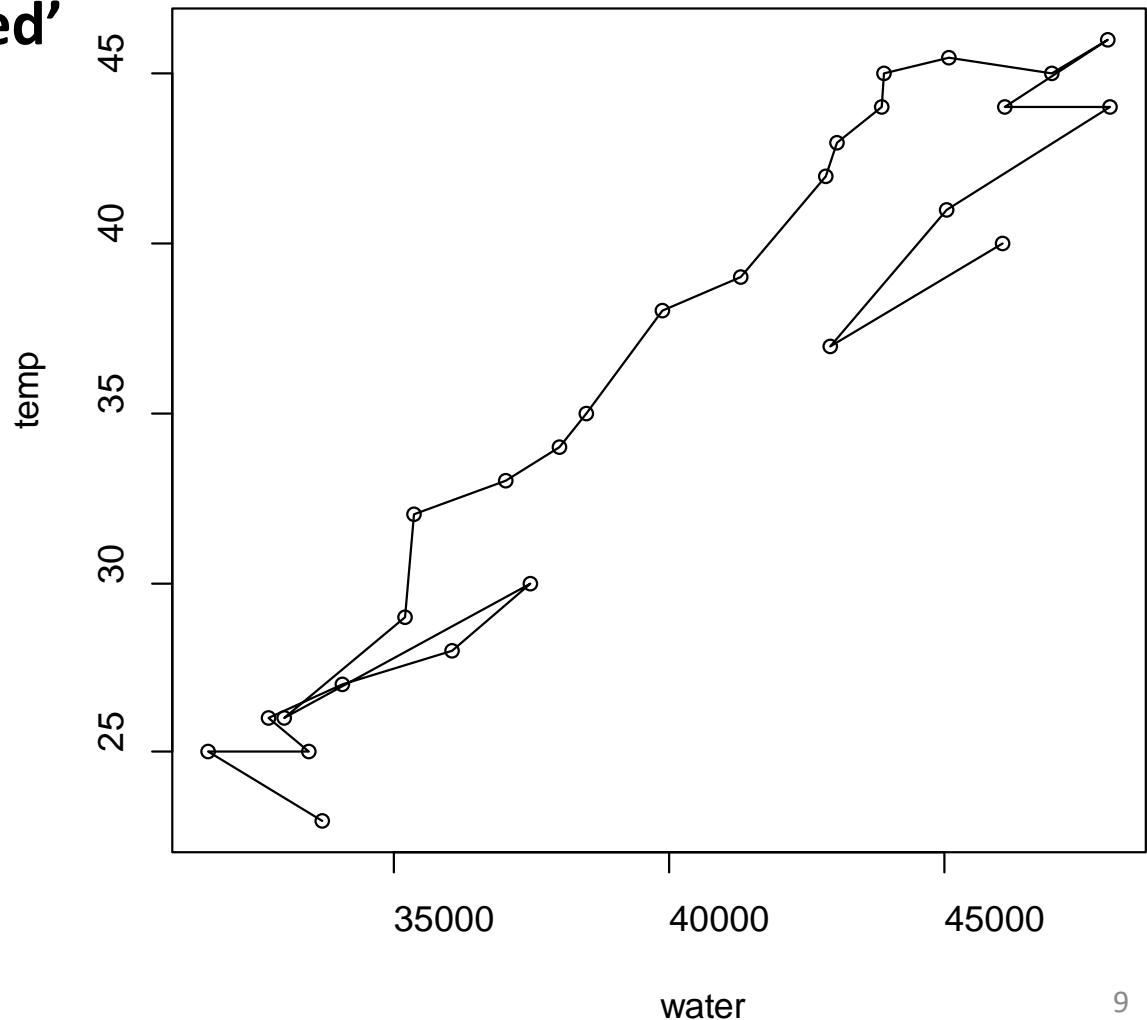
**"b"** for both – line and point



# Scatter plot

```
plot(water, temp, "o")
```

"o" for both 'overplotted'



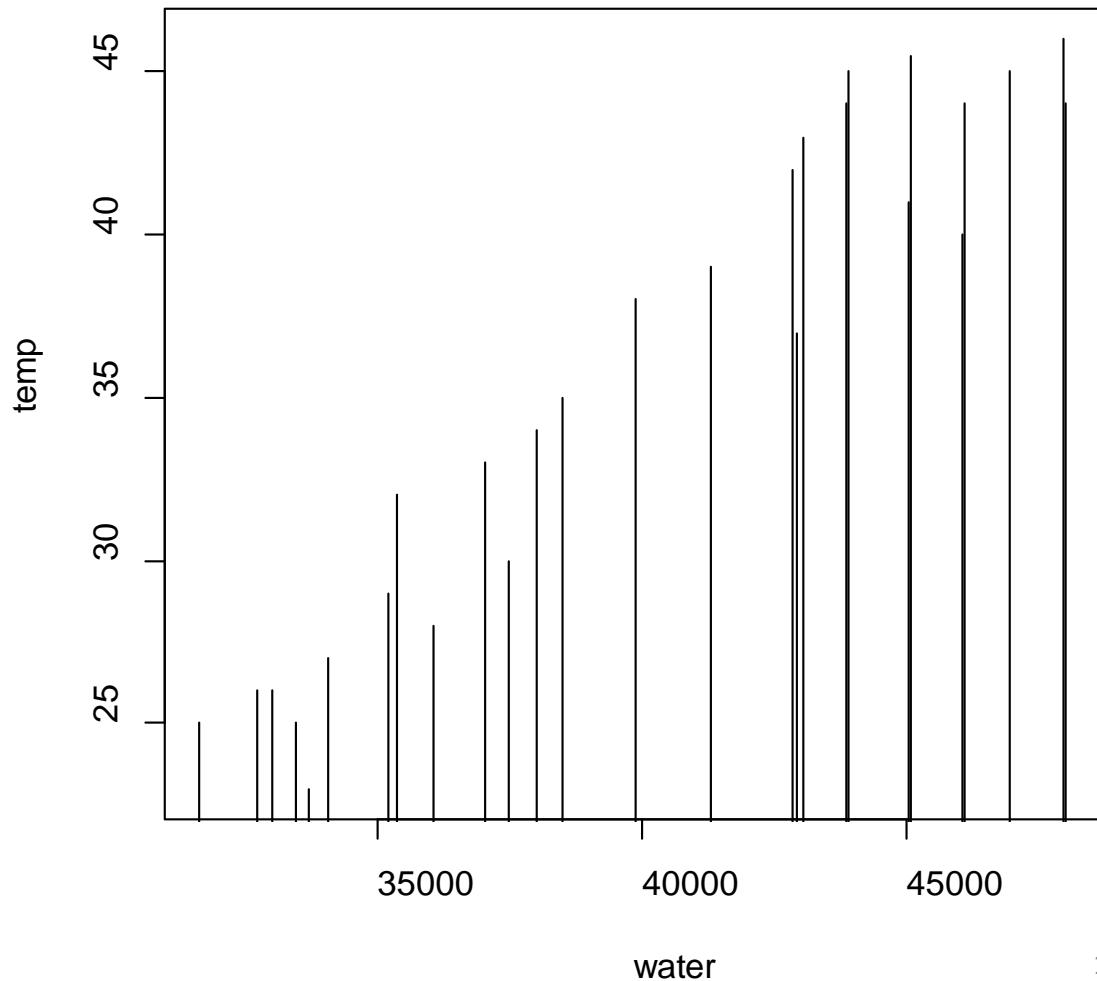
# Scatter plot

```
plot(water, temp, "h")
```

“h” for ‘histogram’

like (or ‘high-density’)

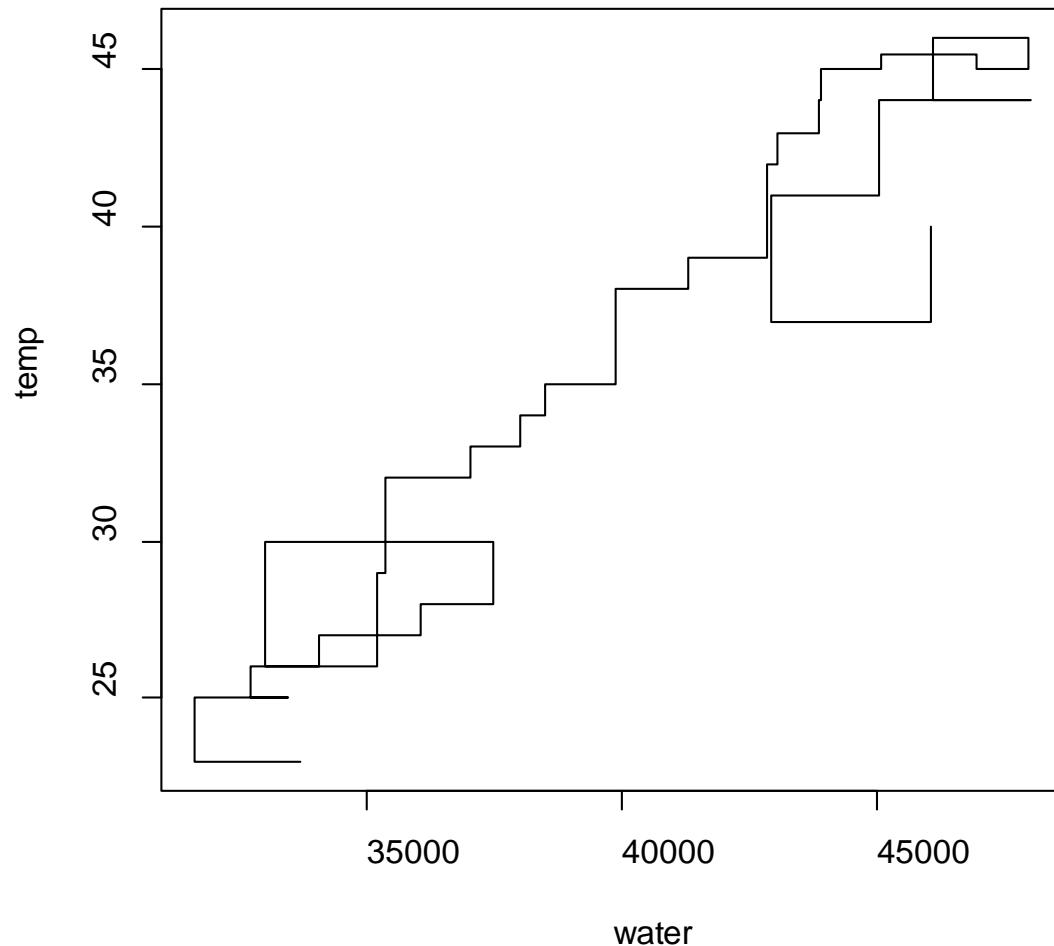
vertical lines



## Scatter plot

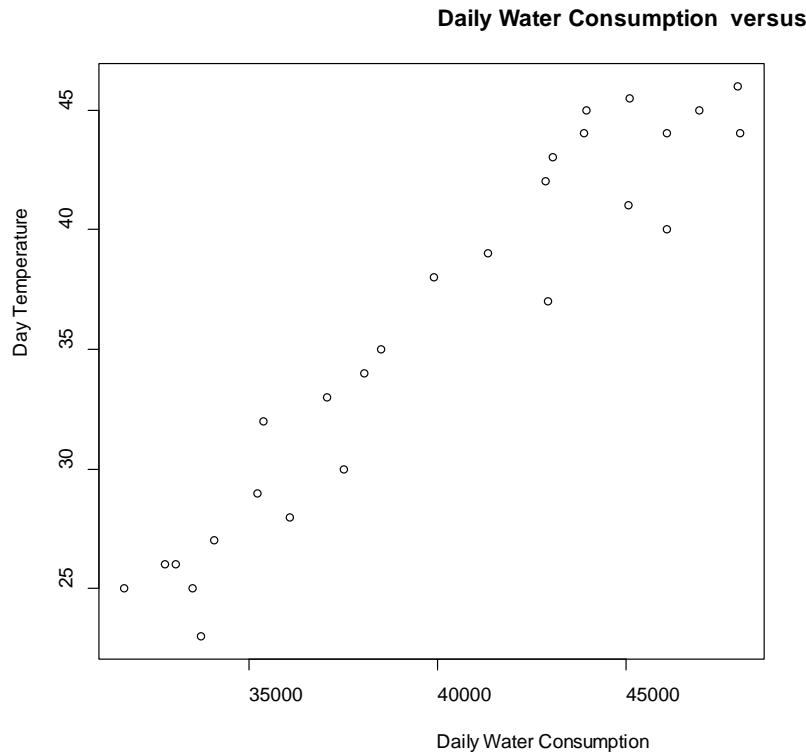
```
plot(water, temp, "s")
```

“s” for stair steps.



## Scatter plot

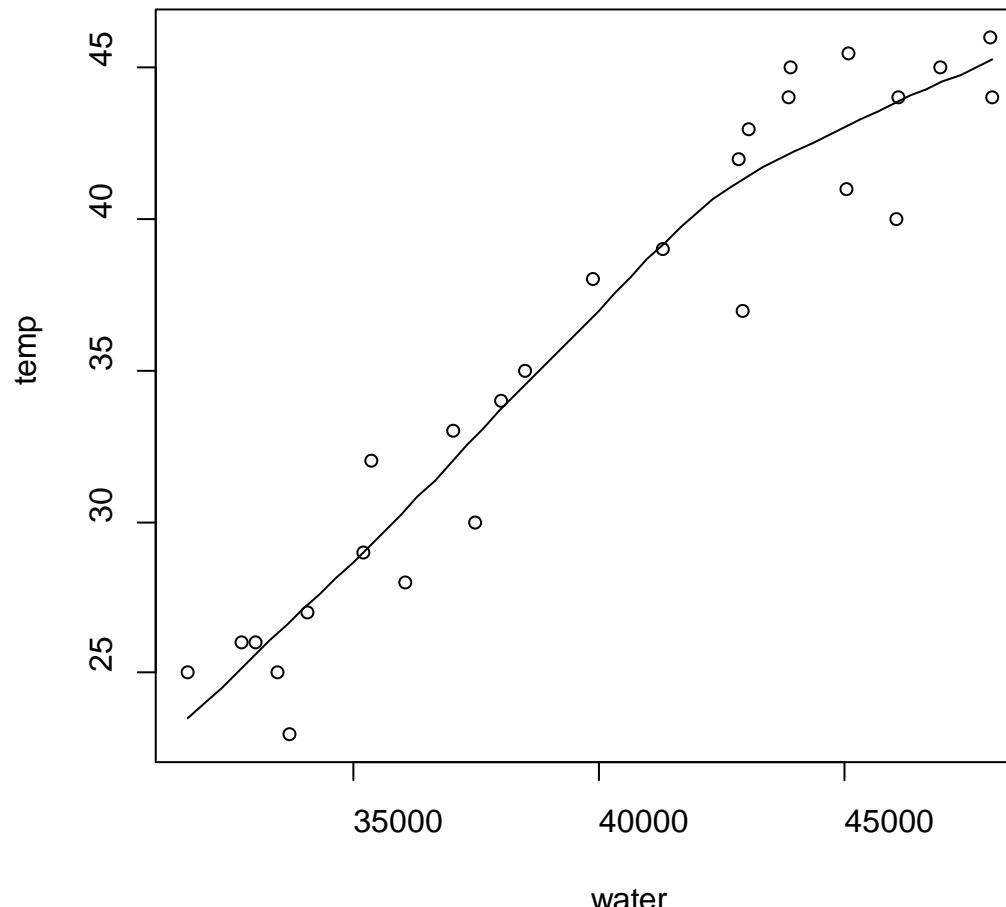
```
> plot(water, temp, xlab=" Daily Water Consumption ", ylab=" Day Temperature ", main=" Daily Water Consumption versus Day Temperature ")
```



# Smooth Scatter plot

`scatter.smooth(x,y)` provides scatter plot with smooth curve

Example: `scatter.smooth(water,temp)`



## Smooth Scatter plot

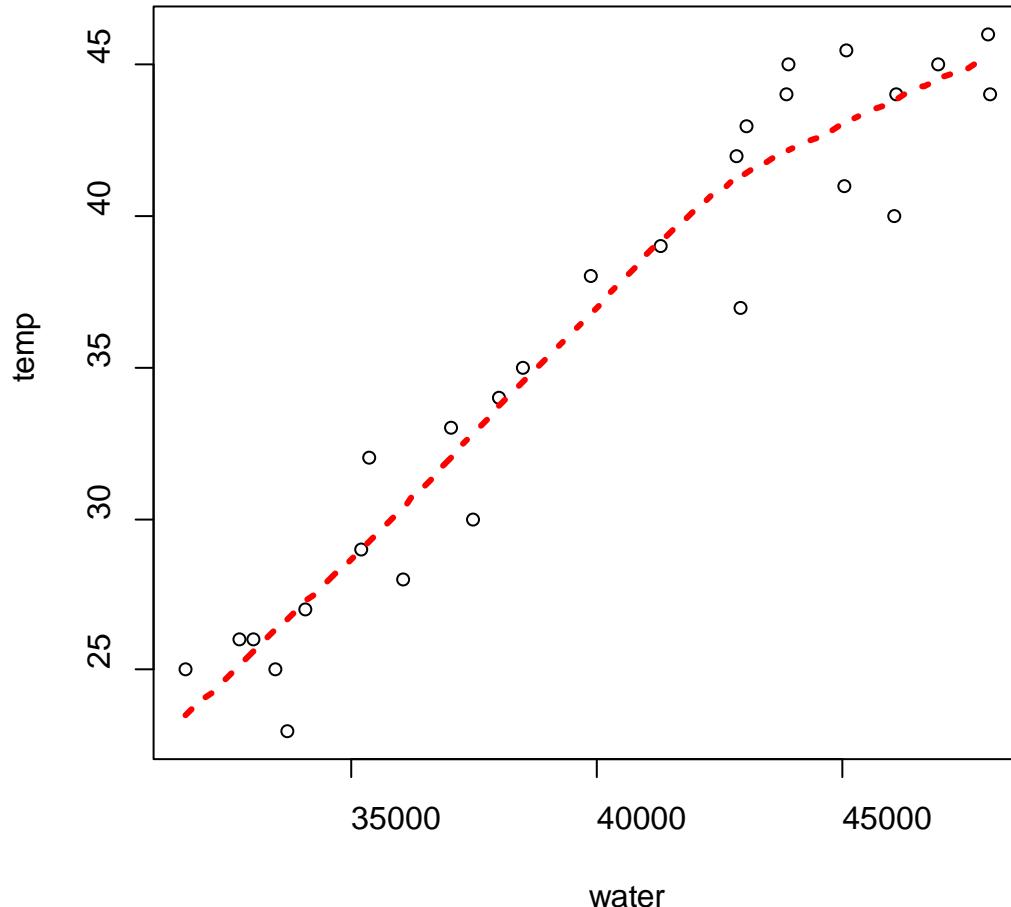
Other options are available.

```
scatter.smooth(x, y = NULL, span = 2/3, degree =  
1, family = c("symmetric", "gaussian"), xlab =  
NULL, ylab = NULL, ylim = range(y, pred$y, na.rm  
= TRUE), evaluation = 50, ..., lpars = list())
```

# Smooth Scatter plot

Example:

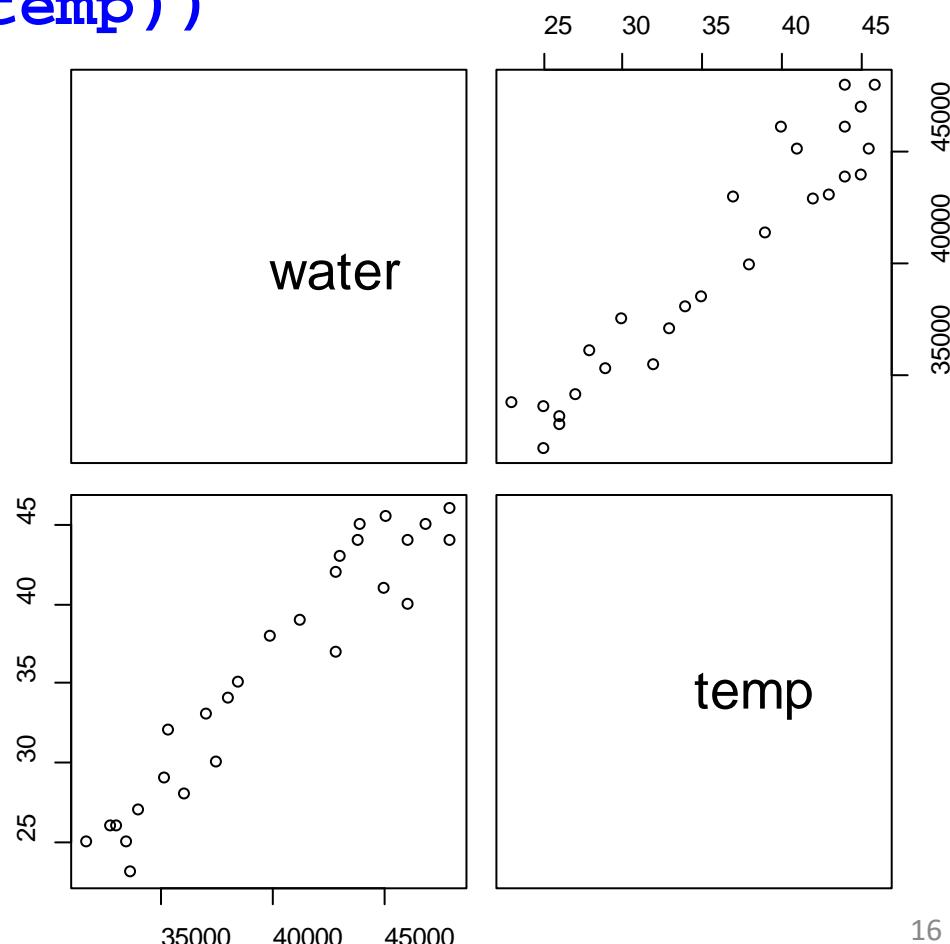
```
> scatter.smooth(water, temp, lpars = list(col =  
"red", lwd = 3, lty = 3))
```



## Matrix Scatter plot

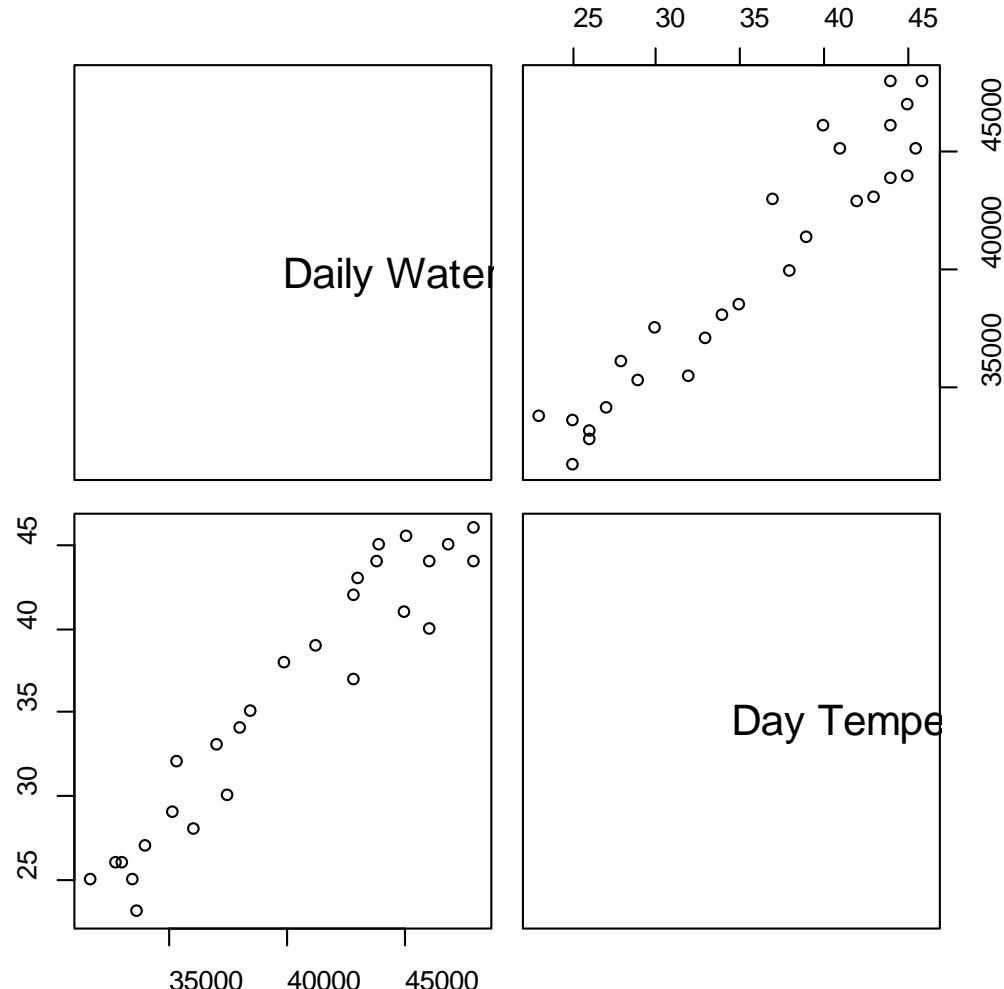
The command `pairs( )` allows the simple creation of a matrix of scatter plots.

```
> pairs( cbind(water,temp) )
```



## Matrix Scatter plot

```
> pairs( cbind(water,temp), labels=c("Daily Water Demand", "Day Temperature") )
```



## 3 Dimensional Scatter Plot:

`scatterplot3d()` Plots a three dimensional (3D) point cloud

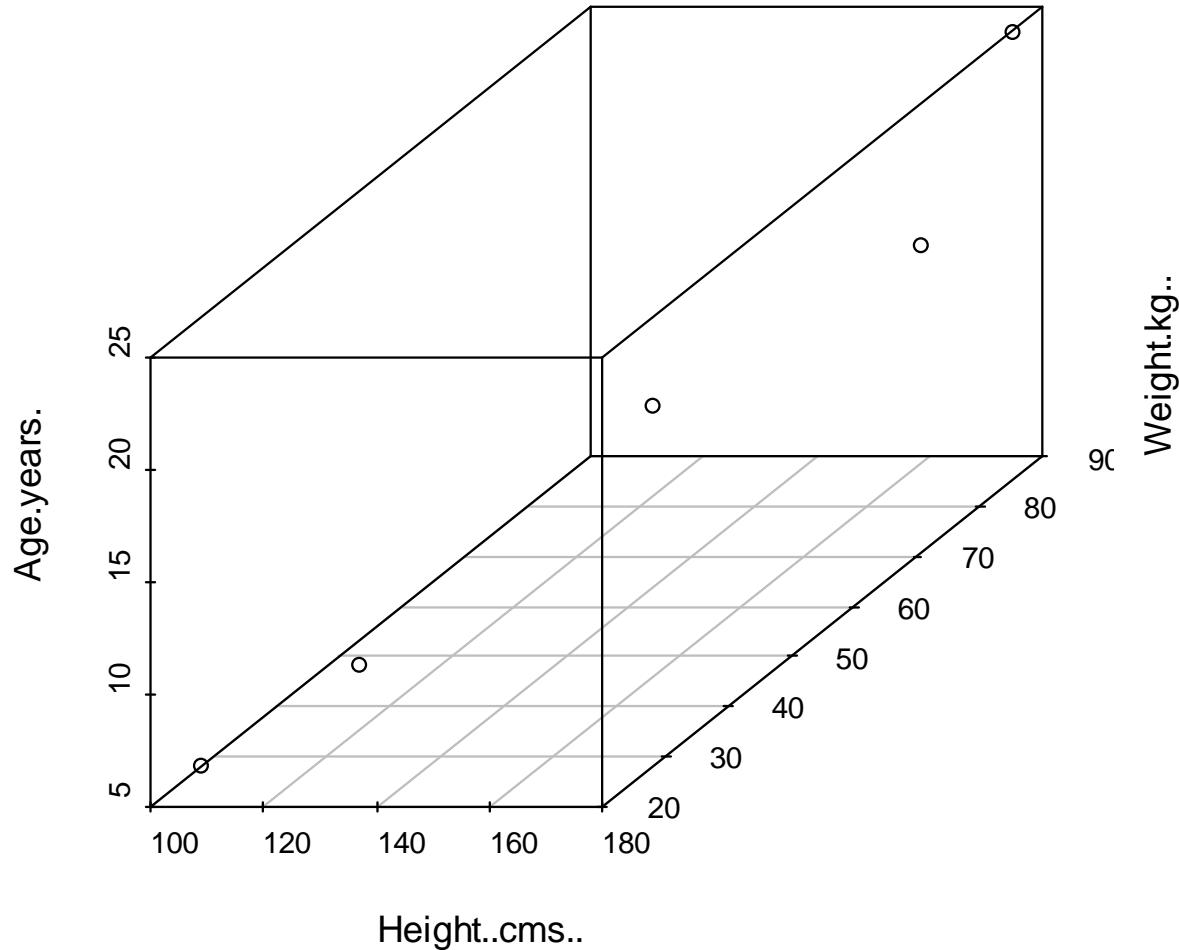
```
> install.packages("scatterplot3d")  
  
> library(scatterplot3d)  
  
> setwd("C:/RCourse/")  
  
> data3d <- read.csv("data-age-height-weight.csv")  
  
> data3d
```

Height...cms... Weight.kg... Age.years.

1	100	28	5
2	120	35	8
3	150	55	15
4	176	74	18
5	180	85	25

## 3 Dimensional Scatter Plot:

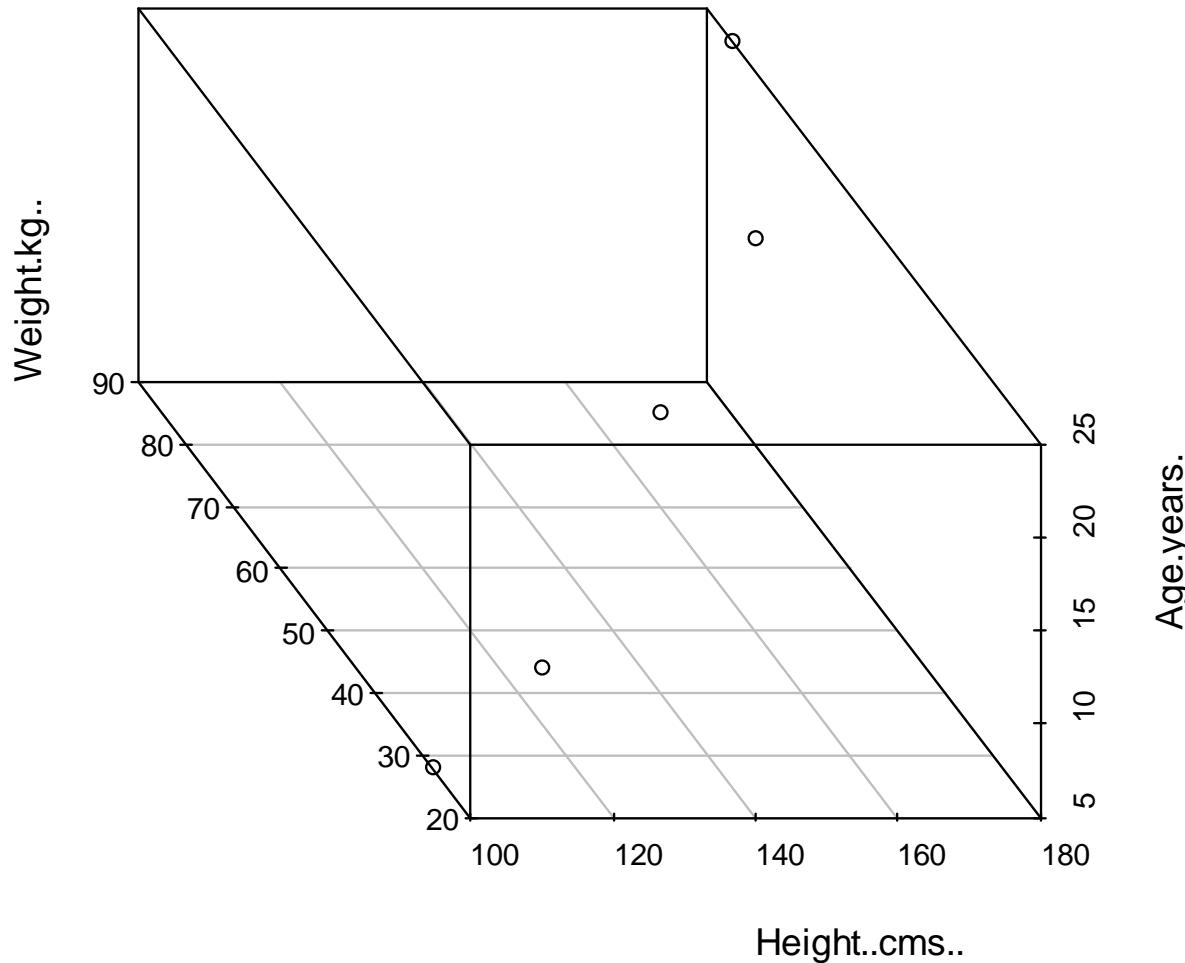
> `scatterplot3d(data3d[,1:3])`



## 3 Dimensional Scatter Plot:

Direction of the figure can be changed.

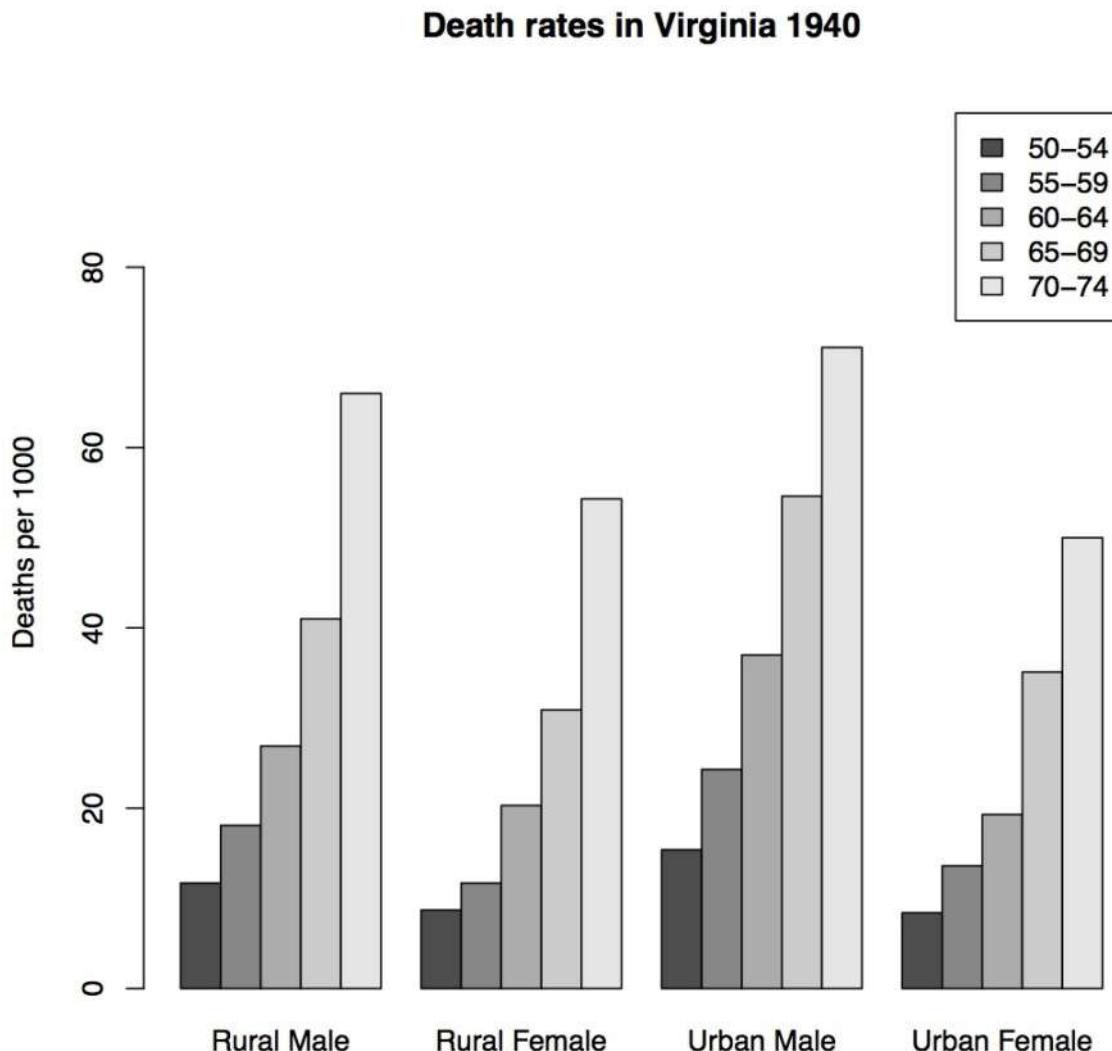
```
> scatterplot3d(data3d[,1:3], angle = 120)
```



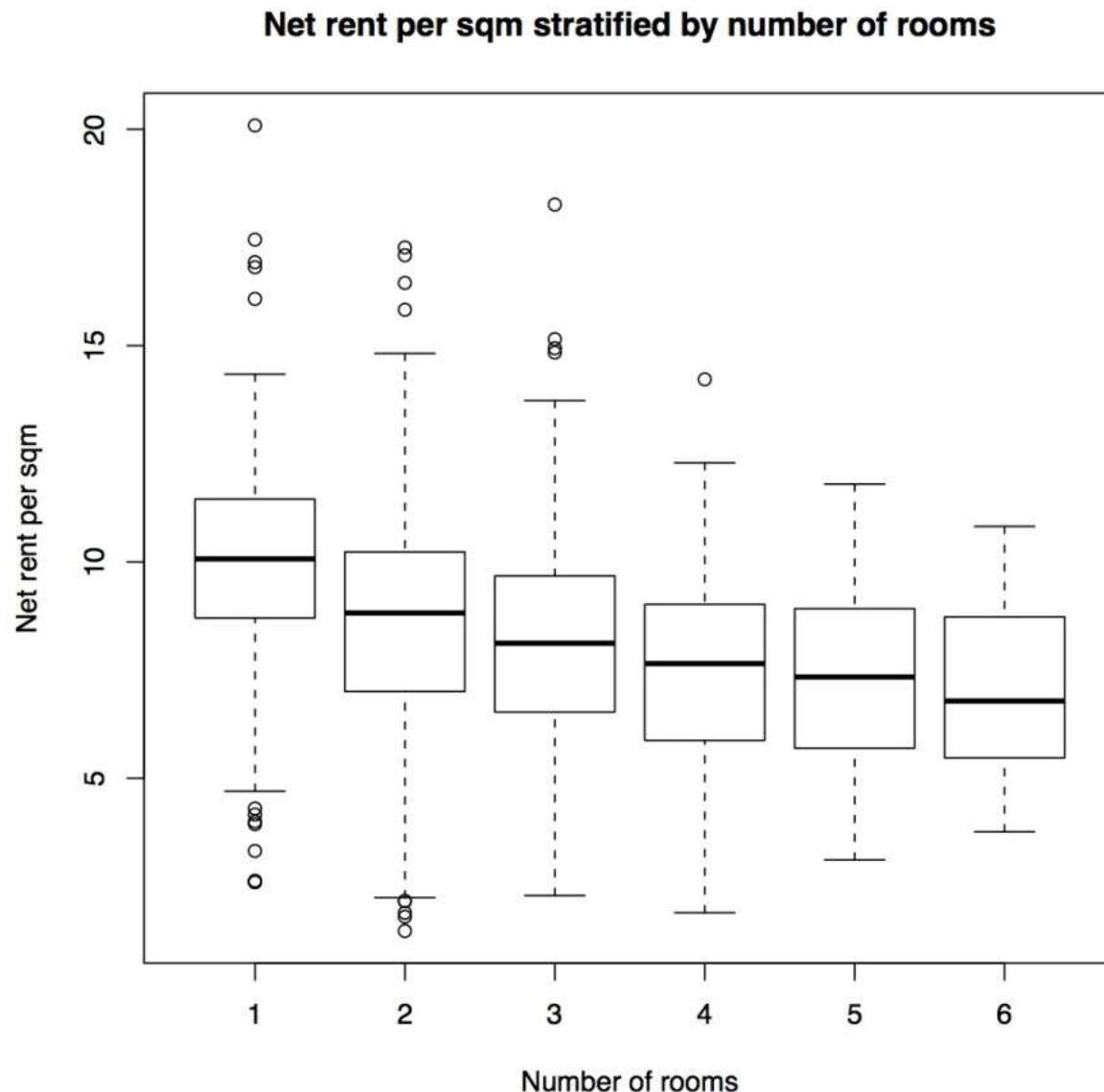
## More functions

- `contour()` for contour lines
- `dotchart()` for dot charts (replacement for bar charts)
- `image()` pictures with colors as third dimension
- `mosaicplot()` mosaic plot for (multidimensional) diagrams of categorical variables (contingency tables)
- `persp()` perspective surfaces over the x–y plane

# Multiple Bar plots are possible



# Grouped box plots are possible



# **Introduction to R Software**

## **Introduction to Statistical Functions**

**:::**

### **Correlation**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Descriptive statistics:**

**First hand tools which gives first hand information.**

- **Central tendency of data**
- **Variation in data**
- **Structure and shape of data tendency**
- **Relationship study (correlation coefficient, rank correlation, correlation ratio, regression etc.)**

## **Bivariate Data**

**Quantitative measures provide quantitative measure of relationship.**

**Graphical plots provide first hand visual information about the nature and degree of relationship between two variables.**

**Relationship can be linear or nonlinear.**

# Bivariate Data

**x, y:** Two data vectors

**Data**  $x = (x_1, x_2, \dots, x_n)$   $y = (y_1, y_2, \dots, y_n)$

**Covariance**  $\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$

**cov(x,y): covariance between x and y**

**Variance**  $\text{var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$

**var(x): Variance of x**

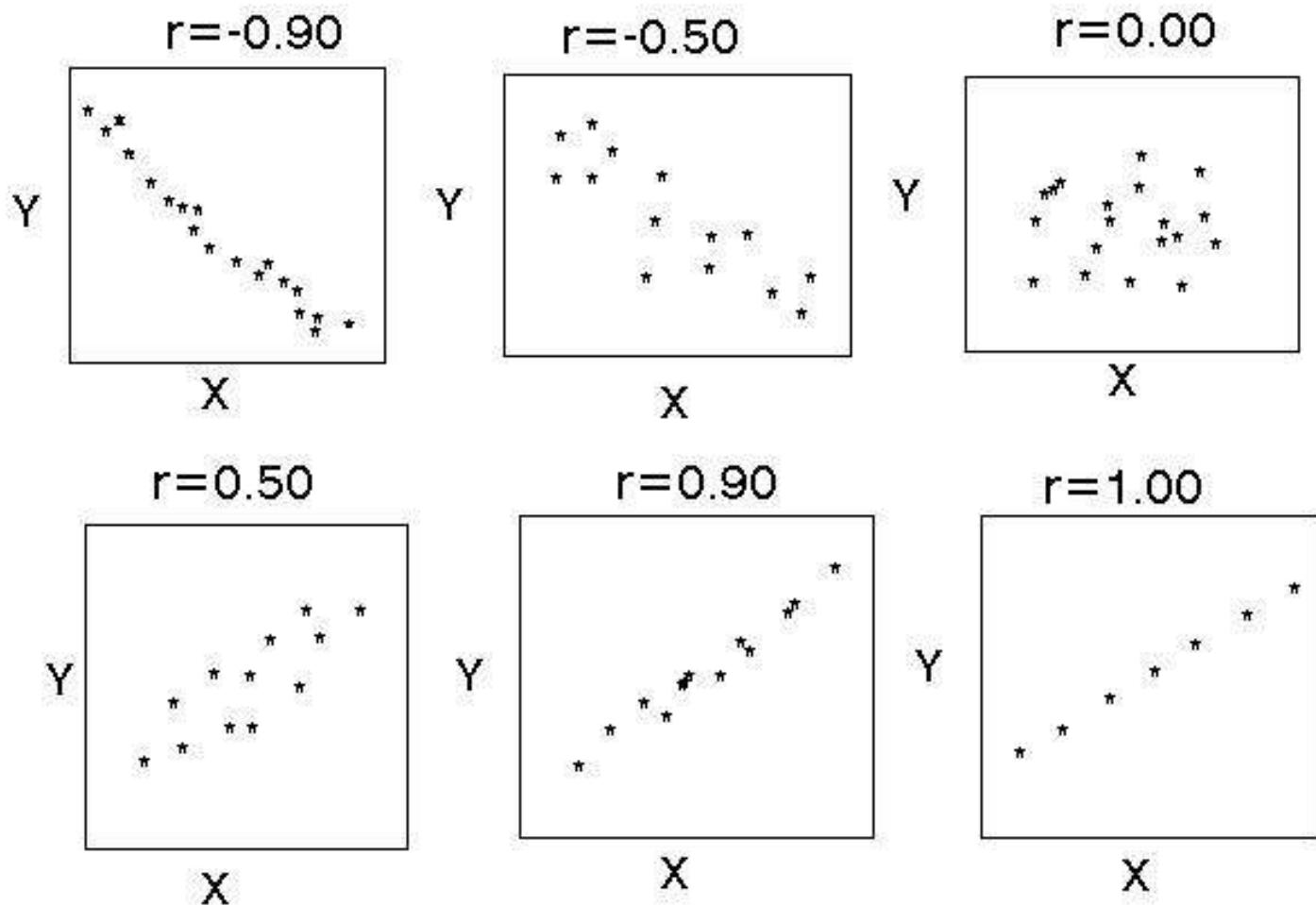
# Correlation coefficient

Measures the degree of linear relationship between the two variables.

$$r_{xy} = \frac{\text{cov}(x, y)}{\sqrt{\text{var}(x) \text{ var}(y)}} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}$$

$$-1 \leq r_{xy} \leq 1$$

**cor(x, y): correlation between x and y**



**Example:**

**Covarianve:**

```
> cov( c(1,2,3,4), c(1,2,3,4) )  
[1] 1.666667
```

R R Console

```
> cov( c(1,2,3,4), c(1,2,3,4) )  
[1] 1.666667
```

```
> cov( c(1,2,3,4), c(-1,-2,-3,-4) )  
[1] -1.666667
```

R R Console

```
> cov( c(1,2,3,4), c(-1,-2,-3,-4) )  
[1] -1.666667
```

## Example:

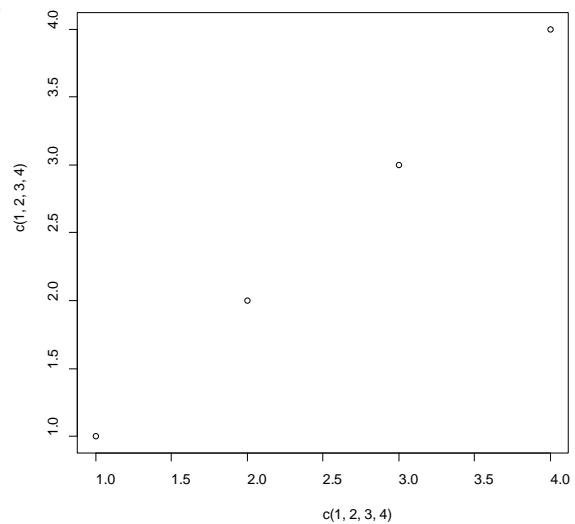
### Correlation coefficient:

#### Exact positive linear dependence

```
> cor( c(1,2,3,4), c(1,2,3,4) )  
[1] 1
```

R R Console

```
> cor( c(1,2,3,4), c(1,2,3,4) )  
[1] 1
```

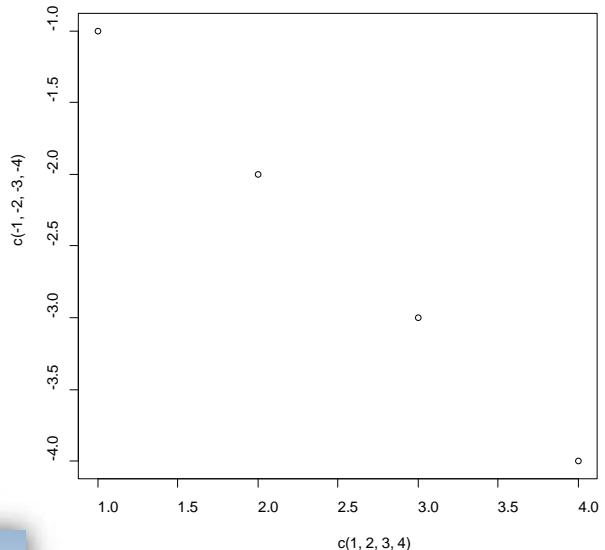


**Example:**

**Correlation coefficient:**

**Exact negative linear dependence**

```
> cor( c(1,2,3,4), c(-1,-2,-3,-4) )  
[1] -1
```



R Console

```
> cor( c(1,2,3,4), c(-1,-2,-3,-4) )  
[1] -1
```

## Example:

Daily water demand in a city depends upon weather temperature.

We know from experience that water consumption increases as weather temperature increases.

Data on 27 days is collected as follows:

Daily water demand (in million litres)

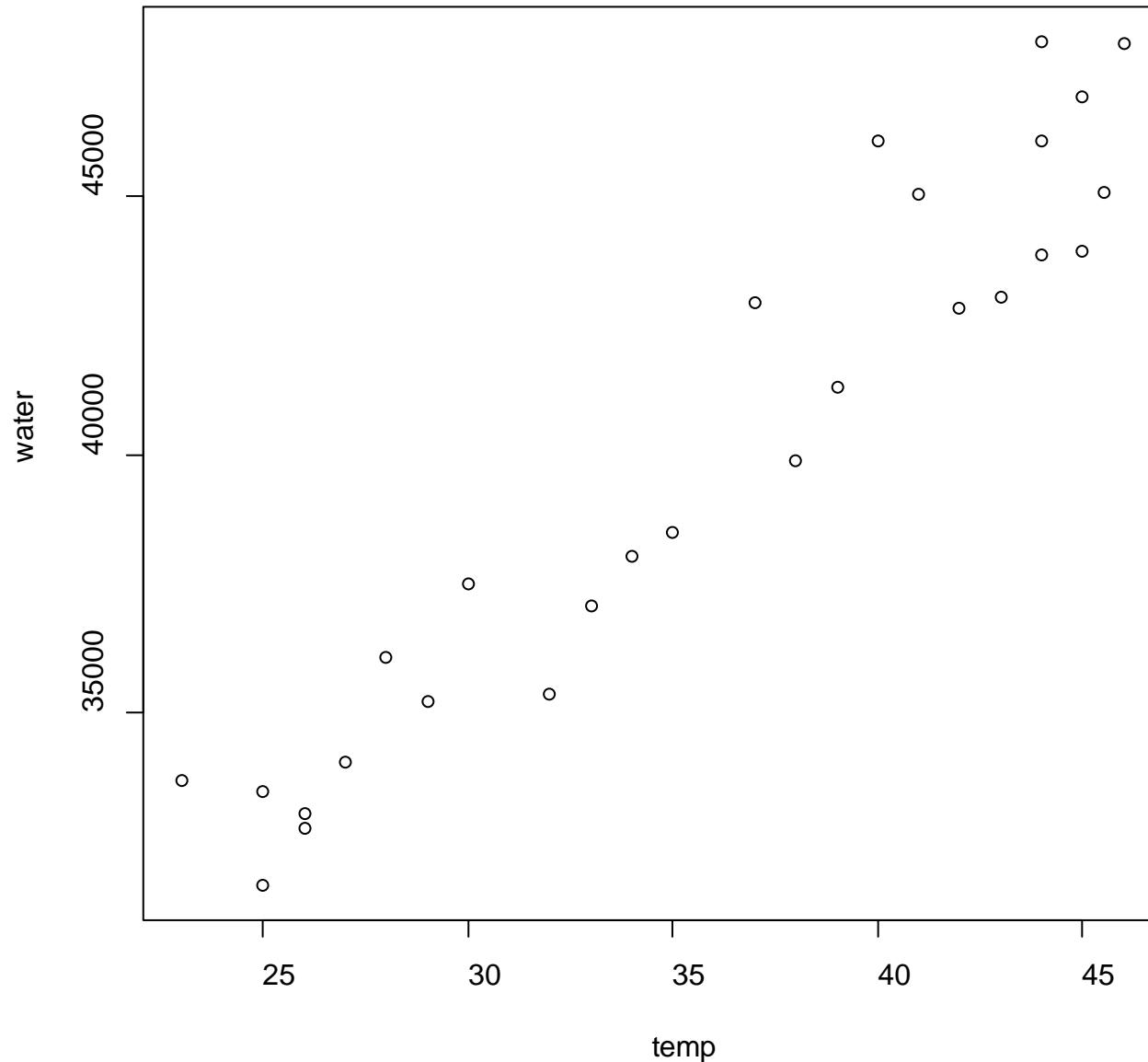
```
water <- c(33710,31666,33495,32758,34067,36069,  
37497,33044,35216, 35383,37066,38037,38495,  
39895,41311,42849,43038,43873,43923, 45078,  
46935,47951,46085,48003,45050,42924,46061)
```

Temperature (in centigrade)

```
temp <- c(23,25,25,26,27,28,30,26,29,32,33,34,  
35,38,39,42,43,44, 45,45.5,45,46,44,44,41,37,40)
```

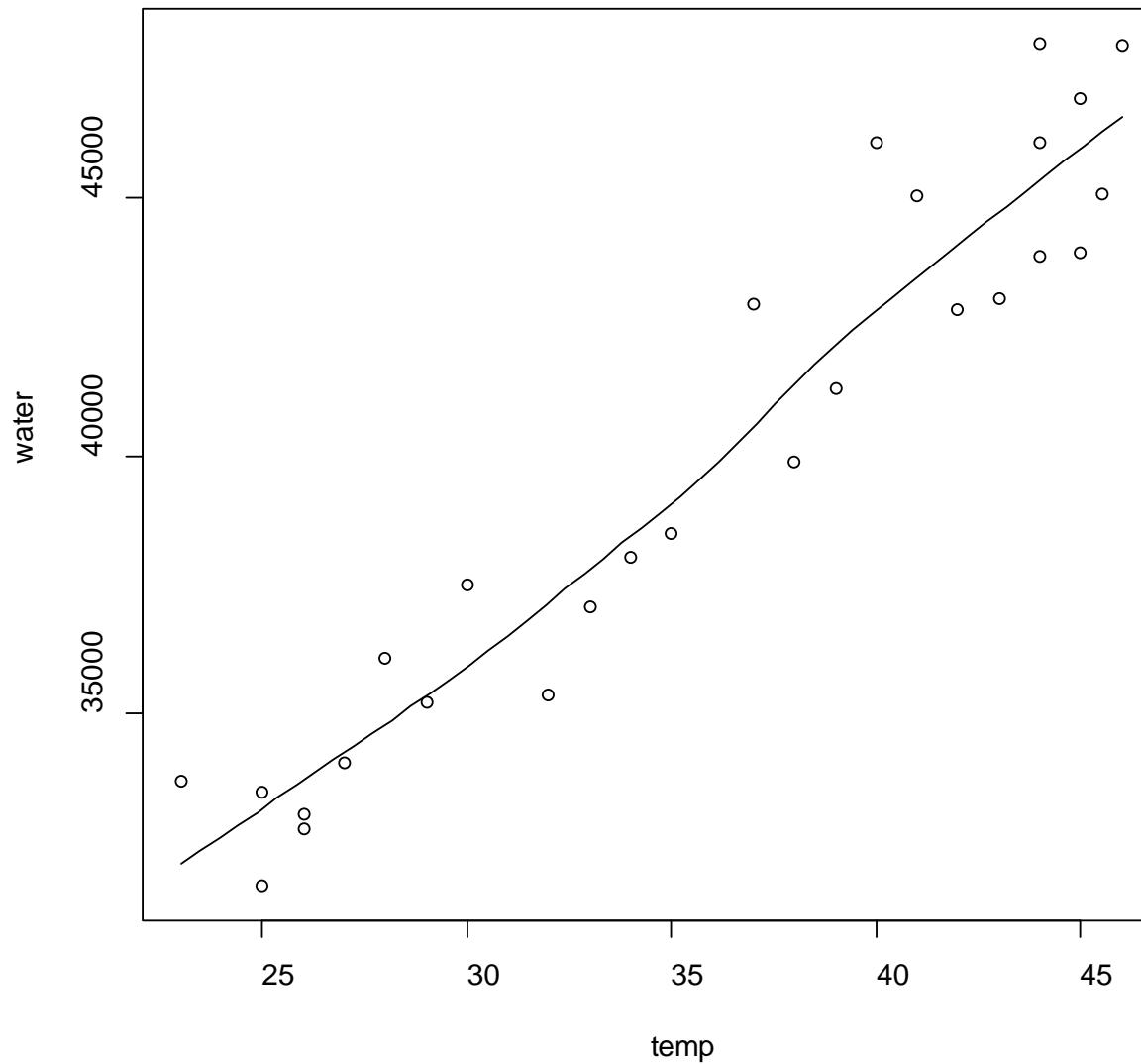
## Example:

```
> plot(temp, water)
```



## Example:

```
> scatter.smooth(temp,water)
```



## Data on Daily water demand

```
> cov(water, temp)  
[1] 39099.5
```

R Console

```
> cov(water, temp)  
[1] 39099.5
```

```
> cor(water, temp)  
[1] 0.9567599
```

R Console

```
> cor(water, temp)  
[1] 0.9567599
```

# **Introduction to R Software**

## **Introduction to Programming with Examples**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## **Steps to write a programme**

- A programme is a set of instructions or commands which are written in a sequence of operations i.e., what comes first and what comes after that.
- The objective of a programme is to obtain a defined outcome based on input variables.
- The computer is instructed to perform the defined task.

## **Steps to write a programme**

- Computer is an obedient worker but it has its own language.
- We do not understand computer's language and computer does not understand our language.
- The software help us and works like an interpreter between us and computer.

## **Steps to write a programme**

- We say something in software's language and software informs it to computer.
- Computer does the task and informs back to software.
- The software translates it to our language and informs us.

## Steps to write a programme

- Programme in R is written as a function using **function**.
- Write down the objective, i.e., what we want to obtain as an outcome.
- Translate it in the language of R.
- Identify the input and output variables.
- Identify the nature of input and output variables, i.e., numeric, string, factor, matrix etc.

# Steps to write a

# Steps to write a programme

## Tips:

- ❖ Loops usually slower the speed of programmes, so better is to use vectors and matrices.
  - ❖ Use # symbol to write comment to understand the syntax.
  - ❖ Use the variable names which are easy to understand.
  - ❖ Don't forget to initialize the variables.

## Example 1

Suppose we want to compute

$$\frac{\sum_{i=1}^n x_i^2}{\sum_{i=1}^n y_i^2} \quad \text{and} \quad \sum_{i=1}^n \left( \frac{x_i}{y_i} \right)^2$$

**Data**  $x_1, x_2, \dots, x_n$

$y_1, y_2, \dots, y_n$

**x, y:** Two data vectors

## Example 1

Input variables : **x, y, n** (if **x** and **y** have different number of observations, choose different numbers, say **n1** and **n2**)

Output variables: **g, h**, 
$$g = \frac{\sum_{i=1}^n x_i^2}{\sum_{i=1}^n y_i^2} \quad \text{and} \quad h = \sum_{i=1}^n \left( \frac{x_i}{y_i} \right)^2$$

We need summation, so use **sum** function or alternatively compute it through vectors.

## Example 1

```
# Remove all data
rm(list = ls())

# Define input data vectors, for example
x = c(10,20,30)
y = c(1,2,3)

+++++START OF FUNCTION+++++
example1 <- function(x,y)

# Start of function body
{
# First give all other input variables

# Computation of number of observations
n <- length(x)
```

CONTD...

## Example 1

CONTD...

```
#Initialize the values to store squared values
x1 <- 0
y1 <- 0
z1 <- 0

#Start of loop
for (i in 1:n)
{
# Define x1, y1 and z1 to store their squares
  x1[i] <- x[i]^2
  y1[i] <- y[i]^2
  z1[i] <- (x[i]/y[i])^2
#End of loop
}
```

CONTD...<sub>11</sub>

## Example 1

CONTD...

```
# Obtain the sum of squared quantities
sum_square_x <- sum(x1)
sum_square_y <- sum(y1)
sum_square_z <- sum(z1)

# Computation of g and h
g <- sum_square_x/sum_square_y
h <- sum_square_z

# Format the output
cat("The value of g and h are", g, "and", h,
"\n", )
}

+++++END OF FUNCTION++++++
```

## Example 1: At a glance

```
example1 <- function(x,y)
{
  n <- length(x)
  x1 <- 0
  y1 <- 0
  z1 <- 0
  for (i in 1:n)
  {
    x1[i] <- x[i]^2
    y1[i] <- y[i]^2
    z1[i] <- (x[i]/y[i])^2
  }
  sum_square_x <- sum(x1)
  sum_square_y <- sum(y1)
  sum_square_z <- sum(z1)
  g <- sum_square_x/sum_square_y
  h <- sum_square_z
  cat("The value of g and h are", g, "and", h,
      "respectively", "\n")
}
```

## Example 1

```
R R Console

> example1 <- function(x,y)
+   # Start of function body
+ {
+   # First give all other input variables
+   # Computation of number of observations
+   n <- length(x)
+   #Initialize the values to store squared values
+   x1 <- 0
+   y1 <- 0
+   z1 <- 0
+   #Start of loop
+   for (i in 1:n)
+   {
+     # Define x1, y1 and z1 to store their squares
+     x1[i] <- x[i]^2
+     y1[i] <- y[i]^2
+     z1[i] <- (x[i]/y[i])^2
+     #End of loop
+   }
+   # Obtain the sum of squared quantities
+   sum_square_x <- sum(x1)
+   sum_square_y <- sum(y1)
+   sum_square_z <- sum(z1)
+   # Computation of g and h
+   g <- sum_square_x/sum_square_y
+   h <- sum_square_z
+   # Format the output
+   cat("The value of g and h are", g, "and", h, "respectively", "\n")
+   # End of function
+ }
```

## Example 1

```
R Console

> example1
function(x,y)
  # Start of function body
{
  # First give all other input variables
  # Computation of number of observations
  n <- length(x)
  #Initialize the values to store squared values
  x1 <- 0
  y1 <- 0
  z1 <- 0
  #Start of loop
  for (i in 1:n)
  {
    # Define x1, y1 and z1 to store their squares
    x1[i] <- x[i]^2
    y1[i] <- y[i]^2
    z1[i] <- (x[i]/y[i])^2
    #End of loop
  }
  # Obtain the sum of squared quantities
  sum_square_x <- sum(x1)
  sum_square_y <- sum(y1)
  sum_square_z <- sum(z1)
  # Computation of g and h
  g <- sum_square_x/sum_square_y
  h <- sum_square_z
  # Format the output
  cat("The value of g and h are", g, "and", h, "respectively", "\n")
  # End of function
}
```

## Example 1

```
> x=c(10,20,30)  
> y=c(1,2,3)  
> example1(x,y)
```

The value of g and h are 100 and 300 respectively

```
> x=c(67,87,26,85,6,45)  
> y=c(54,64,22,94,20,88)  
> example1(x,y)
```

The value of g and h are 0.8996568 and 5.953203  
respectively

Just by changing the values of **x** and **y**, one can get required different outcomes.

## Example 1

```
R R Console
> x=c(10,20,30)
> y=c(1,2,3)
> example1(x,y)
The value of g and h are 100 and 300 respectively
>
> x=c(67,87,26,85,6,45)
> y=c(54,64,22,94,20,88)
> example1(x,y)
The value of g and h are 0.8996568 and 5.953203 respectively
```

# **Introduction to R Software**

## **More Examples of Programming**

**Shalabh**

**Department of Mathematics and Statistics**

**Indian Institute of Technology Kanpur**

## Example 2

Suppose we want to compute

$$f(x, y) = \frac{\left(\frac{x + \ln y}{y}\right)^2}{5 + \left(\frac{x + \ln y}{y}\right)^3} \left[ \exp\left(\frac{x + \ln y}{y}\right) \right]^{\frac{2}{3}}$$

This can be written as

$$f(x, y) = \frac{(g(x, y))^2}{5 + (g(x, y))^3} \left[ \exp(g(x, y)) \right]^{\frac{2}{3}}$$

where  $g(x, y) = \frac{x + \ln y}{y}$

## Example 2

Input variables :  $x, y$

Output variables: :  $f$

We break this function in two components –

- Compute  $g(x,y)$  as a function and then
- compute  $f(x,y)$  by calling  $g(x,y)$ .

## Example 2

```
# Remove all data  
rm(list = ls())  
  
# Define input data vectors  
x  
y
```

CONTD...

## Example 2

CONTD...

```
# define g(x,y)
g <- function(x,y)
# Start of function
{
  (x+log(y))/y
# End of function
}
```

$$g(x, y) = \frac{x + \ln y}{y}$$

```
# define f(x,y)
f<-function(x,y)
{
  (((g(x,y))^2)/(5+(g(x,y))^3))*(exp(g(x,y)))^(2/3)
}
```

$$f(x, y) = \frac{(g(x, y))^2}{5 + (g(x, y))^3} \left[ \exp(g(x, y)) \right]^{\frac{2}{3}}$$

## Example 2: At a glance

```
# define g(x,y)  
  
g <- function(x,y)  
{  
  (x+log(y))/y  
}
```

+++++

```
# define f(x,y)
```

```
f<-function(x,y)  
{  
  (((g(x,y))^2)/(5+(g(x,y))^3))*(exp(g(x,y)))^(2/3)  
}  
# g(x,y) must have been defined earlier.
```

## Example 2

```
R R Console

> # define g(x,y)
> g <- function(x,y)
+   # Start of function
+ {
+   (x+log(y))/y
+   # End of function
+ }
>
> # define f(x,y)
> f<-function(x,y)
+ {
+   (((g(x,y))^2) / (5+(g(x,y))^3)) * (exp(g(x,y)))^(2/3)
+ }
```

## Example 2

R Console

```
> g
function(x,y)
  # Start of function
{
  (x+log(y))/y
  # End of function
}
> f
function(x,y)
{
  (((g(x,y))^2) / (5+(g(x,y))^3)) * (exp(g(x,y)))^(2/3)
}
```

## Example 2

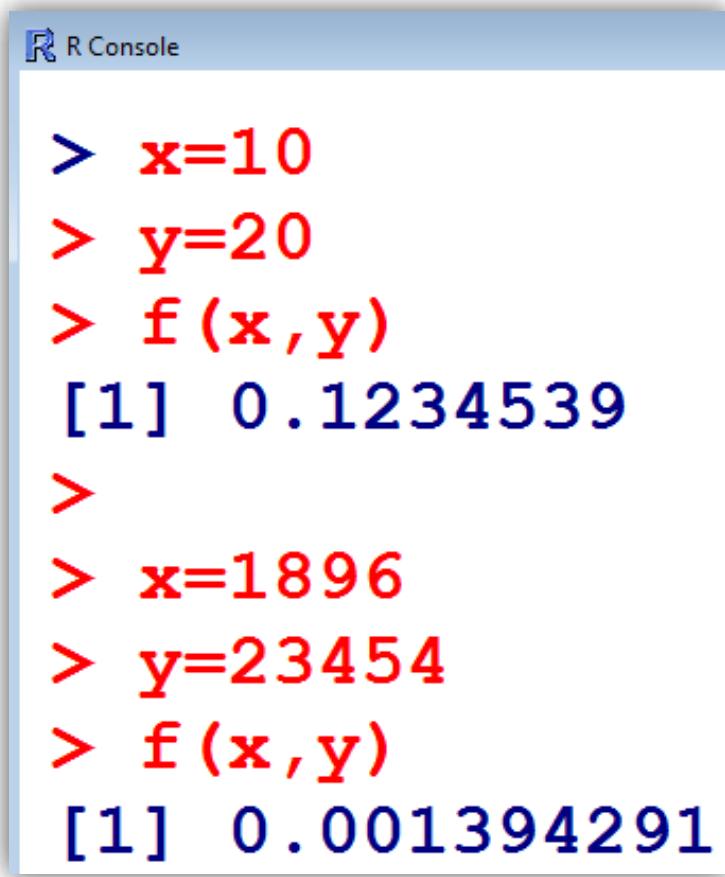
```
> x=10  
> y=20  
> f(x,y)  
[1] 0.1234539
```

```
> x=1896  
> y=23454  
> f(x,y)  
[1] 0.001394291
```

There is no need to calculate the value of  $g(x,y)$ .

Just by changing the values of  $x$  and  $y$ , one can get different required outcomes.

## Example 2



The image shows a screenshot of an R console window titled "R Console". It contains two distinct blocks of R code and their corresponding outputs.

```
> x=10
> y=20
> f(x,y)
[1] 0.1234539

>
> x=1896
> y=23454
> f(x,y)
[1] 0.001394291
```

## Example 3

Suppose we want to compute

$$f(x) = \begin{cases} \exp\left(\frac{x + \ln(1 + x^3)}{x^2}\right) & \text{if } x > 0 \\ 10 & \text{if } x = 0 \\ \frac{2 + x^3}{x} & \text{if } x < 0 \end{cases}$$

and plot with line over a values of x as a sequence starting from -1 to 5 and increasing it by 0.2.

## Example 3

**Input variable : x**

**Output variable: f**

```
# Remove all data
```

```
rm(list = ls())
```

```
# Define input data
```

```
x
```

**CONTD...**

## Example 3

CONTD...

```
f<-function(x)
{
  if(x>0) {exp((x+log(1+x^3))/x^2)}
  else if(x==0) {10}
  else {(2+x^3)/x}
}
```

$$f(x) = \begin{cases} \exp\left(\frac{x + \ln(1 + x^3)}{x^2}\right) & \text{if } x > 0 \\ 10 & \text{if } x = 0 \\ \frac{2 + x^3}{x} & \text{if } x < 0 \end{cases}$$

CONTD...

## Example 3

CONTD...

```
h <- function()
# Start of function
{
# Generation of data on x
x<-seq(-1,5,by=0.2)
# Initialization of y to store values of f(x)
y<-0
```

CONTD...

## Example 3

CONTD...

```
# Generation of f(x) values corresponding to x
for(i in 1:length(x))
{
  y[i]<-f(x[i])
}
# length(x) and length(y) must be same to plot
# y=f(x) with respect to x
plot(x,y,type = "l")
```

### Example 3: At a glance

```
f<-function(x)
{
  if(x>0) {exp((x+log(1+x^3))/x^2)}
  else if(x==0) {10}
  else {(2+x^3)/x}
}

h <- function()
{
  x <- seq(-1,5,by=0.2)
  y <- 0

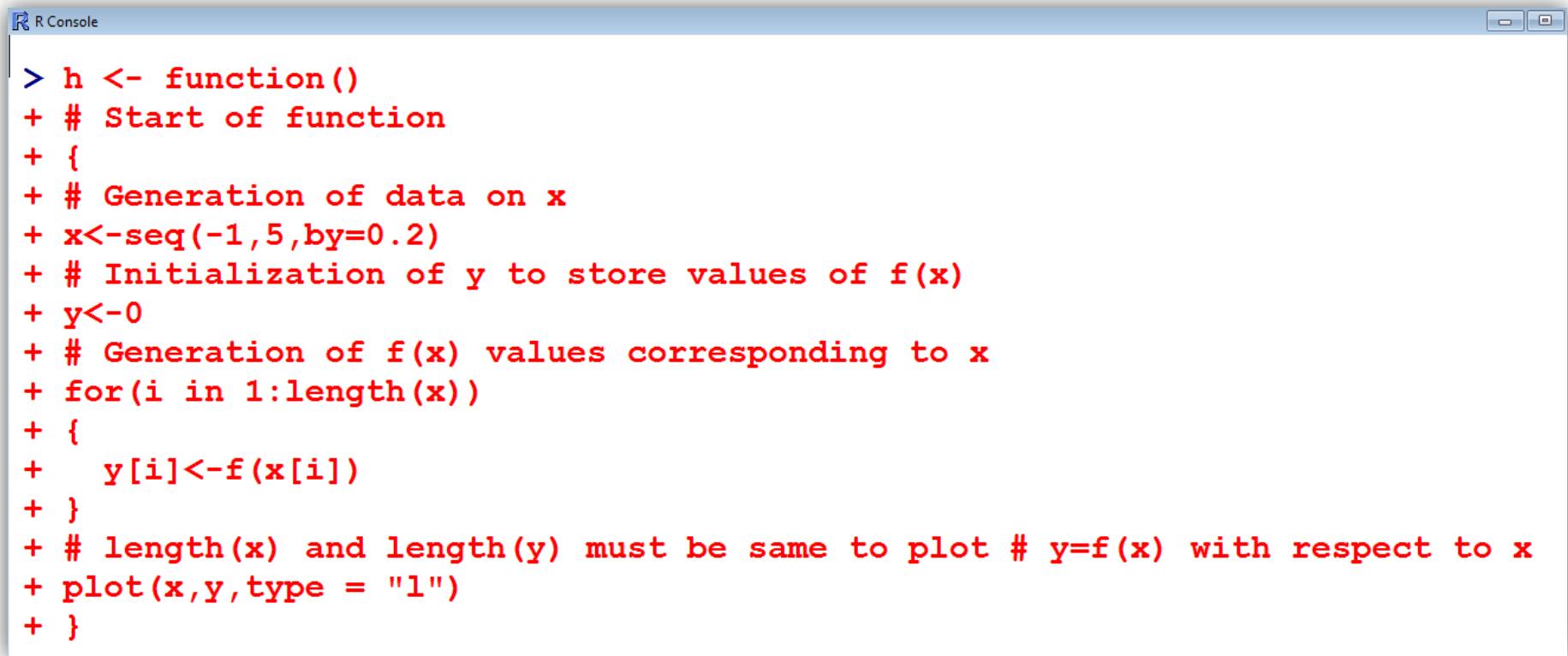
  for(i in 1:length(x))
  {
    y[i] <- f(x[i])
  }
  plot(x,y,type = "l")
}
```

## Example 3

R R Console

```
> f<-function(x)
+ {
+   if(x>0) {exp((x+log(1+x^3))/x^2)}
+   else if(x==0) {10}
+   else {(2+x^3)/x}
+ }
```

## Example 3



The screenshot shows the R Console window with the following R code:

```
> h <- function()
+ # Start of function
+ {
+ # Generation of data on x
+ x<-seq(-1,5,by=0.2)
+ # Initialization of y to store values of f(x)
+ y<-0
+ # Generation of f(x) values corresponding to x
+ for(i in 1:length(x))
+ {
+   y[i]<-f(x[i])
+ }
+ # length(x) and length(y) must be same to plot # y=f(x) with respect to x
+ plot(x,y,type = "l")
+ }
```

## Example 3

R Console

```
> f
function(x)
{
  if(x>0) {exp((x+log(1+x^3))/x^2)}
  else if(x==0) {10}
  else {(2+x^3)/x}
}
```

## Example 3

```
R Console

> h
function()
# Start of function
{
# Generation of data on x
x<-seq(-1,5,by=0.2)
# Initialization of y to store values of f(x)
y<-0
# Generation of f(x) values corresponding to x
for(i in 1:length(x))
{
  y[i]<-f(x[i])
}
# length(x) and length(y) must be same to plot # y=f(x) with respect to x
plot(x,y,type = "l")
```

### Example 3

```
> f(123)
[1] 1.009126
>
> f(-123)
[1] 15128.98
>
> f(0)
[1] 10

> f(8)
[1] 1.249201
> f(-4)
[1] 15.5
> f(0)
[1] 10
```

R Console

```
> f(123)
[1] 1.009126
>
> f(-123)
[1] 15128.98
>
> f(0)
[1] 10
>
>
> f(8)
[1] 1.249201
> f(-4)
[1] 15.5
> f(0)
[1] 10
```

## Example 3

> h( )

