

UNIT WISE SUBJECT NOTES

UNIT-I

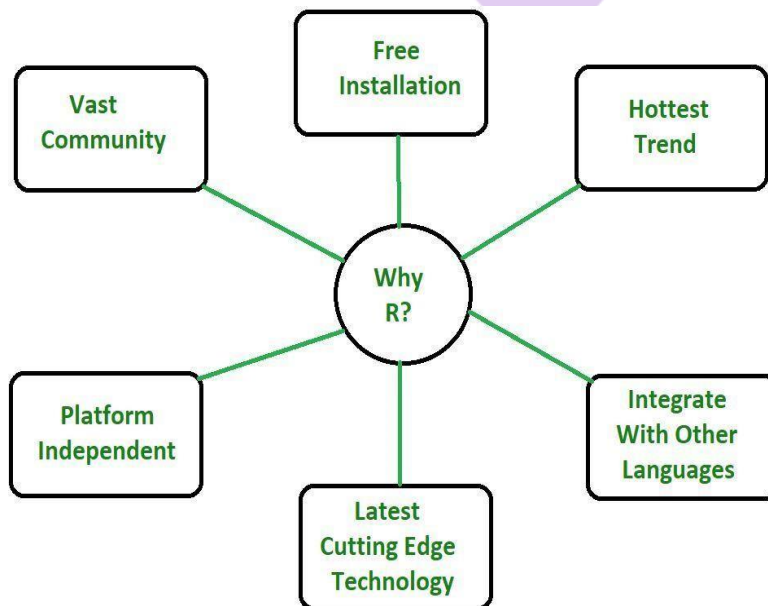
R Programming Language – Introduction

OVERVIEW OF R

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

It was designed by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project conceives in 1992, with an initial version released in 1995 and a stable beta version in 2000.

Why R Programming Language?



R programming is used as a leading tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.

It's a platform-independent language. This means it can be applied to all operating system.

It's an open-source free language. That means anyone can install it in any organization without purchasing a license.

## R-PROGRAMMING(DS3101PC/AM3101PC)

R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.

The R programming language has a vast community of users and it's growing day by day.

R is currently one of the most requested programming languages in the Data Science job market that makes it the hottest trend nowadays.

### Features of R Programming Language

#### Statistical Features of R:

**Basic Statistics:** The most common basic statistics terms are the mean, mode, and median. These are all known as “Measures of Central Tendency.” So using the R language we can measure central tendency very easily.

**Static graphics:** R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.

**Probability distributions:** Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distribution such as Binomial Distribution, Normal Distribution, Chi-squared Distribution and many more.

**Data analysis:** It provides a large, coherent and integrated collection of tools for data analysis.

#### Programming Features of R:

**R Packages:** One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.

**Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages ddR and multidplyr used for distributed programming in R were released in November 2015.

#### Programming in R:

Since R is much similar to other widely used languages syntactically, it is easier to code and learn in R. Programs can be written in R in any of the widely used IDE like R Studio, Rattle, Tinn-R, etc. After writing the program save the file with the extension .r. To run the program use the following command on the command line:

**R file\_name.r**

**Example:**

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# R program to print Welcome to GFG!  
# Below line will print "Welcome to GFG!"  
cat("Welcome to GFG!")
```

Output:

Welcome to GFG!

**Advantages of R:**

R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.

As R programming language is an open source. Thus, you can run R anywhere and at any time.

R programming language is suitable for GNU/Linux and Windows operating system.

R programming is cross-platform which runs on any operating system.

In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

**Disadvantages of R:**

In the R programming language, the standard of some packages is less than perfect.

Although, R commands give little pressure to memory management. So R programming language may consume all available memory.

In R basically, nobody to complain if something doesn't work.

R programming language is much slower than other programming languages such as Python and MATLAB.

**Applications of R:**

We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.

R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.

R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.

Tech giants like Google, Facebook, Bing, Twitter, Accenture, Wipro and many more using R nowadays.

## R-PROGRAMMING(DS3101PC/AM3101PC)

R and Python both play a major role in data science. It becomes confusing for any newbie to choose the better or the most suitable one among the two, R and Python. So take a look at [R vs Python for Data Science](#) to choose which language is more suitable for data science.

### R DATATYPES AND OBJECTS

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are –

1. Vectors
2. Lists
3. Matrices
4. Arrays
5. Factors
6. Data Frames

The simplest of these objects is the vector object and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type	Example	Verify
Logical	TRUE, FALSE	<a href="#">Live Demo</a> <pre>v &lt;- TRUE print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "logical"</pre>
Numeric	12.3, 5, 999	<a href="#">Live Demo</a> <pre>v &lt;-23.5 print(class(v))</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

		it produces the following result – [1] "numeric"
Integer	2L, 34L, 0L	<a href="#">Live Demo</a>  <code>v &lt;-2L</code> <code>print(class(v))</code> it produces the following result – [1] "integer"
Complex	3 + 2i	<a href="#">Live Demo</a>  <code>v &lt;-2+5i</code> <code>print(class(v))</code> it produces the following result – [1] "complex"
Character	'a', "good", "TRUE", '23.4'	<a href="#">Live Demo</a>  <code>v &lt;- "TRUE"</code> <code>print(class(v))</code> it produces the following result – [1] "character"
Raw	"Hello" is stored as 48 65 6c 6c 6f	<a href="#">Live Demo</a>  <code>v &lt;-charToRaw("Hello")</code> <code>print(class(v))</code>  it produces the following result – [1] "raw"

In R programming, the very basic data types are the R-objects called vectors which hold elements of different classes as shown above. Please note in R the number of classes is not confined to only the above six types. For example, we can use many atomic vectors and create an array whose class will become array.

### Vectors

When you want to create vector with more than one element, you should use c() function which means to combine the elements into a vector.

#### Live Demo

```
# Create a vector.
```

```
apple <- c('red','green',"yellow")
```

```
print(apple)
```

```
# Get the class of the vector.
```

```
print(class(apple))
```

When we execute the above code, it produces the following result –

```
[1] "red" "green" "yellow"
```

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

### Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

#### Live Demo

```
# Create a list.
```

```
list1 <- list(c(2,5,3),21.3,sin)
```

```
# Print the list.
```

```
print(list1)
```

When we execute the above code, it produces the following result –

```
[[1]]
```

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

```
[[2]]
```

```
[1] 21.3
```

```
[[3]]
```

```
function (x) .Primitive("sin")
```

### Matrices

## R-PROGRAMMING(DS3101PC/AM3101PC)

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
```

```
M=matrix( c('a','a','b','c','b','a'),nrow=2,ncol=3,byrow= TRUE)
```

```
print(M)
```

When we execute the above code, it produces the following result –

```
[,1] [,2] [,3]
[1,] "a" "a" "b"
[2,] "c" "b" "a"
```

### Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.
```

```
a <- array(c('green','yellow'),dim= c(3,3,2))
```

```
print(a)
```

When we execute the above code, it produces the following result –

```
,, 1
[1,] [2,] [3,]
[1,] "green" "yellow" "green"
[2,] "yellow" "green" "yellow"
[3,] "green" "yellow" "green"
```

```
,, 2
```

```
[1,] [2,] [3,]
[1,] "yellow" "green" "yellow"
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[2,] "green" "yellow" "green"
```

```
[3,] "yellow" "green" "yellow"
```

### Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modeling.

Factors are created using the factor() function. The nlevels functions gives the count of levels.

```
# Create a vector.
```

```
apple_colors<- c('green','green','yellow','red','red','red','green')
```

```
# Create a factor object.
```

```
factor_apple<- factor(apple_colors)
```

```
# Print the factor.
```

```
print(factor_apple)
```

```
print(nlevels(factor_apple))
```

When we execute the above code, it produces the following result –

```
[1] green green yellow red redred green
```

```
Levels: green red yellow
```

```
[1] 3
```

### Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the data.frame() function.

```
# Create the data frame.
```

```
BMI <- data.frame(  
  gender =c("Male","Male","Female"),  
  height =c(152,171.5,165),
```



## R-PROGRAMMING(DS3101PC/AM3101PC)

```
weight =c(81,93,78),  
Age=c(42,38,26)  
)  
print(BMI)
```

When we execute the above code, it produces the following result –

```
gender height weight Age  
1 Male 152.0 81 42  
2 Male 171.5 93 38  
3 Female 165.0 78 26
```

### READING AND WRITING DATA IN R

#### Reading Data in R

Here we will discuss about reading and writing data in R. For reading, (importing) data into R following are some functions.

**read.table()**, and **read.csv()**, for reading tabular data

**readLines()** for reading lines of a text file

**source()** for reading in R code files (inverse of dump)

**dget()** for reading in R code files (inverse of dput)

**load()** for reading in saved workspaces.

Writing Data to files

Following are few functions for writing (exporting) data to files.

**write.table()**, and **write.csv()** exports data to wider range of file format including csv and tab-delimited.

**writeLines()** write text lines to a text-mode connection.

**dump()** takes a vector of names of R objects and produces text representations of the objects on a file (or connection). A dump file can usually be sourced into another R session.

**dput()** writes an ASCII text representation of an R object to a file (or connection) or uses one to recreate the object.

**save()** writes an external representation of R objects to the specified file.

## R-PROGRAMMING(DS3101PC/AM3101PC)

Reading data files with **read.table()**

The **read.table()** function is one of the most commonly used functions for reading data into R. It has a few important arguments.

**file**, the name of a file, or a connection

**header**, logical indicating if the file has a header line

**sep**, a string indicating how the columns are separated

**colClasses**, a character vector indicating the class of each column in the data set

**nrows**, the number of rows in the dataset

**comment.char**, a character string indicating the comment character

**skip**, the number of lines to skip from the beginning

**stringsAsFactors**, should character variables be coded as factors?

**read.table()** and **read.csv()** Examples

```
data <- read.table("foo.txt")
```

```
data <- read.table("D:\\datafiles\\mydata.txt")
```

```
data <- read.csv("D:\\datafiles\\mydata.csv")
```

R will automatically skip lines that begin with a #, figure out how many rows there are (and how much memory needs to be allocated). R also figure out what type of variable is in each column of the table.

Writing data files with **write.table()**

Following are few important arguments usually used in **write.table()** function.

**x**, the object to be written, typically a data frame

**file**, the name of the file which the data are to be written to

**sep**, the field separator string

**col.names**, a logical value indicating whether the column names of **x** are to be written along with **x**, or a character vector of column names to be written

**row.names**, a logical value indicating whether the row names of **x** are to be written along with **x**, or a character vector of row names to be written

**na**, the string to use for missing values in the data

**write.table()** and **write.csv()** Examples

```
x <- data.frame(a = 5, b = 10, c = pi)
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
write.table(x, file = "data.csv", sep = ",")
```

```
write.table(x, "c:\\mydata.txt", sep = "\\t")
```

```
write.csv(x, file = "data.csv")
```

### ESSENTIALS OF THE R LANGUAGE

Essential Fundamentals of R is an integrated program that draws from a variety of introductory topics and courses to provide participants with a solid base of knowledge with which to use R software for any intended purpose. No statistical knowledge, programming knowledge, or experience with R software is necessary. Essential Fundamentals of R (7 sessions) covers those important introductory topics basic to using R functions and data objects for any purpose: installing R and RStudio; interactive versus batch use of R; reading data and datasets into R; essentials of scripting; getting help in R; primitive data types; important data structures; using functions in R; writing user-defined functions; the 'apply' family of functions in R; data set manipulation: and subsetting, and row and column selection. Most sessions present "hands-on" material that demonstrate the execution of R 'scripts' (sets of commands) and utilize many extended examples of R functions, applications, and packages for a variety of common purposes. RStudio, a popular, open source Integrated Development Environment (IDE) for developing and using R applications, is also utilized in the program, supplemented with R-based direct scripts (e.g. 'command-line prompts') when necessary.

#### Who this course is for:

Anyone who is interested in learning to use R software who is relatively new (or 'brand new') to using R

People who wish to learn the essential fundamentals of using R including data types and structures, inputting and outputting data and files, writing user-defined functions, and manipulating data sets

College undergrads and/or graduate students who are looking for an alternative to using SAS or SPSS software

Professionals engaged in quantitative analyses and/or data analyses tasks who seek an alternative to using SAS and/or SPSS software.

#### Install R and RStudio – A Step-by-Step Guide for Beginners

Looking to install R and RStudio in different operating systems? Follow the step by step procedure and download R and RStudio in an easy way.

Check out the installation process of R and RStudio on Linux operating systems, Microsoft Windows, and Mac OS X and also look at some useful R packages that enhance R's capabilities.

How to Install R

In this tutorial, you will learn to-

## R-PROGRAMMING(DS3101PC/AM3101PC)

[Install R and RStudio on Linux](#)

[Install R and RStudio on Windows](#)

[Install R and RStudio on Mac OS X](#)

[Some useful Packages in R](#)

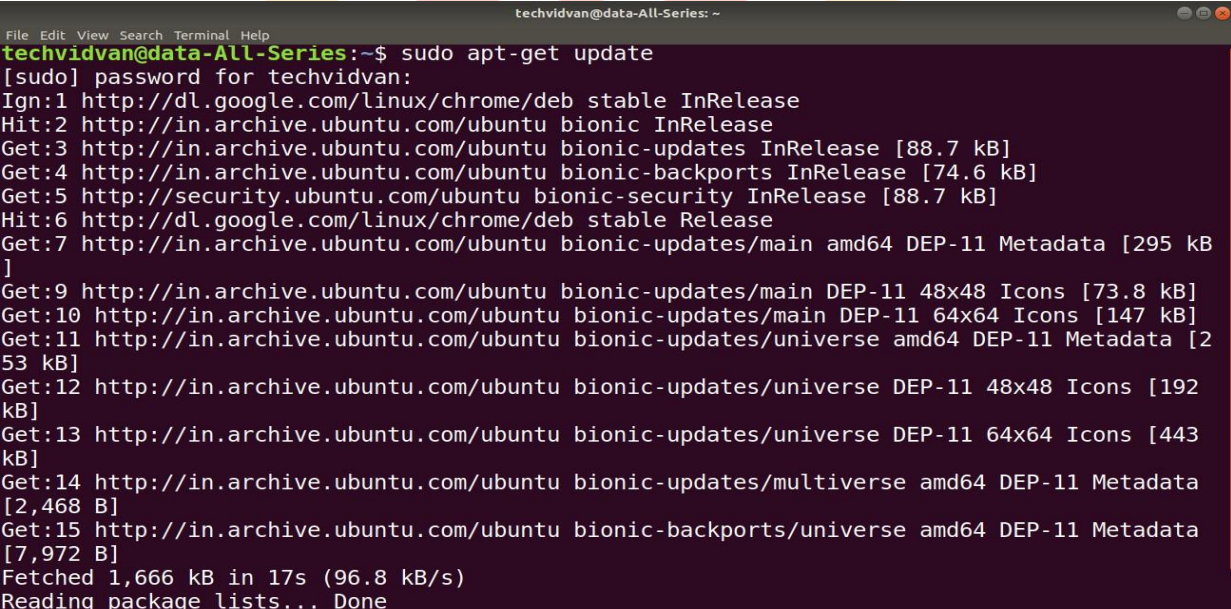
### Installing R and RStudio on Linux

Linux software is often distributed as source code and then compiled by package managers like apt or yum. To install R in Ubuntu, we will have to go through the following steps.

#### Install R on Linux

Install the R-base package using the following code

**sudo apt-get update**



```
techvidvan@data-All-Series: ~$ sudo apt-get update
[sudo] password for techvidvan:
Ign:1 http://dl.google.com/linux/chrome/deb stable InRelease
Hit:2 http://in.archive.ubuntu.com/ubuntu bionic InRelease
Get:3 http://in.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:4 http://in.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:5 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:6 http://dl.google.com/linux/chrome/deb stable Release
Get:7 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 DEP-11 Metadata [295 kB]
]
Get:9 http://in.archive.ubuntu.com/ubuntu bionic-updates/main DEP-11 48x48 Icons [73.8 kB]
Get:10 http://in.archive.ubuntu.com/ubuntu bionic-updates/main DEP-11 64x64 Icons [147 kB]
Get:11 http://in.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 DEP-11 Metadata [253 kB]
Get:12 http://in.archive.ubuntu.com/ubuntu bionic-updates/universe DEP-11 48x48 Icons [192 kB]
Get:13 http://in.archive.ubuntu.com/ubuntu bionic-updates/universe DEP-11 64x64 Icons [443 kB]
Get:14 http://in.archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 DEP-11 Metadata [2,468 B]
Get:15 http://in.archive.ubuntu.com/ubuntu bionic-backports/universe amd64 DEP-11 Metadata [7,972 B]
Fetched 1,666 kB in 17s (96.8 kB/s)
Reading package lists... Done
```

**sudo apt-get install r-base**

your roots to success...

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
techvidvan@data-All-Series: ~  
File Edit View Search Terminal Help  
techvidvan@data-All-Series:~$ sudo apt-get install r-base  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  ess r-doc-info | r-doc-pdf  
The following NEW packages will be installed:  
  r-base  
0 upgraded, 1 newly installed, 0 to remove and 347 not upgraded.  
Need to get 0 B/9,312 B of archives.  
After this operation, 60.4 kB of additional disk space will be used.  
Selecting previously unselected package r-base.  
(Reading database ... 174995 files and directories currently installed.)  
Preparing to unpack .../r-base_3.4.4-1ubuntu1_all.deb ...  
Unpacking r-base (3.4.4-1ubuntu1) ...  
Setting up r-base (3.4.4-1ubuntu1) ...  
techvidvan@data-All-Series:~$ clear  
  
techvidvan@data-All-Series:~$ █
```

After running the command, a confirmation prompt will appear. Answer it with a 'Y' for yes.

### Install RStudio on Linux

**Step – 1** Next comes installing RStudio. To install RStudio, go to [download RStudio](#), click on the download button for RStudio desktop, click the link for the latest R version for your OS and save the .deb file.

**Step – 2** Download and install the gdebi package using the following commands

```
sudo apt install gdebi
```

```
techvidvan@data-All-Series: ~  
File Edit View Search Terminal Help  
techvidvan@data-All-Series:~$ sudo apt install gdebi  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  gnome-icon-theme libgtk2-perl libpango-perl  
Suggested packages:  
  libgtk2-perl-doc  
The following NEW packages will be installed:  
  gdebi gnome-icon-theme libgtk2-perl libpango-perl  
0 upgraded, 4 newly installed, 0 to remove and 347 not upgraded.  
Need to get 10.3 MB of archives.  
After this operation, 19.3 MB of additional disk space will be used.  
Do you want to continue? [Y/n] y  
Get:1 http://in.archive.ubuntu.com/ubuntu bionic/universe amd64 gnome-icon-theme all 3.12.0-3 [9,620 kB]  
Get:2 http://in.archive.ubuntu.com/ubuntu bionic/universe amd64 gdebi all 0.9.5.7+nmu2 [23,7 kB]  
Get:3 http://in.archive.ubuntu.com/ubuntu bionic/universe amd64 libpango-perl amd64 1.227-2build1 [157 kB]  
Get:4 http://in.archive.ubuntu.com/ubuntu bionic/universe amd64 libgtk2-perl amd64 2:1.2499-2-1build1 [544 kB]  
Fetched 10.3 MB in 1s (7,981 kB/s)  
Selecting previously unselected package gnome-icon-theme.
```

Answer with a 'Y' for yes to confirm when prompted.

**Step – 3** Use the following commands to install the .deb package

## R-PROGRAMMING(DS3101PC/AM3101PC)

sudo gdebi /path/to/the/file/.deb

```
techvidvan@data-All-Series: ~
File Edit View Search Terminal Help
techvidvan@data-All-Series:~$ sudo gdebi ~/Downloads/rstudio-1.2.5001-amd64.deb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading state information... Done

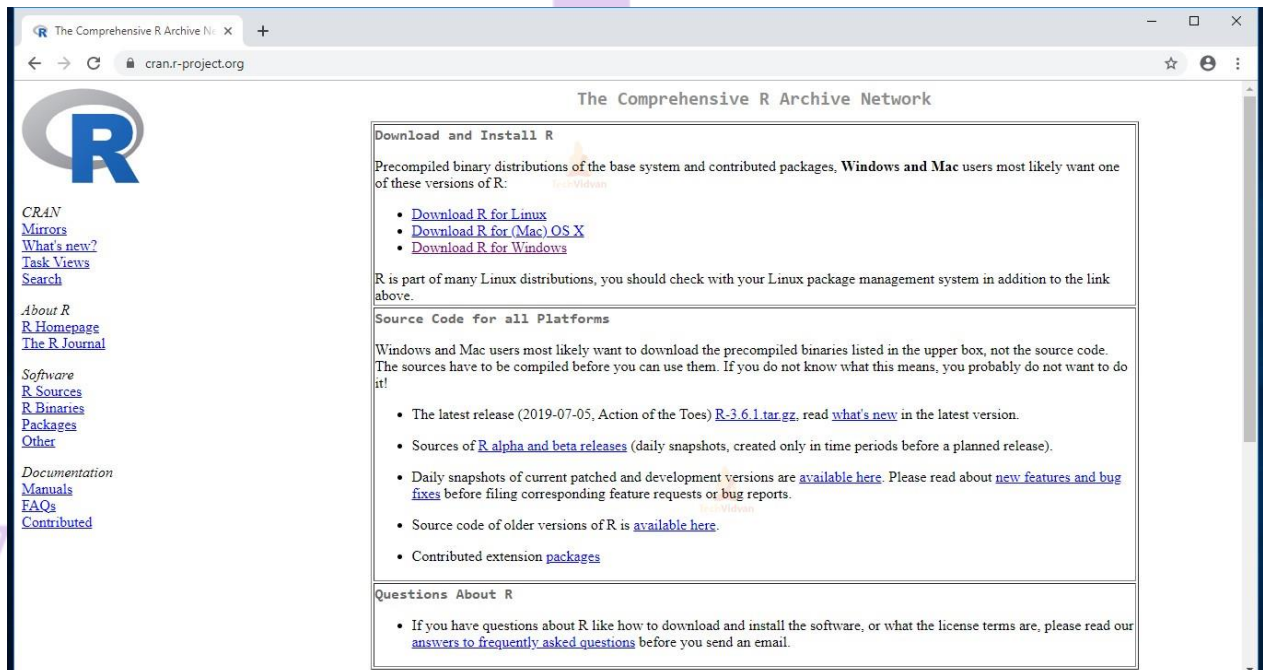
RStudio
RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, and workspace management.
Do you want to install the software package? [y/N]:y
(Reading database ... 181189 files and directories currently installed.)
Preparing to unpack ../rstudio-1.2.5001-amd64.deb ...
Unpacking rstudio (1.2.5001) over (1.2.5001) ...
Setting up rstudio (1.2.5001) ...
Processing triggers for gnome-menus (3.13.3-11ubuntu1) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.1) ...
Processing triggers for mime-support (3.60ubuntu1) ...
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for shared-mime-info (1.9-2) ...
techvidvan@data-All-Series:~$
techvidvan@data-All-Series:~$
techvidvan@data-All-Series:~$
```

### Installing R and RStudio on Windows

To install R and RStudio on windows, go through the following steps:

Install R on windows

Step – 1: Go to [CRAN R project website](https://cran.r-project.org/).

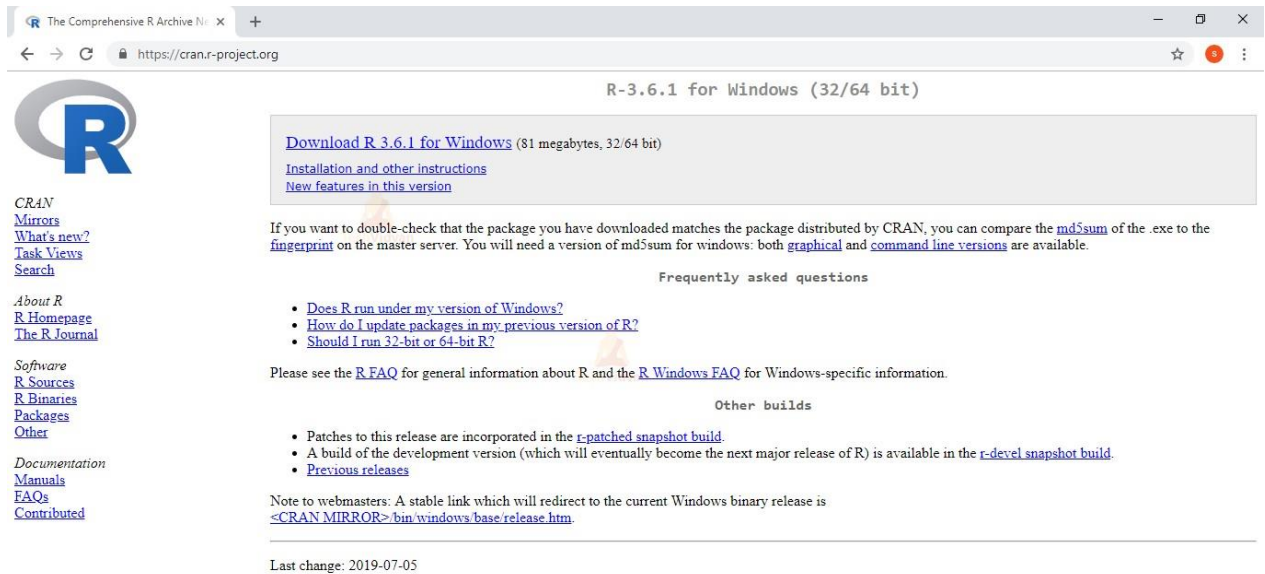


Step – 2: Click on the Download R for Windows link.

Step – 3: Click on the base subdirectory link or install R for the first time link.

## R-PROGRAMMING(DS3101PC/AM3101PC)

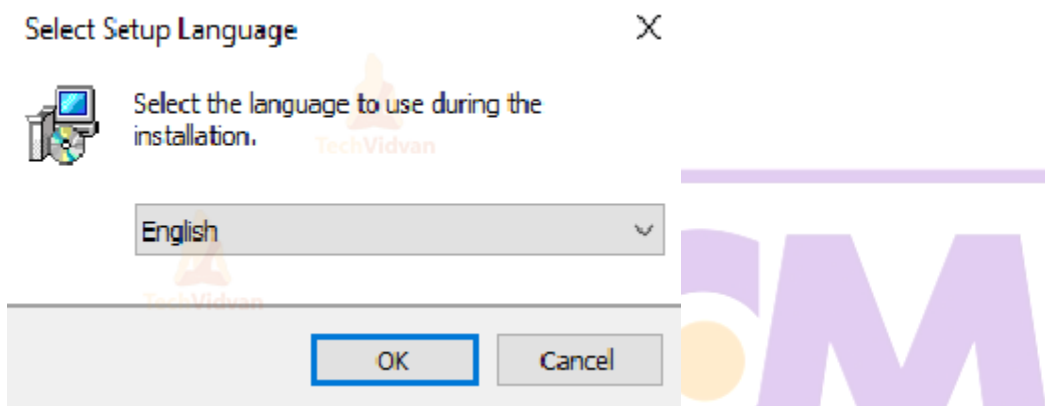
**Step – 4:** Click Download R X.X.X for Windows (X.X.X stand for the latest version of R. eg: 3.6.1) and save the executable .exe file.



The screenshot shows the CRAN website for R 3.6.1 for Windows (32/64 bit). The main content area includes a download link for R 3.6.1 for Windows (81 megabytes, 32/64 bit), installation and other instructions, and new features in this version. Below this, there is a section for frequently asked questions with three bullet points: "Does R run under my version of Windows?", "How do I update packages in my previous version of R?", and "Should I run 32-bit or 64-bit R?". There is also a section for other builds with two bullet points: "Patches to this release are incorporated in the r-patched snapshot build." and "A build of the development version (which will eventually become the next major release of R) is available in the r-devel snapshot build." At the bottom, there is a note to webmasters and a link to the current Windows binary release.

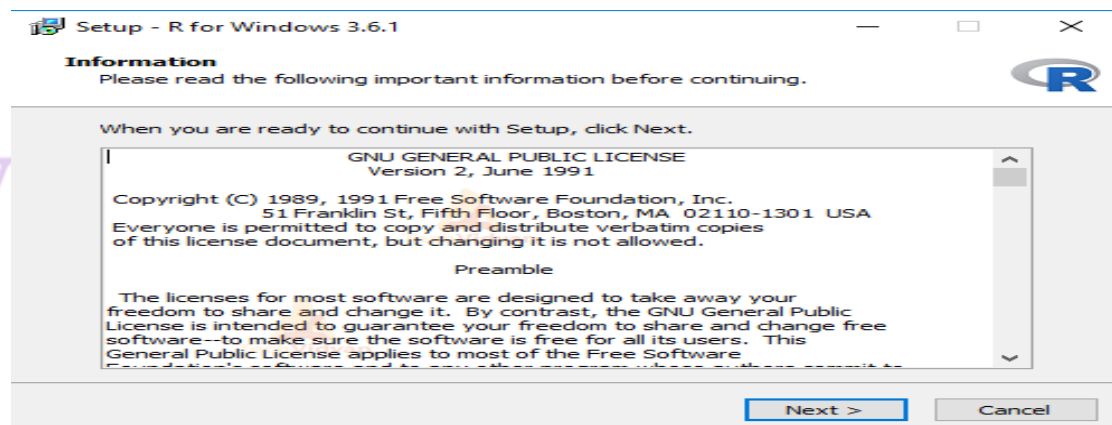
**Step – 5:** Run the .exe file and follow the installation instructions.

Select the desired language and then click Next.



The screenshot shows the "Select Setup Language" dialog box. The title bar reads "Select Setup Language". The main text says "Select the language to use during the installation." Below this, there is a dropdown menu with "English" selected. At the bottom, there are two buttons: "OK" and "Cancel". The "OK" button is highlighted with a blue border.

Read the license agreement and click Next.

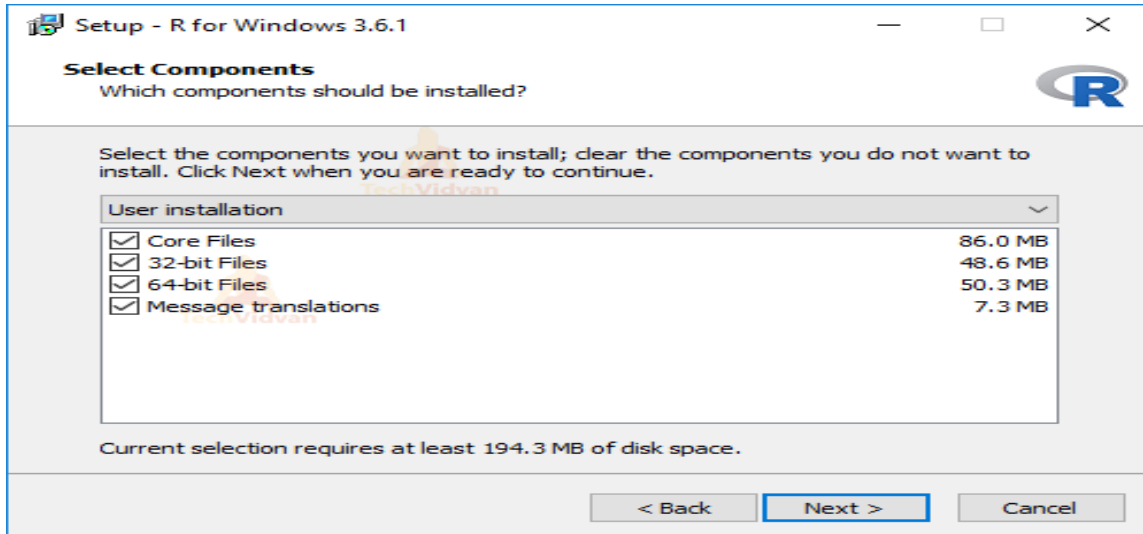


The screenshot shows the "Setup - R for Windows 3.6.1" dialog box. The title bar reads "Setup - R for Windows 3.6.1". The main text says "Information Please read the following important information before continuing." Below this, there is a text area containing the GNU General Public License (Version 2, June 1991) text. At the bottom, there are two buttons: "Next >" and "Cancel". The "Next >" button is highlighted with a blue border.

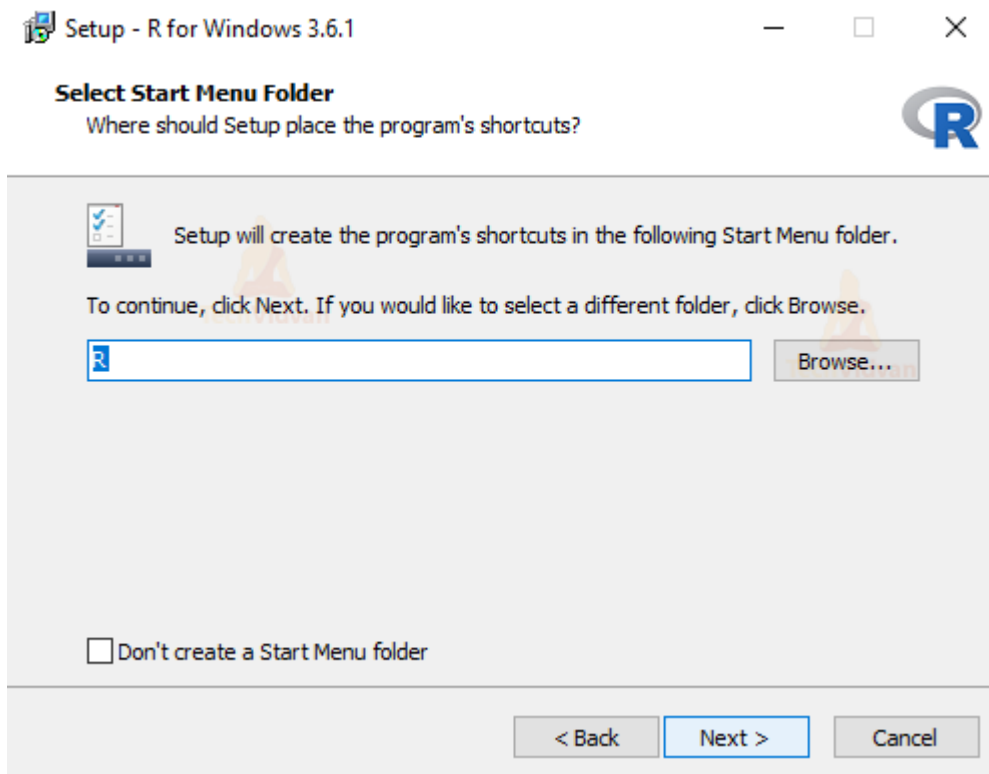
## R-PROGRAMMING(DS3101PC/AM3101PC)

Select the components you wish to install (it is recommended to install all the components).

Click Next.



Enter/browse the folder/path you wish to install R into and then confirm by clicking Next.

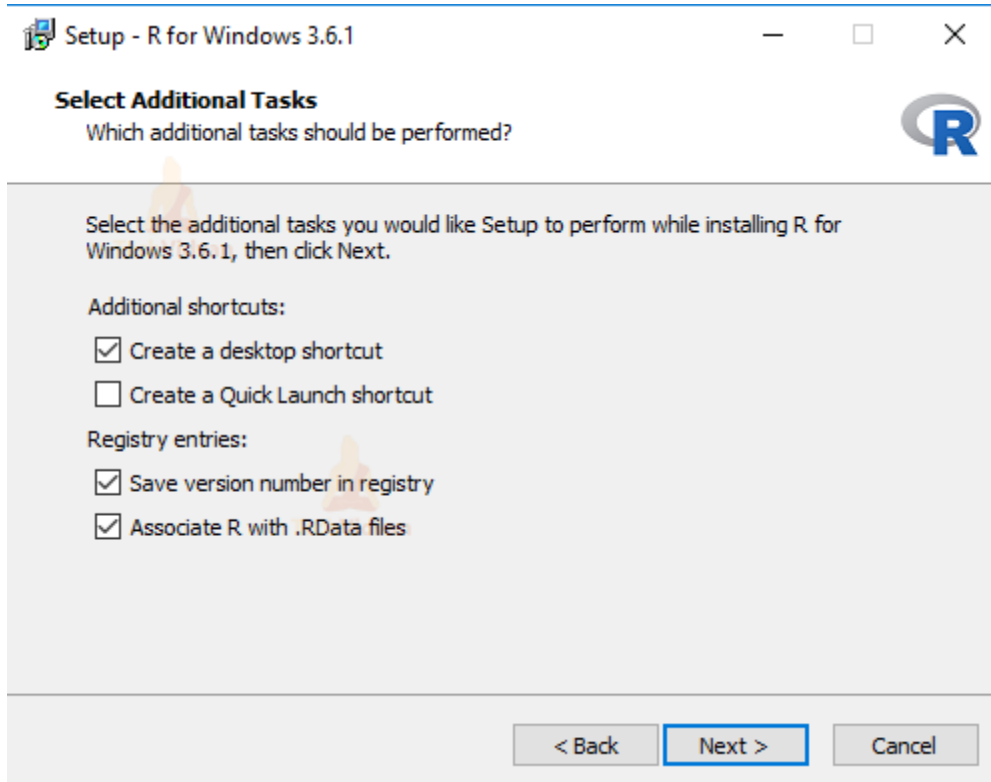


Select additional tasks like creating desktop shortcuts etc. then click Next.

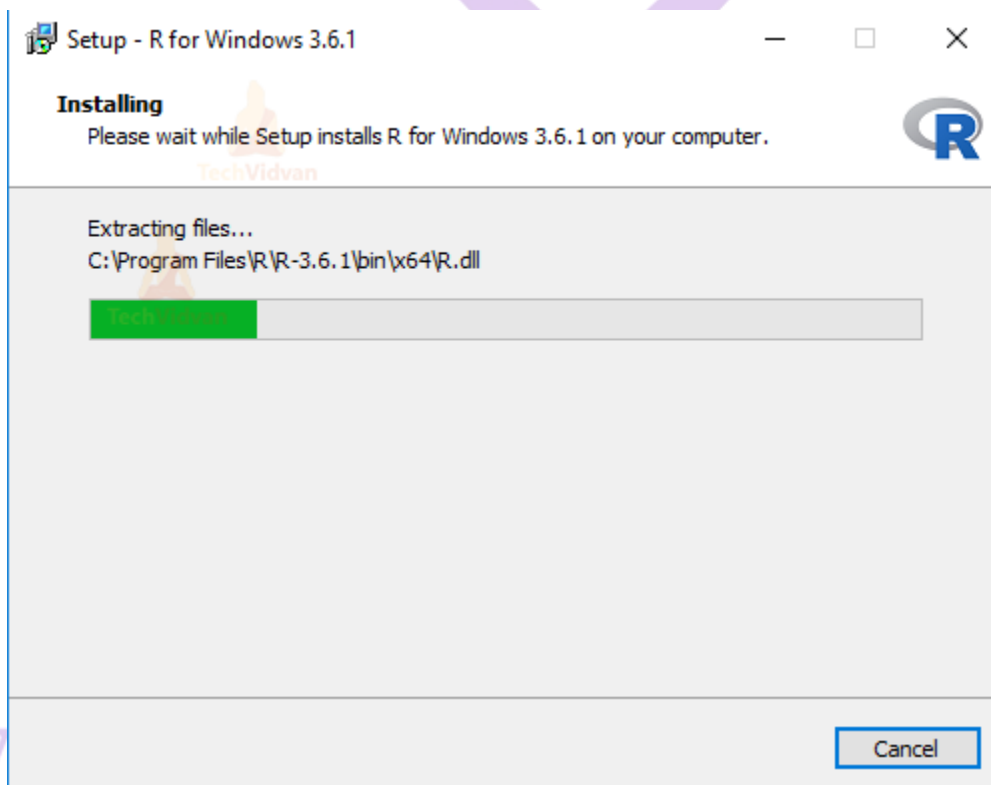
your roots to success...



## R-PROGRAMMING(DS3101PC/AM3101PC)

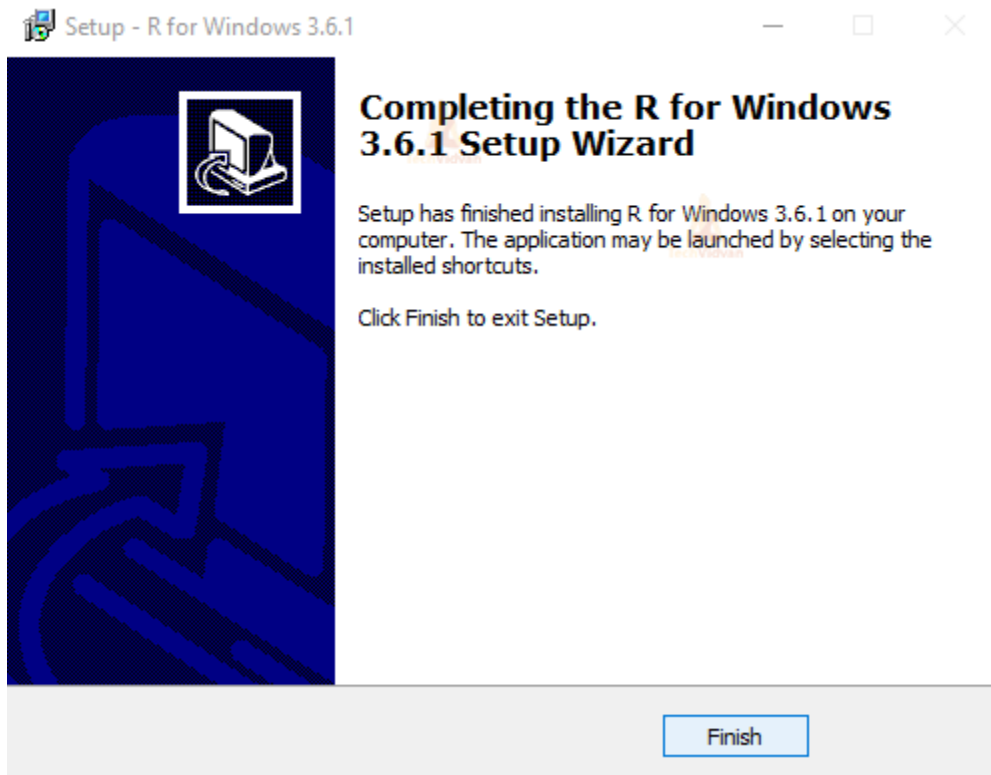


Wait for the installation process to complete.



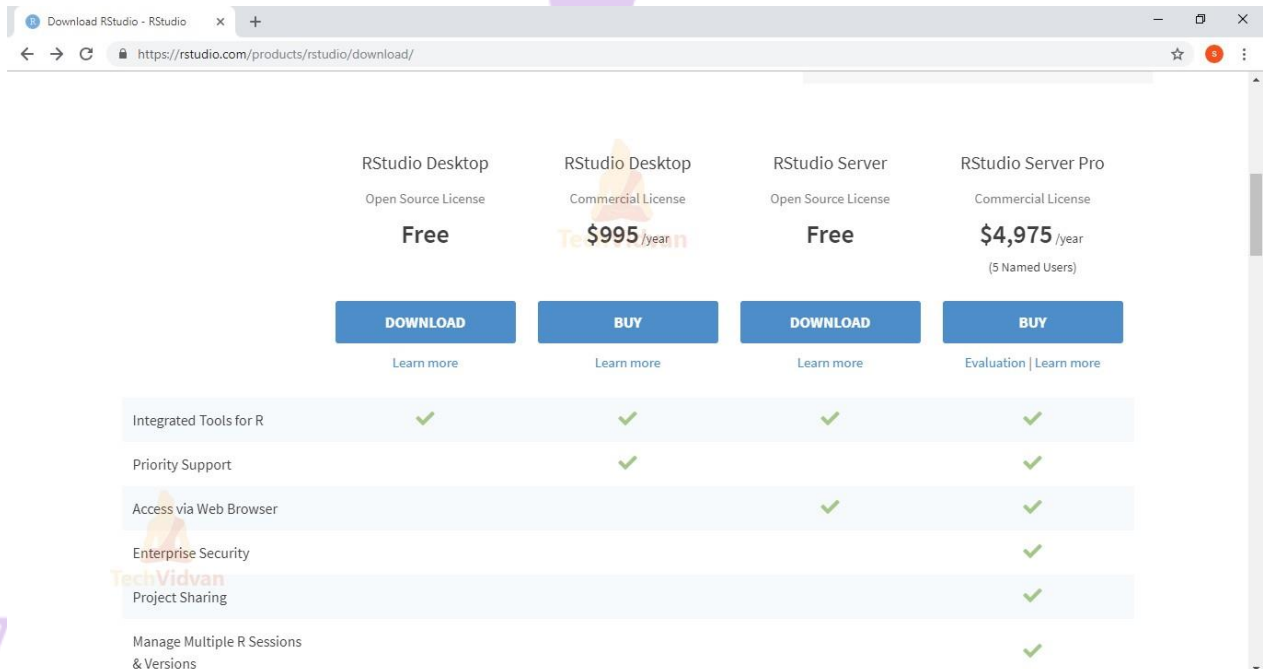
Click on Finish to complete the installation.

## R-PROGRAMMING(DS3101PC/AM3101PC)



### Install RStudio on Windows

Step – 1: With R-base installed, let's move on to installing RStudio. To begin, go to [download RStudio](https://rstudio.com/products/rstudio/download/) and click on the download button for RStudio desktop.



**Step – 2:** Click on the link for the windows version of RStudio and save the .exe file.

**Step – 3:** Run the .exe and follow the installation instructions.

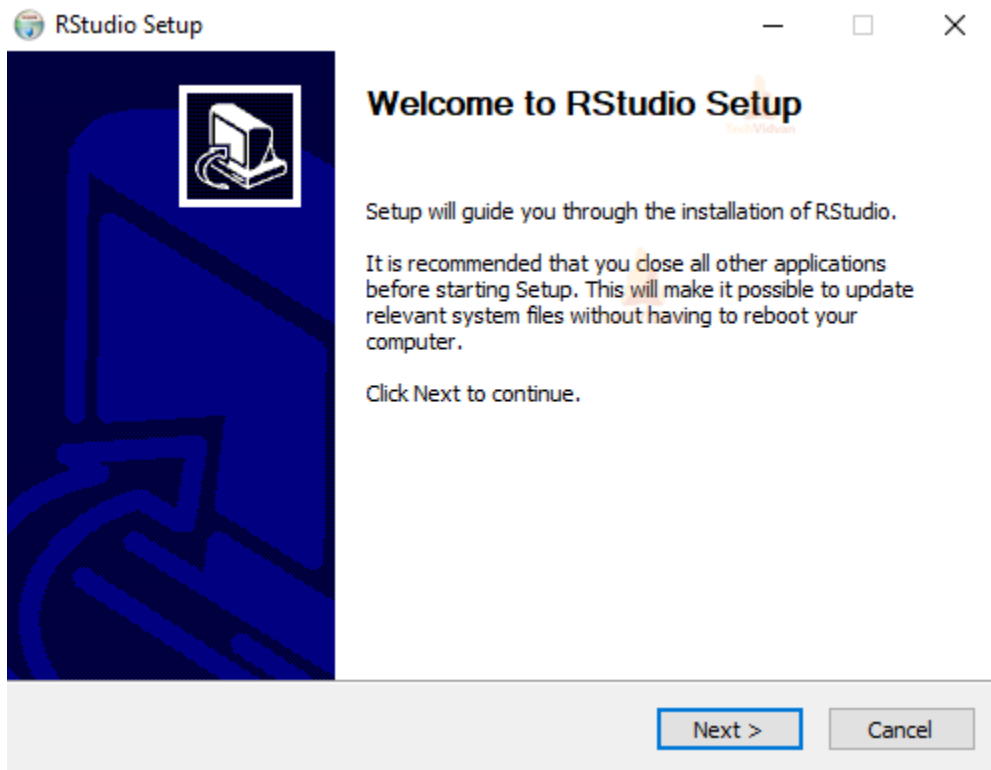
## R-PROGRAMMING(DS3101PC/AM3101PC)

Click Next on

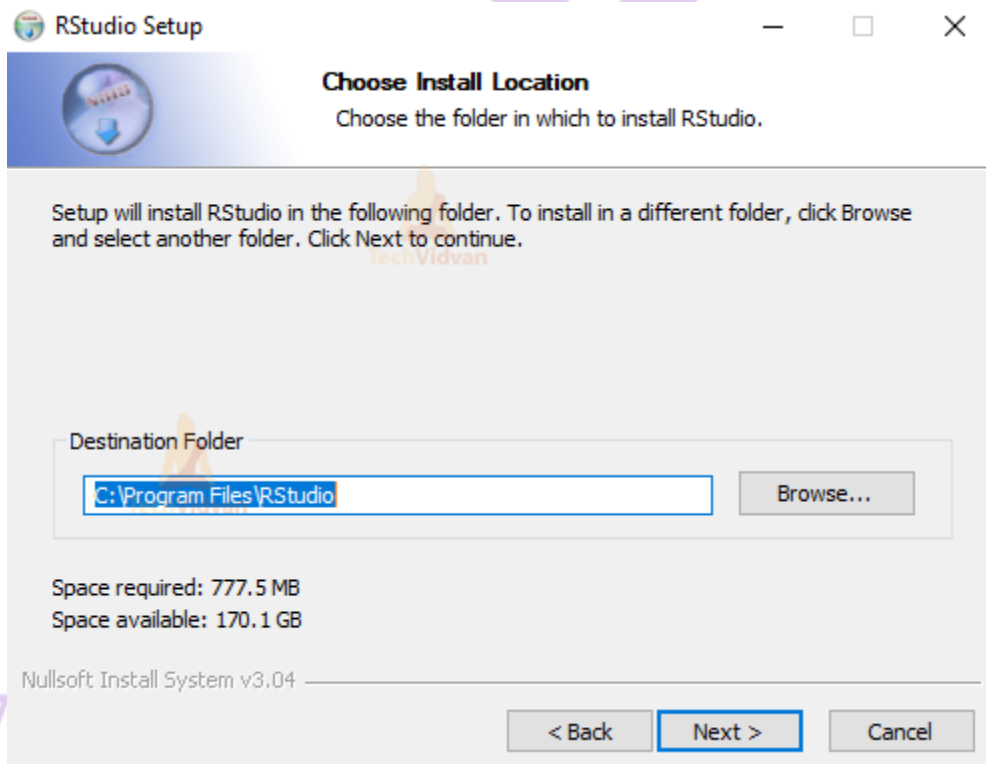
the

welcome

window.

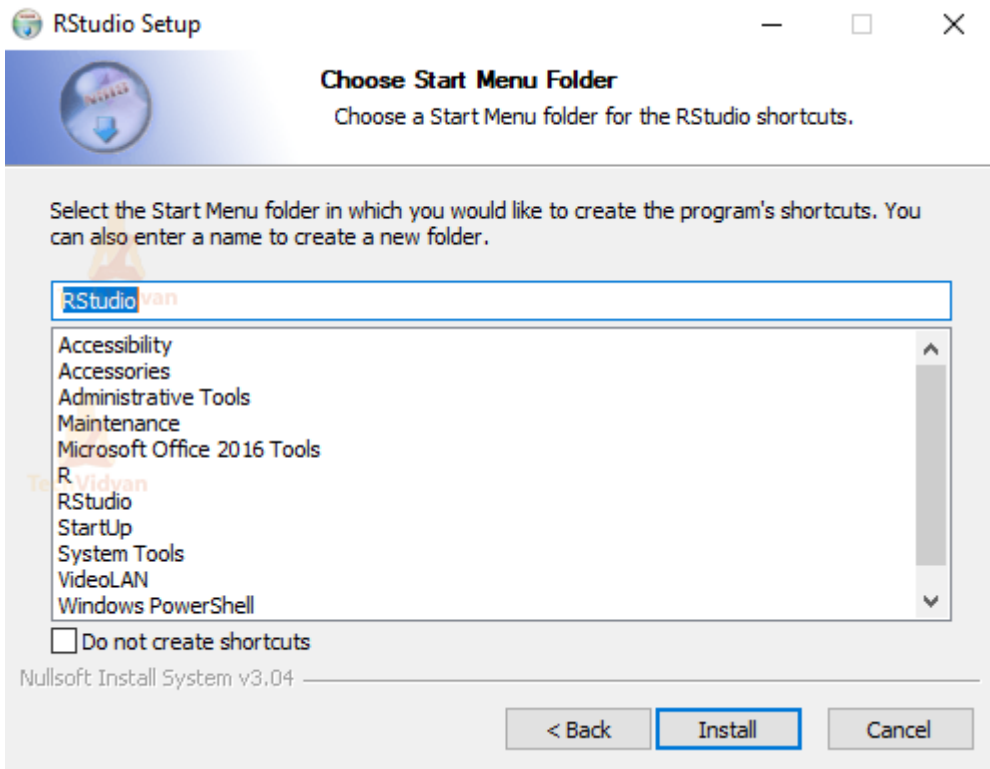


Enter/browse the path to the installation folder and click Next to proceed.

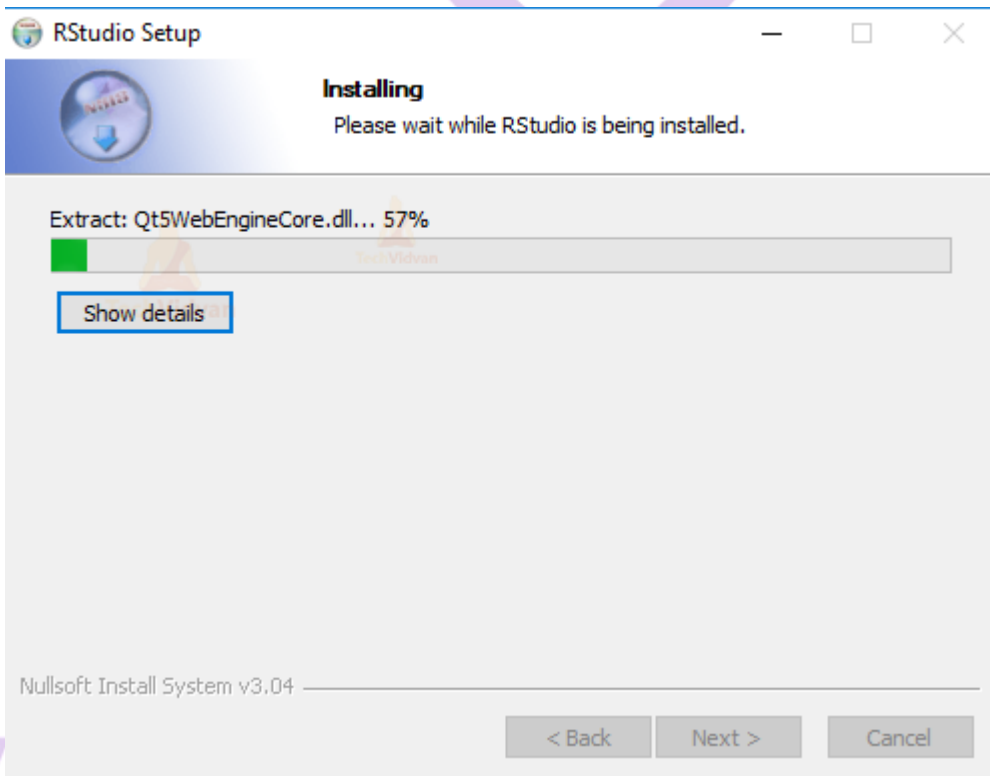


Select the folder for the start menu shortcut or click on do not create shortcuts and then click Next.

## R-PROGRAMMING(DS3101PC/AM3101PC)

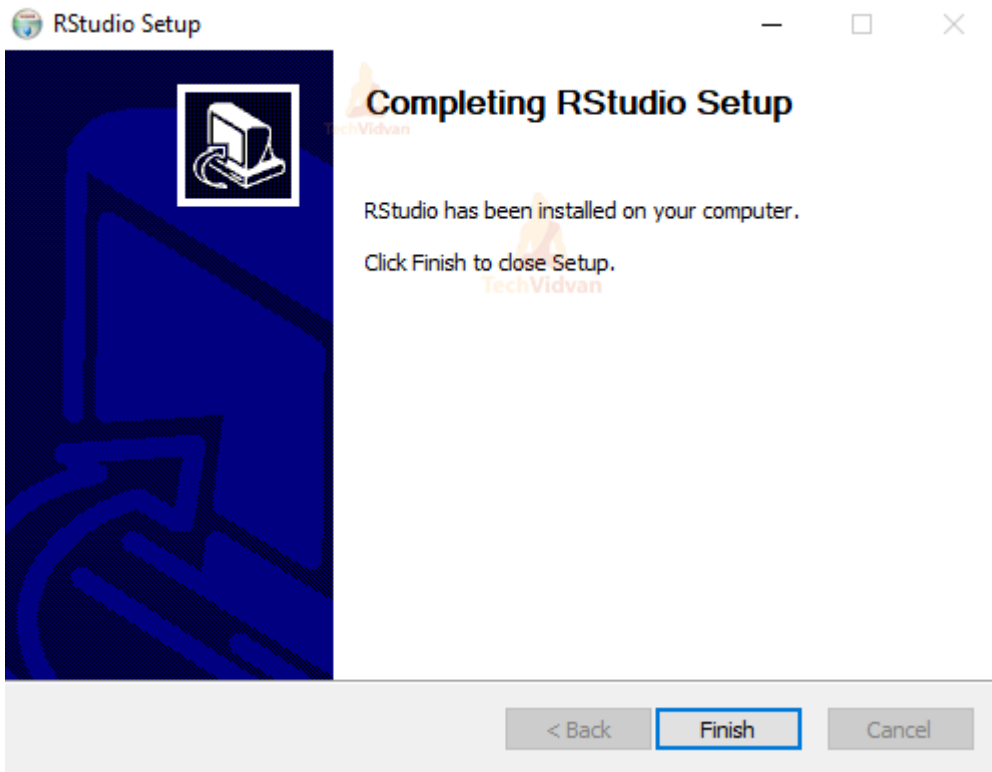


Wait for the installation process to complete.



Click Finish to end the installation.

## R-PROGRAMMING(DS3101PC/AM3101PC)



### Installing R and RStudio on Mac OS X

To install R and RStudio on Mac OS X, go through the following steps:

#### Install R on Mac

**Step – 1:** Go to [CRAN R Project Website](#).

**Step – 2:** Click on the Download for (Mac) OS X link.

**Step – 3:** Click on the link for the pkg file of the latest R version and save it.

**Step – 4:** Double click the downloaded file and follow installation instructions.

#### Install RStudio on Mac OS X

**Step – 1:** With the r-base installed, you need to install RStudio. To do that, go to [download RStudio](#) and click on the download button for the RStudio desktop.

**Step – 2:** Click on the link for the Mac OS X version of RStudio and save the .dmg file.

**Step – 3:** Double click the downloaded file and then drag-and-drop it into your applications folder.

Now with R and RStudio installed in your system, let's look at a few packages that might help you in learning as well as using R to its fullest potential!

#### Some useful Packages in R

## R-PROGRAMMING(DS3101PC/AM3101PC)

CRAN is full of packages for everything you will need while working with R, and it is still growing. Many useful functions of R come in these packages. To install a package, simply run the following command in RStudio:

```
>install.packages("<package name>")
```

Once installed, a package can be made available in the current R session using the command:

```
>library("<package name>")
```

While it may become confusing at times due to the sheer number of options available, here are a few packages that are popular for their reliability and usefulness:

**Tidyverse** – Tidyverse is a collection of packages that work in harmony with each other to clean, process, model, and visualize data. Tidyverse's core package contains packages like ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, and forcats.

**Installr** – **installr** allows you to update R and all its packages with just a single command.

**Rtweet** – Twitter is the prime target for extracting tweets and building models to understand and predict sentiment. The rtweet package allows you to scrap Tweets and perform sentiment analysis.

**MLR (Machine Learning in R)** – MLR is a package that lets you perform all kinds of machine learning tasks. MLR includes all the popular machine learning algorithms used for ML projects.

**Reticulate** – **Reticulate** lets you use Python alongside with R in the R environment. Not only that, but you can also use major Python libraries within R itself.

**R markdown** – R markdown lets you create documents in multiple formats like pdf, HTML, and MS Word documents while embedding R codes, results, and visualizations to produce informative and thorough reports.

**Shiny** – **Shiny** is an R package that lets you make interactive web-apps. Using shiny, you can embed the findings of your analysis into the web-apps. This enables the users to play with your data and the results for deeper understanding, resulting in improved communication of the results.

There are many more packages available on repositories like CRAN, Bioconductor, and GitHub that can be used to improve R's functions and facilities as well as to add new functions.

As you use and explore R some more, you will discover more of them and find new and exciting uses for them all.

### Summary

In this tutorial, we learned how to install R and RStudio in systems with Linux, Windows, and Mac OS X.

We also learned about some useful R packages that improve and enhance R's capabilities and prove to be highly useful.

## R-PROGRAMMING(DS3101PC/AM3101PC)

If you find any difficulty to install R and RStudio then ask us in the comment and our TechVidvans will be happy to help.

### Using R as a Calculator

One of the most basic ways to use R is as a calculator. We can enter many math functions directly into the R Console. We will start to look at the various ways in which we can use R.

### Arithmetic Operators

**R can handle most simple types of math operators.**

#### OperatorDescription

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

For example:

```
# Addition
```

```
5+4
```

```
## [1] 9
```

```
# Subtraction
```

```
124 - 26.82
```

```
## [1] 97.18
```

```
# Multiplication
```

```
5*4
```

```
## [1] 20
```

```
# Division
```

```
35/8
```

```
## [1] 4.375
```

### More Math functions.

We can also use many other math functions. All of these are included in base R without any extra packages.

## R-PROGRAMMING(DS3101PC/AM3101PC)

# Exponentials

```
3^(1/2)
```

```
## [1] 1.732051
```

# Exponential Function

```
exp(1.5)
```

```
## [1] 4.481689
```

# Log base e

```
log(4.481689)
```

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

# Log base 10

```
log10(1000)
```

```
## [1] 3
```

### On Your Own: Swirl Practice

In order to learn R you must do R. Follow the steps below in your RStudio console:

Run this command to pick the course:

```
swirl()
```

You will be prompted to choose a course. Type whatever number is in front of 01 Getting Started. This will then take you to a menu of lessons. For now we will just use lesson 2. Type 2 to choose R as Calculator then follow all the instructions until you are finished.

Once you are finished with the lesson come back to this course and continue.

### Logical Operators

Another important feature of R is the ability to use logic. This is not unique to R as all programming languages can do this, but it will be extremely useful when working with data.

OperatorDescription

< Less Than

> Greater Than

<= Less Than or Equal To

>= Greater Than or Equal To



## R-PROGRAMMING(DS3101PC/AM3101PC)

OperatorDescription

== Exactly Equal To

!= Not Equal To

!a Not a

a&b a AND b

We can then see an example of this:

```
a <- c(1:12)
```

If we wanted to know where  $a > 9$  or where  $a < 4$  we would expect to see the values: 1 2 3 10 11 12.

Having R do this

```
a
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12
```

```
a > 9
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
```

```
## [12] TRUE
```

We can see that what R gives is are Boolean values as to whether or not each element of a is greater than 9. Similarly:

```
a < 4
```

```
## [1] TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
```

```
## [12] FALSE
```

One simple thing we might try is to just combine these 2 values into a conditional

```
a > 9 | a < 4
```

```
## [1] TRUE TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE
```

```
## [12] TRUE
```

This again just gives us Booleans and not necessarily the values that we were hoping to see. We use this to index a in the following manner

```
a[a > 9 | a < 4]
```

```
## [1] 1 2 3 10 11 12
```

We will look at indexing with much further detail as we move through this course.

**On Your Own: Swirl Practice**

In order to learn R you must do R. Follow the steps below in your RStudio console:

Run this command to pick the course:

**swirl()**

You will be prompted to choose a course. Type whatever number is in front of 01 Getting Started. This will then take you to a menu of lessons. For now we will just use lesson 3. Type 3 to choose Logic then follow all the instructions until you are finished.

Once you are finished with the lesson come back to this course and continue.

**Further Operators**

There are other operators that we will use that are not necessarily mathematical in nature. Each one of them is crucial for the use of R.

Description	R Symbol	Example
Comment	#	# This is a comment
Assignment	<-	x <- 5
Assignment	->	5 -> x
Assignment	=	x = 5
Concatenation operator	c	c(1,2,4)
Modular	%%	25 %% 6
Sequence from a to b by hseq		seq(a,b,h)
Sequence Operator	:	0:3

Quick Check Practice

**script.R**

**R Console**

```
# Create a variable a, equal to 5
# Do this by typing: a <- 5
```

1

your roots to success...

2

3

4

5

6

```
# Create a variable a, equal to 5
```

```
# Do this by typing: a <- 5
```

```
# Square a
```

```
HintRun
```

**Powered by DataCamp**

### Math Functions in R

We also have access to a wide variety of mathematical functions that are already built into R.

Description	R Symbol
-------------	----------

Square Root	sqrt
-------------	------

floor(x)	floor
----------	-------

\ceil(x)	ceiling
----------	---------

Logarithm	log
-----------	-----

Exponential function, $e^x$	exp
-----------------------------	-----

Factorial, !	factorial
--------------	-----------

Quick Check Practice

[script.R](#)

[R Console](#)

```
# Create a variable b, equal to 4
```

1

2

3  
4  
5  
6  
7  
8  
9

```
# Create a variable b, equal to 4  
# What is the log of b?  
# what is the exponential of the log of b?
```

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

### Types of Operators

We have the following types of operators in R programming –

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Miscellaneous Operators

### Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

Operator	Description	Example
+	Adds two vectors	<a href="#">Live Demo</a>  v <-c(2,5.5,6) t <-c(8,3,4) print(v+t)  it produces the following result –  [1] 10.0 8.5 10.0

## R-PROGRAMMING(DS3101PC/AM3101PC)

–	Subtracts second vector from the first	<a href="#">Live Demo</a> <pre>v &lt;-c(2,5.5,6) t &lt;-c(8,3,4) print(v-t)</pre> it produces the following result – [1] -6.0 2.5 2.0
*	Multiplies both vectors	<a href="#">Live Demo</a> <pre>v &lt;-c(2,5.5,6) t &lt;-c(8,3,4) print(v*t)</pre> it produces the following result – [1] 16.0 16.5 24.0
/	Divide the first vector with the second	<a href="#">Live Demo</a> <pre>v &lt;-c(2,5.5,6) t &lt;-c(8,3,4) print(v/t)</pre> When we execute the above code, it produces the following result – [1] 0.250000 1.833333 1.500000
%%	Give the remainder of the first vector with the second	<a href="#">Live Demo</a> <pre>v &lt;-c(2,5.5,6) t &lt;-c(8,3,4) print(v%%t)</pre> it produces the following result – [1] 2.0 2.5 2.0

## R-PROGRAMMING(DS3101PC/AM3101PC)

%/%	The result of division of first vector with second (quotient)	<a href="#">Live Demo</a> <code>v &lt;-c(2,5.5,6)</code> <code>t &lt;-c(8,3,4)</code> <code>print(v%/%t)</code> it produces the following result – <code>[1] 0 1 1</code>
^	The first vector raised to the exponent of second vector	<a href="#">Live Demo</a> <code>v &lt;-c(2,5.5,6)</code> <code>t &lt;-c(8,3,4)</code> <code>print(v^t)</code> it produces the following result – <code>[1] 256.000 166.375 1296.000</code>

### Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	<a href="#">Live Demo</a> <code>v &lt;-c(2,5.5,6,9)</code> <code>t &lt;-c(8,2.5,14,9)</code> <code>print(v&gt;t)</code> it produces the following result – <code>[1] FALSE TRUE FALSE FALSE</code>
<	Checks if each element of the first vector is less than the corresponding element of the second	<a href="#">Live Demo</a> <code>v &lt;-c(2,5.5,6,9)</code>

## R-PROGRAMMING(DS3101PC/AM3101PC)

	vector.	<pre>t &lt;-c(8,2.5,14,9) print(v &lt; t) it produces the following result – [1] TRUE FALSE TRUE FALSE</pre>
==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	<pre><a href="#">Live Demo</a> v &lt;-c(2,5.5,6,9) t &lt;-c(8,2.5,14,9) print(v == t) it produces the following result – [1] FALSE FALSE FALSE TRUE</pre>
<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	<pre><a href="#">Live Demo</a> v &lt;-c(2,5.5,6,9) t &lt;-c(8,2.5,14,9) print(v&lt;=t) it produces the following result – [1] TRUE FALSE TRUE TRUE</pre>
>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	<pre><a href="#">Live Demo</a> v &lt;- c(2,5.5,6,9) t &lt;-c(8,2.5,14,9) print(v&gt;=t) it produces the following result – [1] FALSE TRUE FALSE TRUE</pre>
!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	<pre><a href="#">Live Demo</a> v &lt;-c(2,5.5,6,9) t &lt;-c(8,2.5,14,9)</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

		<pre>print(v!=t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE TRUE TRUE FALSE</pre>
--	--	--

### Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
&	<p>It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(3,1,TRUE,2+3i)</pre> <pre>t &lt;- c(4,1,FALSE,2+3i)</pre> <pre>print(v&amp;t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE TRUE FALSE TRUE</pre>
	<p>It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(3,0,TRUE,2+2i)</pre> <pre>t &lt;- c(4,0,FALSE,2+3i)</pre> <pre>print(v t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE FALSE TRUE TRUE</pre>
!	<p>It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(3,0,TRUE,2+2i)</pre> <pre>print(!v)</pre> <p>it produces the following result –</p>



## R-PROGRAMMING(DS3101PC/AM3101PC)

		[1] FALSE TRUE FALSE FALSE
--	--	----------------------------

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example
&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	<a href="#">Live Demo</a>  v <- c(3,0,TRUE,2+2i) t <- c(1,3,TRUE,2+3i) print(v&&t)  it produces the following result –  [1] TRUE
	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	<a href="#">Live Demo</a>  v <- c(0,0,TRUE,2+2i) t <- c(0,3,TRUE,2+3i) print(v  t)  it produces the following result –  [1] FALSE

### Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example
<- or = or <<-	Called Left Assignment	<a href="#">Live Demo</a>  v1 <- c(3,1,TRUE,2+3i) v2 <<- c(3,1,TRUE,2+3i) v3 = c(3,1,TRUE,2+3i)  print(v1)  print(v2)

## R-PROGRAMMING(DS3101PC/AM3101PC)

		<pre>print(v3)</pre> <p>it produces the following result –</p> <pre>[1] 3+0i 1+0i 1+0i 2+3i</pre> <pre>[1] 3+0i 1+0i 1+0i 2+3i</pre> <pre>[1] 3+0i 1+0i 1+0i 2+3i</pre>
<pre>-&gt;</pre> <p>or</p> <pre>-&gt;&gt;</pre>	<p>Called Right Assignment</p>	<p><a href="#">Live Demo</a></p> <pre>c(3,1,TRUE,2+3i)-&gt; v1</pre> <pre>c(3,1,TRUE,2+3i)-&gt;&gt; v2</pre> <pre>print(v1)</pre> <pre>print(v2)</pre> <p>it produces the following result –</p> <pre>[1] 3+0i 1+0i 1+0i 2+3i</pre> <pre>[1] 3+0i 1+0i 1+0i 2+3i</pre>

### Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	<p><a href="#">Live Demo</a></p> <pre>v &lt;-2:8</pre> <pre>print(v)</pre> <p>it produces the following result –</p> <pre>[1] 2 3 4 5 6 7 8</pre>
%in%	This operator is used to identify if an element	<a href="#">Live Demo</a>

## R-PROGRAMMING(DS3101PC/AM3101PC)

	belongs to a vector.	<pre>v1 &lt;-8 v2 &lt;-12 t &lt;-1:10 print(v1 %in% t) print(v2 %in% t) it produces the following result – [1] TRUE [1] FALSE</pre>
%**%	<p>This operator is used to multiply a matrix with its transpose.</p>	<pre>M =matrix( c(2,6,5,1,10,4),nrow=2,ncol =3,byrow = TRUE) t = M %**% t(M) print(t) it produces the following result – [,1] [,2] [1,] 65 82 [2,] 82 117</pre>

### Complex

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

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

```
> z # print the value of z
```

```
[1] 1+2i
```

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

```
[1] "complex"
```

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

```
> sqrt(-1) # square root of -1
```

```
[1] NaN
```

Warning message:

```
In sqrt(-1) : NaNs produced
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

Instead, we have to use the complex value  $-1 + 0i$ .

```
> sqrt(-1+0i) # square root of -1+0i  
[1] 0+1i
```

An alternative is to coerce  $-1$  into a complex value.

```
> sqrt(as.complex(-1))  
[1] 0+1i
```

< IntegerupLogical >

Tags:

R Introduction

as.complex

class

sqrt

How to Round Numbers in R (5 Examples)

**You can use the following functions to round numbers in R:**

`round(x, digits = 0)`: Rounds values to specified number of decimal places.

`signif(x, digits = 6)`: Rounds values to specified number of significant digits.

`ceiling(x)`: Rounds values up to nearest integer.

`floor(x)`: Rounds values down to nearest integer.

`trunc(x)`: Truncates (cuts off) decimal places from values.

**The following examples show how to use each of these functions in practice.**

Example 1: `round()` Function in R

The following code shows how to use the `round()` function in R:

```
#define vector of data  
  
data <- c(.3, 1.03, 2.67, 5, 8.91)  
  
#round values to 1 decimal place
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
round(data, digits = 1)
```

```
[1] 0.3 1.0 2.7 5.0 8.9
```

Example 2: signif() Function in R

**The following code shows how to use the signif() function to round values to a specific number of significant digits in R:**

```
#define vector of data
```

```
data <- c(.3, 1.03, 2.67, 5, 8.91)
```

```
#round values to 3 significant digits
```

```
signif(data, digits = 3)
```

```
[1] 0.30 1.03 2.67 5.00 8.91
```

Example 3: ceiling() Function in R

**The following code shows how to use the ceiling() function to round values up to the nearest integer:**

```
#define vector of data
```

```
data <- c(.3, 1.03, 2.67, 5, 8.91)
```

```
#round values up to nearest integer
```

```
ceiling(data)
```

```
[1] 1 2 3 5 9
```

**Example 4: floor() Function in R**

The following code shows how to use the floor() function to round values down to the nearest integer:

```
#define vector of data
```

```
data <- c(.3, 1.03, 2.67, 5, 8.91)
```

```
#round values down to nearest integer
```

```
floor(data)
```

```
[1] 0 1 2 5 8
```

**Example 5: trunc() Function in R**

## R-PROGRAMMING(DS3101PC/AM3101PC)

The following code shows how to use the trunc() function to truncate (cut off) decimal places from values:

```
#define vector of data  
data <- c(.3, 1.03, 2.67, 5, 8.91)  
  
#truncate decimal places from values  
trunc(data)  
[1] 0 1 2 5 8
```

"" and "" for the remainder and the quotient

### Ask Question

Asked 10 years, 3 months ago

Modified 4 years, 8 months ago

Viewed 135k times

### **Report this ad**

73

I am wondering how and why the operator %% and %/% are for the remainder and the quotient.

Is there any reason or history that R developer had given them the meaning they have?

```
>0%/% 10
```

```
[1]0
```

```
>30%% 10
```

```
[1]0
```

```
>35%/% 10
```

```
[1]3
```

```
>35%% 10
```

```
[1]5
```

In R, you can assign your own operators using %[characters]%. A trivial example:

```
'%p%' <- function(x, y){x^2 + y}
```

```
2 %p% 3 # result: 7
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

While I agree with BlueTrin that %% is pretty standard, I have a suspicion %/% may have something to do with the sort of operator definitions I showed above - perhaps it was easier to implement, and makes sense: %/% means do a special sort of division (integer division)

### Variable names and assignment

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name, var.name	valid	Can start with a dot(.) but the dot(.) should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

### Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using print() or cat() function. The cat() function combines multiple items into a continuous print output.

# Assignment using equal operator.

```
var.1=c(0,1,2,3)
```

# Assignment using leftward operator.

```
var.2<- c("learn","R")
```

# Assignment using rightward operator.

```
c(TRUE,1)->var.3
```

```
print(var.1)
```

```
cat ("var.1 is ",var.1,"\n")
```

```
cat ("var.2 is ",var.2,"\n")
```

```
cat ("var.3 is ",var.3,"\n")
```

When we execute the above code, it produces the following result –

```
[1] 0 1 2 3
```

```
var.1 is 0 1 2 3
```

```
var.2 is learn R
```

```
var.3 is 1 1
```

Note – The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

### **Data Type of a Variable**

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x<-"Hello"
```

```
cat("The class of var_x is ",class(var_x),"\n")
```

```
var_x<-34.5
```

```
cat(" Now the class of var_x is ",class(var_x),"\n")
```

```
var_x<-27L
```

```
cat(" Next the class of var_x becomes ",class(var_x),"\n")
```

When we execute the above code, it produces the following result –



## R-PROGRAMMING(DS3101PC/AM3101PC)

The class of var\_xis character

Now the class of var\_xis numeric

Next the class of var\_xbecomes integer

### Finding Variables

To know all the variables currently available in the workspace we use the ls() function. Also the ls() function can use patterns to match the variable names.

```
print(ls())
```

When we execute the above code, it produces the following result –

```
[1] "my var" "my_new_var" "my_var" "var.1"  
[5] "var.2" "var.3" "var.name" "var_name2."  
[9] "var_x" "varname"
```

Note – It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names

```
# List the variables starting with the pattern "var".
```

```
print(ls(pattern = "var"))
```

When we execute the above code, it produces the following result –

```
[1] "my var" "my_new_var" "my_var" "var.1"  
[5] "var.2" "var.3" "var.name" "var_name2."  
[9] "var_x" "varname"
```

The variables starting with dot(.) are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

```
print(ls(all.name = TRUE))
```

When we execute the above code, it produces the following result –

```
[1] ".cars" ".Random.seed" ".var_name" ".varname" ".varname2"  
[6] "my var" "my_new_var" "my_var" "var.1" "var.2"  
[11] "var.3" "var.name" "var_name2." "var_x"
```

### Deleting Variables

Variables can be deleted by using the rm() function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

```
rm(var.3)
```

```
print(var.3)
```

When we execute the above code, it produces the following result –

```
[1] "var.3"
```

```
Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the rm() and ls() function together.

```
rm(list = ls())
```

```
print(ls())
```

When we execute the above code, it produces the following result –

```
character(0)
```

Integer

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

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

```
> y      # print the value of y
```

```
[1] 3
```

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

```
[1] "integer"
```

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

```
[1] TRUE
```

We can also declare an integer by appending an L suffix.

```
> y = 3L
```

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

```
[1] TRUE
```

Incidentally, we can coerce a numeric value into an integer with the as.integer function.

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

```
[1] 3
```

And we can parse a string for decimal values in much the same way.

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

```
[1] 5
```

On the other hand, it is erroneous trying to parse a non-decimal string.

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
> as.integer("Joe") # coerce an non-decimal string
[1] NA
Warning message:
NAs introduced by coercion
```

Often, it is useful to perform arithmetic on logical values. Just like the C language, TRUE has the value 1, while FALSE has value 0.

```
> as.integer(TRUE) # the numeric value of TRUE
[1] 1
> as.integer(FALSE) # the numeric value of FALSE
[1] 0
```

## UNIT-II

### Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some “logic” into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are

**if and else: testing a condition and acting on it**

**for: execute a loop a fixed number of times**

**while: execute a loop while a condition is true**

**repeat: execute an infinite loop (must break out of it to stop)**

**break: break the execution of a loop**

**next: skip an iteration of a loop**

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions. However, these constructs do not have to be used in functions and it's a good idea to become familiar with them before we delve into functions.

if-else

The **if-else combination** is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

For starters, you can just use the if statement.

```
if(<condition>) {
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
## do something  
}
```

## Continue with rest of code

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an else clause.

```
if(<condition>) {  
## do something  
} else {  
## do something else  
}
```

You can have a series of tests by following the initial if with any number of else ifs.

```
if(<condition1>) {  
## do something  
} elseif(<condition2>) {  
## do something different  
} else {  
## do something different  
}
```

Here is an example of a valid if/else structure.

```
## Generate a uniform random number  
x <-runif(1, 0, 10)  
if(x > 3) {  
  y <- 10  
} else {  
  y <- 0  
}
```

The value of y is set depending on whether  $x > 3$  or not. This expression can also be written a different, but equivalent, way in R.

```
y <-if(x >3) {  
  10  
} else {  
  0  
}
```

Neither way of writing this expression is more correct than the other. Which one you use will depend on your preference and perhaps those of the team you may be working with.

Of course, the else clause is not necessary. You could have a series of if clauses that always get executed if their respective conditions are true.

```
if(<condition1>) {  
}  
if(<condition2>) {  
}
```

### for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a for loop wasn't sufficient.

In R, for loops take an iterator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
>for(iin1:10) {  
+print(i)  
+ }
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

```
[1] 6
```

```
[1] 7
```

```
[1] 8
```

```
[1] 9
```

```
[1] 10
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, executes the code within the curly braces, and then the loop exits.

The following three loops all have the same behavior.

```
> x <-c("a", "b", "c", "d")
```

```
>
```

```
>for(iin1:4) {
```

```
+## Print out each element of 'x'
```

```
+print(x[i])
```

```
+ }
```

```
[1] "a"
```

```
[1] "b"
```

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

```
[1] "d"
```

The `seq_along()` function is commonly used in conjunction with `for` loops in order to generate an integer sequence based on the length of an object (in this case, the object `x`).

```
>## Generate a sequence based on length of 'x'
```

```
>for(iinseq_along(x)) {
```

```
+print(x[i])
```

```
+ }
```

```
[1] "a"
```

```
[1] "b"
```

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

```
[1] "d"
```

It is not necessary to use an index-type variable.

```
>for(letter in x) {  
+print(letter)  
+ }
```

```
[1] "a"  
[1] "b"  
[1] "c"  
[1] "d"
```

For one line loops, the curly braces are not strictly necessary.

```
>for(iin1:4) print(x[i])  
[1] "a"  
[1] "b"  
[1] "c"  
[1] "d"
```

However, I like to use curly braces even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won't be burned because you forgot to add curly braces (and you will be burned by this).

### **Nested for loops**

for loops can be nested inside of each other.

```
x <-matrix(1:6, 2, 3)  
for(iinseq_len(nrow(x))) {  
  for(j inseq_len(ncol(x))) {  
    print(x[i, j])  
  }  
}
```

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

### **while Loops**

## R-PROGRAMMING(DS3101PC/AM3101PC)

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <-0
```

```
>while(count <10) {
```

```
+print(count)
```

```
+   count <- count +1
```

```
+ }
```

```
[1] 0
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

```
[1] 6
```

```
[1] 7
```

```
[1] 8
```

```
[1] 9
```

While loops can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in the test.

```
> z <-5
```

```
>set.seed(1)
```

```
>
```

```
>while(z >=3&& z <=10) {
```

```
+   coin <-rbinom(1, 1, 0.5)
```

```
+ 
```

```
+if(coin ==1) { ## random walk
```

```
+   z <- z +1
```



```
+ }else {  
+     z <- z -1  
+ }  
+ }  
>print(z)  
[1] 2
```

Conditions are always evaluated from left to right. For example, in the above code, if z were less than 3, the second test would not have been evaluated.

### repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.

One possible paradigm might be in an iterative algorithm where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get "close enough" to the solution.

```
x0 <-1  
tol<-1e-8  
  
repeat {  
    x1 <-computeEstimate()  
  
    if(abs(x1 - x0) <tol) { ## Close enough?  
        break  
    } else {  
        x0 <- x1  
    }  
}
```

your roots to success...

## R-PROGRAMMING(DS3101PC/AM3101PC)

Note that the above code will not run if the `computeEstimate()` function is not defined (I just made it up for the purposes of this demonstration).

The loop above is a bit dangerous because there's no guarantee it will stop. You could get in a situation where the values of `x0` and `x1` oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a `for` loop and then report whether convergence was achieved or not.

### next, break

`next` is used to skip an iteration of a loop.

```
for(iin1:100) {  
  if(i<=20) {  
    ## Skip the first 20 iterations  
    next  
  }  
  ## Do something here  
}
```

`break` is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(iin1:100) {  
  print(i)  
  
  if(i>20) {  
    ## Stop loop after 20 iterations  
    break  
  }  
}
```

### Summary

Control structures like `if`, `while`, and `for` allow you to control the flow of an R program

Infinite loops should generally be avoided, even if (you believe) they are theoretically correct.

Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the “apply” functions are more useful.

### R Functions

## R-PROGRAMMING(DS3101PC/AM3101PC)

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

### Creating a Function

To create a function, use the function() keyword:

Example

```
my_function<- function() { # create a function with the name my_function
  print("Hello World!")
}
```

### Call a Function

To call a function, use the function name followed by parenthesis, like my\_function():

Example

```
my_function<- function() {
  print("Hello World!")
}
my_function() # call the function named my_function
```

### OUTPUT:

```
[1] "Hello World!"
```

### Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

### Example

```
my_function<- function(fname) {
  paste(fname, "Griffin")
}
```

```
my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

### OUTPUT:

```
[1] "Peter Griffin"  
[1] "Lois Griffin"  
[1] "Stewie Griffin"
```

### **Parameters or Arguments?**

The terms "parameter" and "argument" can be used for the same thing: information that are passed into a function.

### **From a function's perspective:**

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

### **Number of Arguments**

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less:

### **Example**

This function expects 2 arguments, and gets 2 arguments:

```
my_function<- function(fname, lname) {  
  paste(fname, lname)  
}  
my_function("Peter", "Griffin")
```

### **OUTPUT:**

```
1] "Peter Griffin"
```

If you try to call the function with 1 or 3 arguments, you will get an error:

### **Example**

This function expects 2 arguments, and gets 1 argument:

```
my_function<- function(fname, lname) {  
  paste(fname, lname)  
}
```

```
my_function("Peter")
```

### **OUTPUT:**

```
Error in paste(fname, lname) :  
  argument "lname" is missing, with no default  
Calls: my_function -> print -> paste  
Execution halted
```

### Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without an argument, it uses the default value:

#### Example

```
my_function<- function(country = "Norway") {  
  paste("I am from", country)  
}  
  
my_function("Sweden")  
my_function("India")  
my_function() # will get the default value, which is Norway  
my_function("USA")
```

#### OUTPUT:

```
[1] "I am from Sweden"  
[1] "I am from India"  
[1] "I am from Norway"  
[1] "I am from USA"
```

#### Return Values

To let a function return a result, use the return() function:

#### Example

```
my_function<- function(x) {  
  return (5 * x)  
}  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

#### OUTPUT:

```
[1] 15  
[1] 25  
[1] 45
```

The output of the code above will be:

```
[1] 15  
[1] 25  
[1] 45
```

## Nested Functions

There are two ways to create a nested function:

Call a function within another function.

Write a function within a function.

### Example

Call a function within another function:

```
Nested_function<- function(x, y) {  
  a <- x + y  
  return(a)  
}
```

```
Nested_function(Nested_function(2,2), Nested_function(3,3))
```

### OUTPUT:

```
[1] 10
```

### Example Explained

The function tells x to add y.

The first input Nested\_function(2,2) is "x" of the main function.

The second input Nested\_function(3,3) is "y" of the main function.

The output is therefore  $(2+2) + (3+3) = 10$ .

### Example

Write a function within a function:

```
Outer_func<- function(x) {  
  Inner_func<- function(y) {  
    a <- x + y  
    return(a)  
  }  
  return (Inner_func)
```

```
}  
output <- Outer_func(3) # To call the Outer_func  
output(5)
```

### OUTPUT:

```
[1] 8
```

### Example Explained

## R-PROGRAMMING(DS3101PC/AM3101PC)

- You cannot directly call the function because the Inner\_func has been defined (nested) inside the Outer\_func.
- We need to call Outer\_func first in order to call Inner\_func as a second step.
- We need to create a new variable called output and give it a value, which is 3 here.
- We then print the output with the desired value of "y", which in this case is 5.
- The output is therefore 8 (3 + 5).

### Recursion

R also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly, recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

### Example

```
tri_recursion<- function(k) {  
  if (k > 0) {  
    result <- k + tri_recursion(k - 1)  
    print(result)  
  } else {  
    result = 0  
    return(result)  
  }  
}  
tri_recursion(6)
```

OUTPUT:

```
[1] 1  
[1] 3  
[1] 6  
[1] 10  
[1] 15  
[1] 21
```

**Variable:**

## R-PROGRAMMING(DS3101PC/AM3101PC)

A variable is a memory allocated for the storage of specific data and the name associated with the variable is used to work around this reserved block. The name given to a variable is known as its variable name. Usually a single variable stores only the data belonging to a certain data type. The name is so given to them because when the program executes there is subject to change hence it **varies from time to time**.

### Variables in R

R Programming Language is a dynamically typed language, i.e. the R Language Variables are not declared with a data type rather they take the data type of the R-object assigned to them. This feature is also shown in languages like Python and PHP.

- Declaring and Initializing Variables in R Language
- R supports three ways of variable assignment:
- Using equal operator- data is copied from right to left.
- Using leftward operator- data is copied from right to left.
- Using rightward operator- data is copied from left to right.

### R Variables Syntax:

#using equal to operator

```
variable_name = value
```

#using leftward operator

```
variable_name<- value
```

#using rightward operator

```
value ->variable_name
```

### Example: Creating Variables in R

R

```
# R program to illustrate
```

```
# Initialization of variables
```

```
# using equal to operator
```

```
var1 = "hello"
```

```
print(var1)
```

```
# using leftward operator
```

```
var2 <- "hello"
```



```
print(var2)
# using rightward operator
"hello"-> var3
print(var3)
```

**Output:**

```
[1] "hello"
[1] "hello"
[1] "hello"
```

**Important Methods for Variables**

R provides some useful methods to perform operations on variables. These methods are used to determine the data type of the variable, finding a variable, deleting a variable, etc. Following are some of the methods used to work on variables:

**class() function**

This built-in function is used to determine the data type of the variable provided to it. The variable to be checked is passed to this as an argument and it prints the data type in return.

Syntax:

```
class(variable)
```

Example:

```
R
var1 = "hello"
print(class(var1))
```

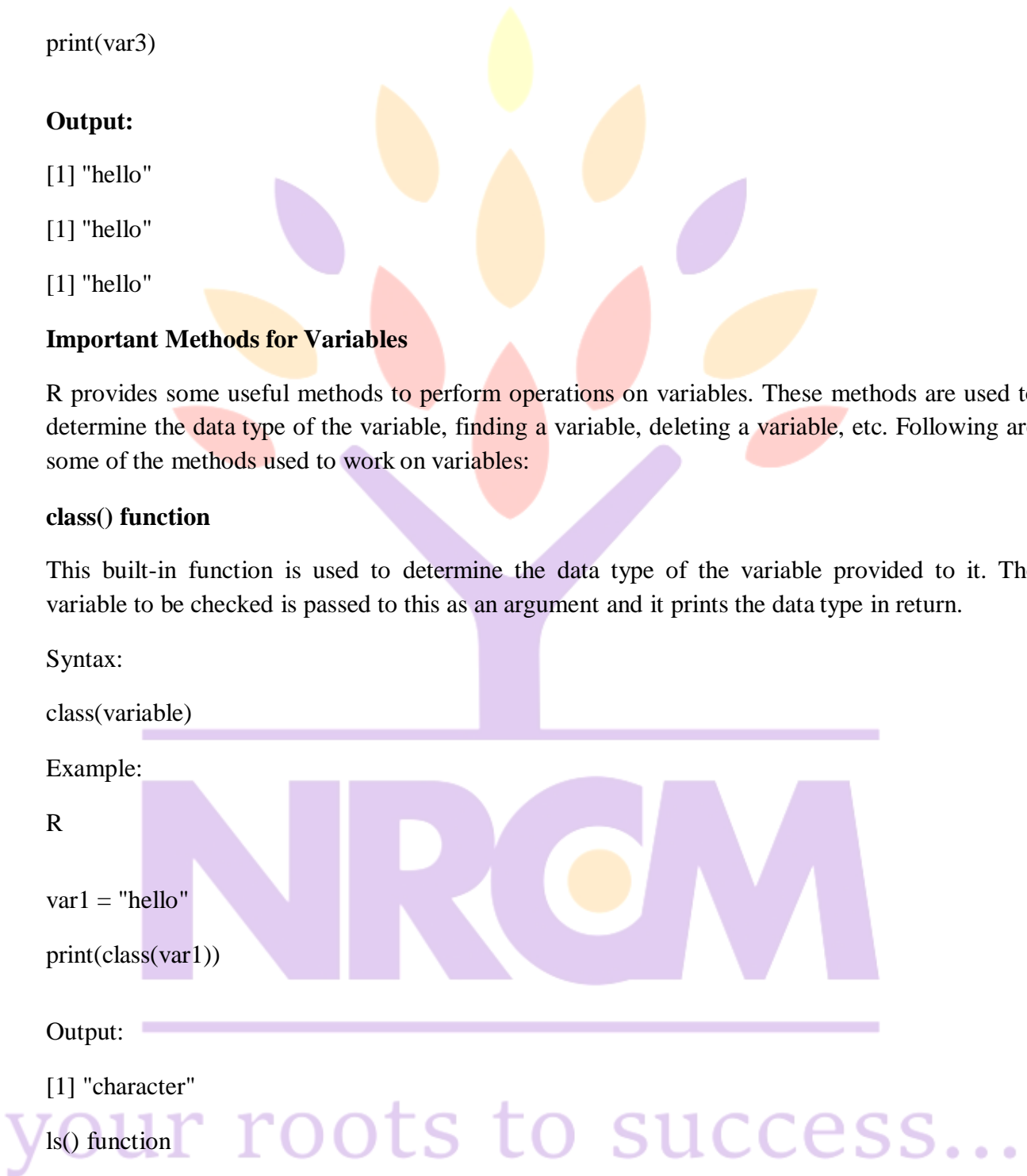
Output:

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

**ls() function**

This built-in function is used to know all the present variables in the workspace. This is generally helpful when dealing with a large number of variables at once and helps prevent overwriting any of them.

Syntax:



ls()

**Example:**

```
# using equal to operator
```

```
var1 = "hello"
```

```
# using leftward operator
```

```
var2 < - "hello"
```

```
# using rightward operator
```

```
"hello"-> var3
```

```
print(ls())
```

**Output:**

```
[1] "var1" "var2" "var3"
```

**rm() function**

This is again a built-in function used to delete an unwanted variable within your workspace. This helps clear the memory space allocated to certain variables that are not in use thereby creating more space for others. The name of the variable to be deleted is passed as an argument to it.

Syntax:

```
rm(variable)
```

**Example:**

```
# using equal to operator
```

```
var1 = "hello"
```

```
# using leftward operator
```

```
var2 < - "hello"
```

```
# using rightward operator
```

```
"hello"-> var3
```

```
# Removing variable
```

```
rm(var3)
```

```
print(var3)
```

**Output:**

Error in print(var3) : object 'var3' not found

**Scope of Variables in R programming**

The location where we can find a variable and also access it if required is called the scope of a variable. There are mainly two types of variable scopes:

**Global Variables:**

Global variables are those variables that exist throughout the execution of a program. It can be changed and accessed from any part of the program.

As the name suggests, Global Variables can be accessed from any part of the program.

They are available throughout the lifetime of a program.

They are declared anywhere in the program outside all of the functions or blocks.

Declaring global variables: Global variables are usually declared outside of all of the functions and blocks. They can be accessed from any portion of the program.

```
# R program to illustrate
```

```
# usage of global variables
```

```
# global variable
```

```
global = 5
```

```
# global variable accessed from
```

```
# within a function
```

```
display = function(){
```

```
  print(global)
```

```
}
```

```
display()
```

```
# changing value of global variable
```

```
global = 10
```

```
display()
```

### **Output:**

```
[1] 5
```

```
[1] 10
```

In the above code, the variable 'global' is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

### **Local Variables:**

Local variables are those variables that exist only within a certain part of a program like a function and are released when the function call ends. Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

### **Declaring local variables:**

**Local variables are declared inside a block.**

### **Scope of Variable in R**

In R, variables are the containers for storing data values. They are reference, or pointers, to an object in memory which means that whenever a variable is assigned to an instance, it gets mapped to that instance. A variable in R can store a vector, a group of vectors or a combination of many R objects.

### **Example:**

```
# R program to demonstrate
```

```
# variable assignment
```

```
# Assignment using equal operator
```

```
var1 =c(0, 1, 2, 3)
```

```
print(var1)
```

```
# Assignment using leftward operator
```

```
var2 <-c("Python", "R")
```

```
print(var2)
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# A Vector Assignment
a =c(1, 2, 3, 4)
print(a)
b =c("Debi", "Sandeep", "Subham", "Shiba")
print(b)
# A group of vectors Assignment using list
c =list(a, b)
print(c)
```

### Output:

```
[1] 0 1 2 3
[1] "Python" "R"
[1] 1 2 3 4
[1] "Debi" "Sandeep" "Subham" "Shiba"
[[1]]
[1] 1 2 3 4
[[2]]
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

### Naming convention for Variables

The variable name in R has to be Alphanumeric characters with an exception of underscore('\_') and period('.'), the special characters which can be used in the variable names.

The variable name has to be started always with an alphabet.

Other special characters like('!', '@', '#', '\$') are not allowed in the variable names.

### Example:

```
# R program to demonstrate
# rules for naming the variables
# Correct naming
```

```
b2 =7  
  
# Correct naming  
Amiya_Profession="Student"  
  
# Correct naming  
Amiya.Profession="Student"  
  
# Wrong naming  
2b=7  
  
# Wrong naming  
Amiya@Profession="Student"
```

Above code when executed will generate an error because of the wrong naming of variables.

Error: unexpected symbol in "2b"

Execution halted

### Scope of a variable

The location where we can find a variable and also access it if required is called the scope of a variable. There are mainly two types of variable scopes:

**Global Variables:** Global variables are those variables that exist throughout the execution of a program. It can be changed and accessed from any part of the program.

**Local Variables:** Local variables are those variables that exist only within a certain part of a program like a function and are released when the function call ends.

```
# global variable  
global = 5  
  
# a function  
f = function(){  
    # local variable with same  
    # name as that of global variable  
    global = 2  
    print(global)  
}
```

### Global Variable

As the name suggests, Global Variables can be accessed from any part of the program.

They are available throughout the lifetime of a program.

## R-PROGRAMMING(DS3101PC/AM3101PC)

They are declared anywhere in the program outside all of the functions or blocks.

Declaring global variables: Global variables are usually declared outside of all of the functions and blocks. They can be accessed from any portion of the program.

```
# R program to illustrate
# usage of global variables
# global variable
global=5
# global variable accessed from
# within a function
display =function(){
  print(global)
}
display()
# changing value of global variable
global=10
display()
```

### Output:

```
[1] 5
```

```
[1] 10
```

In the above code, the variable 'global' is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

### Local Variable

Variables defined within a function or block are said to be local to those functions.

Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

Declaring local variables: Local variables are declared inside a block.

### Example:

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# R program to illustrate
# usage of local variables
func=function(){
  # this variable is local to the
  # function func() and cannot be
  # accessed outside this function
  age =18
}
print(age)
```

### Output:

Error in print(age) : object 'age' not found

The above program displays an error saying “object ‘age’ not found”. The variable age was declared within the function “func()” so it is local to that function and not visible to the portion of the program outside this function.

To correct the above error we have to display the value of variable age from the function “func()” only.

### Example:

```
# R program to illustrate
# usage of local variables

func=function(){
  # this variable is local to the
  # function func() and cannot be
  # accessed outside this function
  age =18
  print(age)
}
```



```
cat("Age is:\n")
```

```
func()
```

**Output:**

Age is:

[1] 18

**Accessing Global Variables**

Global Variables can be accessed from anywhere in the code unlike local variables that have a scope restricted to the block of code in which they are created.

Example:

```
f=function() {  
  # a is a local variable here  
  a <-1  
}  
f()  
# Can't access outside the function  
print(a) # This'll give error
```

**Output:**

Error in print(a) : object 'a' not found

In the above code, we see that we are unable to access variable “a” outside the function as it’s assigned by an assignment operator(<-) that makes “a” as a local variable. To make assignments to global variables, a super assignment operator(<<-) is used.

**How super assignment operator works?**

When using this operator within a function, it searches for the variable in the parent environment frame, if not found it keeps on searching the next level until it reaches the global environment. If the variable is still not found, it is created and assigned at the global level.

Example:

```
# R program to illustrate
# Scope of variables
outer_function=function(){
  inner_function=function(){
    # Note that "<<-" operator here
    # makes a as global variable
    a <<-10
    print(a)
  }
  inner_function()
  print(a)
}
outer_function()
# Can access outside the function
print(a)
```

### Output:

```
[1] 10
[1] 10
[1] 10
```

When the statement “a <<- 10” is encountered within inner\_function(), it looks for the variable “a” in the outer\_function() environment. When the search fails, it searches in R\_GlobalEnv. Since “a” is not defined in this global environment as well, it is created and assigned there which is now referenced and printed from within inner\_function() as well as outer\_function().

### Dynamic Scoping in R Programming

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool. The scoping rules for R are the main feature that makes it different from the original S language. R language uses [lexical scoping](#) or static scoping. A common alternative is Dynamic scoping.

## R-PROGRAMMING(DS3101PC/AM3101PC)

### Concept Behind Dynamic Scoping

Consider the following function:

R

```
f <- function(x, y){  
  x^2 + y/z  
}
```

This function has 2 formal arguments  $x$  and  $y$ . In the body of the function, there is another symbol  $z$ . In this case,  $z$  is a free variable. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the body). Lexical scoping in R means that the values of free variables are searched for in the environment in which the function was defined. Searching for the value of the free variable means:

If the value of the symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.

The search continues down the sequence of the parent environment until the users hit the top-level environment; this usually the global environment (workspace) or the namespace of a package.

If the value for a given symbol cannot be found once the empty environment has arrived, then an error is thrown. The value for the free variable is given in the below image.

```
> f<-function(x){x*x}  
> f  
function(x){x*x}  
> environment(f)  
<environment: R_GlobalEnv>  
> parent.env(environment(f))  
<environment: 0x00000209eed31190>  
attr(,"name")  
[1] "tools:rstudio"  
> |
```

Typically a function is defined in the global environment so that the value of free variables is just found in the user's workspace however in R one can have functions defined inside other functions. For example, have a look at the below code.

```
make.power<- function(n){
```

```
pow <- function(x){  
  x = x^n  
}  
pow  
}
```

```
cube <- make.power(3)  
square <- make.power(2)  
print(cube(3))  
print(square(3))
```

**Output:**

```
[1] 27
```

```
[1] 9
```

The power function takes the argument x and raised the power n. So that makes the function pow return inside the function value. n is a free variable and is defines in the pow function. What's in a function environment?

R

```
ls(environment(cube))
```

```
[1] "n" "pow"
```

```
get("n", environment(cube))
```

```
[1] 3
```

```
ls(environment(square))
```

```
[1] "n" "pow"
```

```
get("n", environment(square))
```

your roots to success...

[1] 2

**Let's consider another example:**

```
y <- 10
# creating a function f which takes argument x
f <- function(x){
  # y is free variable
  y <- 2
  # g function is also a free variable
  # not defined in the scope of function
  y^2 + g(x)
}
g <- function(x){
  # y is free variable
  # value of y in this function is 10
  x*y
}
```

**What is the value of y in this case?**

With the lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case, the global environment, so the value of y is 10.

With the dynamic scoping the value of y is looked up in the environment from which the function was called (sometimes referred to as calling [environment](#)).

In R the calling environment is known as the parent frame. So the value of y would be 2.

When a function is defined in the global environment and is subsequently called from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
g <- function(x){
```

```
a <- 3
# in this case x is function a formal argument and
# a is local variable and
# y is free variable
x + a + y
}
# printing value of g(2)
print(g(2))
```

**If we call g(2) it will give an error like the following:**

Output:

Error in g(2) : object 'y' not found

**After assigning value to y call g(2) as follows:**

```
g <- function(x){
  a <- 3
  x + a + y}
# assigning value to y
y <- 3
# print g(2)
print(g(2))
```

**Output:**

[1] 8

### Lexical Scoping in R Programming

Lexical Scoping in [R programming](#) means that the values of the free variables are searched for in the environment in which the function was defined. An environment is a collection of symbols, value, and pair, every environment has a parent environment it is possible for an environment to have multiple children but the only environment without the parent is the empty environment. If the value of the symbol is not found in the environment in which the function was defined then the search is continued in the parent environment. In R, the free variable bindings are resolve by

first looking in the environment in which the function was created. This is called Lexical Scoping.

### Why Lexical Scoping?

Lexical Scoping is a set of rules that helps to determine how R represents the value of a symbol. It is an in-built rule in R which automatically works at the language level. It is mostly used to specify statistical calculations. Lexical scoping looks up to symbol based on how functions were nested initially when they were created and not on how they were nested when they called upon. When we use lexical scoping we don't have to know how the function is called and to figure out where the value of the variable will be looked upon. We only have to look at the function's definition.

Lexical meaning in lexical scoping is completely different from usual English definition which means relating to words or vocabulary of a language which is distinguished from grammar and construction rather it comes from computer science term lexing meaning by which process that converts code represented as text to meaningful pieces which is comprehensible by programming language. Consider the following example:

```
f<-function(x, y)
{
  x *y *z
}
```

In this:

**x and y are formal arguments**

**z as the free variable**

Therefore, the scoping rules of the language determine how values are assigned to free variables. Free variables are not formal arguments and not local variables that are assigned inside of the function body.

### Principles of Lexical Scoping

There are four basic principles behind R's implementation of lexical scoping:

1. Name Masking
2. Functions vs variables
3. A fresh start
- 4. Dynamic Lookup**
5. Let's discuss each principle one by one.

### Name Masking

## R-PROGRAMMING(DS3101PC/AM3101PC)

The following example illustrates the most basic principle of lexical scoping, and you should have no problem predicting the output.

If variable is not defined inside the function:

Example:

```
c <-10
f<-function(a, b)
{
  a +b +c
}
f(8, 5)
```

**Output:**

```
[1] 23
```

It takes the c value as 10 and then adds these numbers and finally we are having 23 as output.

If name is not defined inside the function:

If a name isn't defined inside a function, R will look one level up.

Example:

```
a <-10
b <-function()
{
  c <-11
  c(a, c)
}
b()
```

**Output:**

```
[1] 10 11
```

**When one function is defined inside another function:**

The same rules apply if a function is defined inside another function: look inside the current



## R-PROGRAMMING(DS3101PC/AM3101PC)

function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages.

Example:

```
a <-10
g <-function(){
  b <-20
  h <-function(){
    c <-30
    c(a, b, c)
  }
  h()
}
g()
```

**Output:**

```
[1] 10 20 30
```

**When functions are created by another function:**

The same rules apply to closures, functions created by other functions.

Example:

```
a <-function(z){
  b <-10
  function(){
    z +4*b
  }
}
```

```
x <-a(10)
x()
```

**Output:**

```
[1] 50
```

R returns the accurate value of b after calling the function because x preserves the environment in which it was defined. The environment includes the value of b.

### Functions vs Variables

The same principles apply regardless of the type of associated value — finding functions works exactly the same way as finding variables:

Example:

```
a <-function(x) 10*x
b <-function(){
  a <-function(x) x +10
  a(12)
}
b()
```

### Output:

```
[1] 22
```

### A Fresh Start

When a function is called, a new environment is created every time. Each acknowledgement is completely independent because a function cannot tell what happened when it was run last time.

Example:

```
a <-function(){
  if(!exists("z"))
  {
    z <-10
  }
  else
  {
    z <-z+10
  }
  z
}
```

your roots to success...

```
}
```

a()

**Output:**

```
[1] 10
```

**Dynamic Lookup**

Lexical scoping controls where to look for values not when to look for them. R looks for the values when the function is executed not when it is created. The output of the function can be different depending on objects outside its environment.

**Example:**

```
g <-function() x^3
```

```
x <-10
```

```
g()
```

**Output:**

```
[1] 1000
```

There is a function in R which is findGlobals() from codetools and it helps us to find all global variables being used in a function and lists all the external dependencies of a function. findGlobals() find the global variables and functions which are used by the closure.

Example:

```
aGlobal<-rnorm(10)
```

```
bGlobal<-rnorm(10)
```

```
f <-function()
```

```
{
```

```
  a <-aGlobal
```

```
  b <-bGlobal
```

```
  plot(b ~ a)
```

```
}
```

```
codetools::findGlobals(f)
```

**Output:**

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[1] "{" "~" "<-" "aGlobal" "bGlobal" "plot"
```

We can manually change the environment to the empty environment `emptyenv()`. `emptyenv()` is a totally empty environment.

### Dates and Times

R has developed a special representation for dates and times. Dates are represented by the `Date` class and times are represented by the `POSIXct` or the `POSIXlt` class. Dates are stored internally as the number of days since 1970-01-01 while times are stored internally as the number of seconds since 1970-01-01.

It's not important to know the internal representation of dates and times in order to use them in R. I just thought those were fun facts.

### Dates in R

Dates are represented by the `Date` class and can be coerced from a character string using the `as.Date()` function. This is a common way to end up with a `Date` object in R.

```
>## Coerce a 'Date' object from character
```

```
> x <- as.Date("1970-01-01")
```

```
> x
```

```
[1] "1970-01-01"
```

You can see the internal representation of a `Date` object by using the `unclass()` function.

```
> unclass(x)
```

```
[1] 0
```

```
> unclass(as.Date("1970-01-02"))
```

```
[1] 1
```

### Times in R

Times are represented by the `POSIXct` or the `POSIXlt` class. `POSIXct` is just a very large integer under the hood. It uses a useful class when you want to store times in something like a data frame. `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month. This is useful when you need that kind of information.

There are a number of generic functions that work on dates and times to help you extract pieces of dates and/or times.

- **weekdays:** give the day of the week
- **months:** give the month name
- **quarters:** give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)

## R-PROGRAMMING(DS3101PC/AM3101PC)

- Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
> x <-Sys.time()
```

```
> x
```

```
[1] "2022-05-31 09:26:50 EDT"
```

```
>class(x) ## 'POSIXct' object
```

```
[1] "POSIXct""POSIXt"
```

The `POSIXlt` object contains some useful metadata.

```
> p <-as.POSIXlt(x)
```

```
>names(unclass(p))
```

```
[1] "sec""min""hour""mday""mon""year""wday""yday"
```

```
[9] "isdst""zone""gmtoff"
```

```
>p$wday## day of the week
```

```
[1] 2
```

You can also use the `POSIXct` format.

```
> x <-Sys.time()
```

```
> x      ## Already in 'POSIXct' format
```

```
[1] "2022-05-31 09:26:50 EDT"
```

```
>unclass(x) ## Internal representation
```

```
[1] 1654003610
```

```
>x$sec## Can't do this with 'POSIXct'!
```

```
Error in x$sec:$ operator is invalid for atomic vectors
```

```
> p <-as.POSIXlt(x)
```

```
>p$sec## That's better
```

```
[1] 50.11802
```

Finally, there is the `strptime()` function in case your dates are written in a different format. `strptime()` takes a character vector that has dates and times and converts them into to a `POSIXlt` object.

```
>datestring<-c("January 10, 2012 10:40", "December 9, 2011 9:10")
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
> x <-strptime(datestring, "%B %d, %Y %H:%M")
> x
[1] "2012-01-10 10:40:00 EST""2011-12-09 09:10:00 EST"
>class(x)
```

```
[1] "POSIXlt""POSIXt"
```

The weird-looking symbols that start with the % symbol are the formatting strings for dates and times. I can never remember the formatting strings. Check ?strptime for details. It's probably not worth memorizing this stuff.

### Operations on Dates and Times

You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons too (i.e. ==, <=)

```
> x <-as.Date("2012-01-01")
> y <-strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
> x-y
```

Warning: Incompatible methods ("-.Date", "-.POSIXt") for "-"

Error in x - y: non-numeric argument to binary operator

```
> x <-as.POSIXlt(x)
> x-y
```

Time difference of 356.3095 days

The nice thing about the date/time classes is that they keep track of all the annoying things about dates and times, like leap years, leap seconds, daylight savings, and time zones.

Here's an example where a leap year gets involved.

```
> x <-as.Date("2012-03-01")
> y <-as.Date("2012-02-28")
> x-y
```

**Time difference of 2 days**

Here's an example where two different time zones are in play (unless you live in GMT timezone, in which case they will be the same!).

```
>## My local time zone
> x <-as.POSIXct("2012-10-25 01:00:00")
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
> y <-as.POSIXct("2012-10-25 06:00:00", tz ="GMT")
```

```
> y-x
```

### Time difference of 1 hours

#### Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the Date class
- Times use the POSIXct and POSIXlt class
- Character strings can be coerced to Date/Time classes using the strptime function or the as.Date, as.POSIXlt, or as.POSIXct

## R DATA STRUCTURES

### R data structures are:

1. R data frames
2. R matrices
3. R lists
4. R vectors
5. R arrays
6. R factors
7. Data Frames

### Data Frames are data displayed in a format as a table.

Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

Use the data.frame() function to create a data frame:

Example

```
# Create a data frame
```

```
Data_Frame<- data.frame (
```

```
  Training = c("Strength", "Stamina", "Other"),
```

```
  Pulse = c(100, 150, 120),
```

```
  Duration = c(60, 30, 45)
```

```
)
```

```
# Print the data frame
```

```
Data_Frame
```

Output:

```
Training Pulse Duration
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
1 Strength 100    60
2 Stamina 150    30
3  Other 120    45
```

### Summarize the Data

Use the summary() function to summarize the data from a Data Frame:

Example

```
Data_Frame<- data.frame (
Training = c("Strength", "Stamina", "Other"),
Pulse = c(100, 150, 120),
Duration = c(60, 30, 45)
)
```

Data\_Frame

```
summary(Data_Frame)
```

Output:

### Training Pulse Duration

```
1 Strength 100    60
2 Stamina 150    30
3  Other 120    45
```

```
  Training  Pulse  Duration
Other :1 Min. :100.0 Min. :30.0
Stamina :1 1st Qu.:110.0 1st Qu.:37.5
Strength:1 Median :120.0 Median :45.0
      Mean :123.3 Mean :45.0
      3rd Qu.:135.0 3rd Qu.:52.5
```

```
      Max. :150.0 Max. :60.0
```

your roots to success...

You will learn more about the summary() function in the statistical part of the R tutorial.

Access Items

We can use single brackets [ ], double brackets [[ ]] or \$ to access columns from a data frame:



**Example**

```
Data_Frame<- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

```
Data_Frame[1]
```

```
Data_Frame[["Training"]]
```

```
Data_Frame$Training
```

**Output:**

```
Training
```

```
1 Strength
```

```
2 Stamina
```

```
3 Other
```

```
[1] Strength Stamina Other
```

```
Levels: Other Stamina Strength
```

```
[1] Strength Stamina Other
```

```
Levels: Other Stamina Strength
```

**Add Rows**

Use the rbind() function to add new rows in a Data Frame:

Example

```
Data_Frame<- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

```
# Add a new row
```

```
New_row_DF<- rbind(Data_Frame, c("Strength", 110, 110))
```

```
# Print the new row
```

```
New_row_DF
```

**Output:**

Training Pulse Duration

```
1 Strength 100    60
2 Stamina 150    30
3  Other 120    45
4 Strength 110   110
```

**Add Columns**

Use the cbind() function to add new columns in a Data Frame:

Example

```
Data_Frame<- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new column
New_col_DF<- cbind(Data_Frame, Steps = c(1000, 6000, 2000))

# Print the new column
New_col_DF
```

**Output:**

```
Training Pulse Duration Steps
1 Strength 100    60 1000
2 Stamina 150    30 6000
3  Other 120    45 2000
```

**Remove Rows and Columns**

Use the c() function to remove rows and columns in a Data Frame:

**Example**

```
Data_Frame<- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
```

```
)  
  
# Remove the first row and column  
Data_Frame_New<- Data_Frame[-c(1), -c(1)]  
  
# Print the new data frame  
Data_Frame_New
```

**Output:**

```
Pulse Duration  
2 150 30  
3 120 45
```

**Amount of Rows and Columns**

Use the dim() function to find the amount of rows and columns in a Data Frame:

Example

```
Data_Frame<- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

```
dim(Data_Frame)
```

Output:

```
[1] 3 3
```

You can also use the ncol() function to find the number of columns and nrow() to find the number of rows:

**Example**

```
Data_Frame<- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

```
ncol(Data_Frame)  
nrow(Data_Frame)
```

**Output:**

```
[1] 3
```

```
[1] 3
```

### Data Frame Length

Use the **length()** function to find the number of columns in a Data Frame (similar to ncol()):

#### Example

```
Data_Frame<- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
length(Data_Frame)
```

#### Output:

```
[1] 3
```

### Combining Data Frames

Use the **rbind()** function to combine two or more data frames in R vertically:

#### Example

```
Data_Frame1 <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

```
Data_Frame2 <- data.frame (  
  Training = c("Stamina", "Stamina", "Strength"),  
  Pulse = c(140, 150, 160),  
  Duration = c(30, 30, 20)  
)
```

```
New_Data_Frame<- rbind(Data_Frame1, Data_Frame2)  
New_Data_Frame
```

#### Output:

```
Training Pulse Duration  
1 Strength 100      60  
2 Stamina 150      30  
3  Other 120      45
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
4 Stamina 140 30
5 Stamina 150 30
6 Strength 160 20
```

**And use the cbind()** function to combine two or more data frames in R horizontally:

### Example

```
Data_Frame3 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)

New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4)
New_Data_Frame1
```

Output:

```
Training Pulse Duration Steps Calories
1 Strength 100 60 3000 300
2 Stamina 150 30 6000 400
3 Other 120 45 2000 300
```

### Matrices

A matrix is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.

A matrix can be created with the matrix() function. Specify the nrow and ncol parameters to get the amount of rows and columns:

### Example

```
# Create a matrix
thismatrix<- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)

# Print the matrix
thismatrix
```

Output:

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

Note: Remember the c() function is used to concatenate items together.

You can also create a matrix with strings:

Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
thismatrix
```

**output:**

```
[1] [,2]
[1,] "apple" "cherry"
[2,] "banana" "orange"
```

### Access Matrix Items

You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
thismatrix[1, 2]
```

**output:**

```
[1] "cherry"
```

The whole row can be accessed if you specify a comma after the number in the bracket:

Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
thismatrix[2,]
```

**output:**

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[1] "banana" "orange"
```

The whole column can be accessed if you specify a comma before the number in the bracket:

### Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
thismatrix[,2]
```

### output:

```
[1] "cherry" "orange"
```

### Access More Than One Row

More than one row can be accessed if you use the c() function:

### Example

```
thismatrix<-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"),  
nrow = 3, ncol = 3)
```

```
thismatrix[c(1,2),]
```

### output:

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

```
[1,] "apple" "orange" "pear"
```

```
[2,] "banana" "grape" "melon"
```

### Access More Than One Column

More than one column can be accessed if you use the c() function:

### Example

```
thismatrix<-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"),  
nrow = 3, ncol = 3)
```

```
thismatrix[, c(1,2)]
```

### output:

```
[,1] [,2]
```

```
[1,] "apple" "orange"
```

```
[2,] "banana" "grape"
```

```
[3,] "cherry" "pineapple"
```

### Add Rows and Columns

Use the cbind() function to add additional columns in a Matrix:

#### Example

```
thismatrix<-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"),  
nrow = 3, ncol = 3)
```

```
newmatrix<- cbind(thismatrix, c("strawberry", "blueberry", "raspberry"))
```

```
# Print the new matrix  
newmatrix
```

#### output:

```
[,1] [,2] [,3] [,4]  
[1,] "apple" "orange" "pear" "strawberry"  
[2,] "banana" "grape" "melon" "blueberry"  
[3,] "cherry" "pineapple" "fig" "raspberry"
```

Note: The cells in the new column must be of the same length as the existing matrix.

Use the rbind() function to add additional rows in a Matrix:

#### Example

```
thismatrix<-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"),  
nrow = 3, ncol = 3)
```

```
newmatrix<- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))
```

```
# Print the new matrix  
newmatrix
```

#### output:

```
[,1] [,2] [,3]  
[1,] "apple" "orange" "pear"  
[2,] "banana" "grape" "melon"  
[3,] "cherry" "pineapple" "fig"
```



## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[4,] "strawberry" "blueberry" "raspberry"
```

Note: The cells in the new row must be of the same length as the existing matrix.

### Remove Rows and Columns

Use the `c()` function to remove rows and columns in a Matrix:

#### Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange", "mango", "pineapple"), nrow = 3,  
ncol=2)
```

```
#Remove the first row and the first column
```

```
thismatrix<- thismatrix[-c(1), -c(1)]
```

```
thismatrix
```

#### output:

```
[1] "mango" "pineapple"
```

Check if an Item Exists

To find out if a specified item is present in a matrix, use the `%in%` operator:

#### Example

Check if "apple" is present in the matrix:

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
"apple" %in% thismatrix
```

#### output:

```
[1] TRUE
```

### Number of Rows and Columns

Use the `dim()` function to find the number of rows and columns in a Matrix:

#### Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
dim(thismatrix)
```

#### output:

```
[1] 2 2
```

### Matrix Length

Use the length() function to find the dimension of a Matrix:

#### Example

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
length(thismatrix)
```

#### output:

```
[1] 4
```

Total cells in the matrix is the number of rows multiplied by number of columns.

In the example above: Dimension =  $2*2 = 4$ .

### Loop Through a Matrix

You can loop through a Matrix using a for loop. The loop will start at the first row, moving right:

#### Example

Loop through the matrix items and print them:

```
thismatrix<- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
for (rows in 1:nrow(thismatrix)) {  
  for (columns in 1:ncol(thismatrix)) {  
    print(thismatrix[rows, columns])  
  }  
}
```

#### output:

```
[1] "apple"  
[1] "cherry"  
[1] "banana"  
[1] "orange"
```

### Combine two Matrices

Again, you can use the rbind() or cbind() function to combine two or more matrices together:

Example

```
# Combine matrices
```

```
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)
```

```
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)
```

```
# Adding it as a rows
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
Matrix_Combined<- rbind(Matrix1, Matrix2)
```

```
Matrix_Combined
```

```
# Adding it as a columns
```

```
Matrix_Combined<- cbind(Matrix1, Matrix2)
```

```
Matrix_Combined
```

### Output:

```
[,1] [,2]
```

```
[1,] "apple" "cherry"
```

```
[2,] "banana" "grape"
```

```
[3,] "orange" "pineapple"
```

```
[4,] "mango" "watermelon"
```

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

```
[1,] "apple" "cherry" "orange" "pineapple"
```

```
[2,] "banana" "grape" "mango" "watermelon"
```

### Lists

A list in R can contain many different data types inside it. A list is a collection of data which is ordered and changeable.

To create a list, use the list() function:

Example

```
# List of strings
```

```
thislist<- list("apple", "banana", "cherry")
```

```
# Print the list
```

```
thislist
```

Output:

```
[[1]]
```

```
[1] "apple"
```

```
[[2]]
```

```
[1] "banana"
```

```
[[3]]  
[1] "cherry"
```

### Access Lists

You can access the list items by referring to its index number, inside brackets. The first item has index 1, the second item has index 2, and so on:

### Example

```
thislist<- list("apple", "banana", "cherry")
```

```
thislist[1]
```

output:

```
[[1]]  
[1] "apple"
```

### Change Item Value

To change the value of a specific item, refer to the index number:

### Example

```
thislist<- list("apple", "banana", "cherry")
```

```
thislist[1] <- "blackcurrant"
```

```
# Print the updated list
```

```
thislist
```

output:

```
[[1]]  
[1] "blackcurrant"
```

```
[[2]]  
[1] "banana"
```

```
[[3]]  
[1] "cherry"
```

### List Length

To find out how many items a list has, use the length() function:

### Example

```
thislist<- list("apple", "banana", "cherry")
```

```
length(thislist)
```

output:

```
[1] 3
```

### Check if Item Exists

To find out if a specified item is present in a list, use the %in% operator:

Example

Check if "apple" is present in the list:

```
thislist<- list("apple", "banana", "cherry")
```

```
"apple" %in% thislist
```

output:

```
[1] TRUE
```

### Add List Items

To add an item to the end of the list, use the append() function:

Example

Add "orange" to the list:

```
thislist<- list("apple", "banana", "cherry")
```

```
append(thislist, "orange")
```

output:

```
[[1]]
```

```
[1] "apple"
```

```
[[2]]
```

```
[1] "banana"
```

```
[[3]]
```

```
[1] "cherry"
```

```
[[4]]
```

```
[1] "orange"
```

To add an item to the right of a specified index, add "after=index number" in the append() function:

Example

Add "orange" to the list after "banana" (index 2):

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
thislist<- list("apple", "banana", "cherry")
```

```
append(thislist, "orange", after = 2)
```

### output:

```
[[1]]  
[1] "apple"  
  
[[2]]  
[1] "banana"  
  
[[3]]  
[1] "orange"  
  
[[4]]  
[1] "cherry"
```

### Remove List Items

You can also remove list items. The following example creates a new, updated list without an "apple" item:

### Example

Remove "apple" from the list:

```
thislist<- list("apple", "banana", "cherry")
```

```
newlist<- thislist[-1]
```

```
# Print the new list  
newlist
```

output:

```
[[1]]  
[1] "banana"
```

```
[[2]]  
[1] "cherry"
```

### Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range, by using the : operator:

### Example

## R-PROGRAMMING(DS3101PC/AM3101PC)

Return the second, third, fourth and fifth item:

```
thislist<- list("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
```

```
(thislist)[2:5]
```

**output:**

```
[[1]]
```

```
[1] "banana"
```

```
[[2]]
```

```
[1] "cherry"
```

```
[[3]]
```

```
[1] "orange"
```

```
[[4]]
```

```
[1] "kiwi"
```

Note: The search will start at index 2 (included) and end at index 5 (included).

Remember that the first item has index 1.

### Loop Through a List

You can loop through the list items by using a for loop:

**Example**

Print all items in the list, one by one:

```
thislist<- list("apple", "banana", "cherry")
```

```
for (x in thislist) {  
  print(x)  
}
```

**output:**

```
[1] "apple"
```

```
[1] "banana"
```

```
[1] "cherry"
```

### Join Two Lists

There are several ways to join, or concatenate, two or more lists in R.

The most common way is to use the `c()` function, which combines two elements together:

**Example**

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
list1 <- list("a", "b", "c")
list2 <- list(1,2,3)
list3 <- c(list1,list2)
```

list3

### output:

```
[[1]]
[1] "a"

[[2]]
[1] "b"

[[3]]
[1] "c"

[[4]]
[1] 1

[[5]]
[1] 2

[[6]]
[1] 3
```

### Vectors

A vector is simply a list of items that are of the same type.

To combine the list of items to a vector, use the c() function and separate the items by a comma.

In the example below, we create a vector variable called fruits, that combine strings:

### Example

```
# Vector of strings
fruits <- c("banana", "apple", "orange")
```

```
# Print fruits
```

```
fruits
```

### Output:

```
[1] "banana" "apple" "orange"
```

In this example, we create a vector that combines numerical values:

### Example



## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# Vector of numerical values
numbers <- c(1, 2, 3)
```

```
# Print numbers
numbers
```

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

To create a vector with numerical values in a sequence, use the `:` operator:

### Example

```
# Vector with numerical values in a sequence
numbers <- 1:10
```

```
numbers
```

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

You can also create numerical values with decimals in a sequence, but note that if the last element does not belong to the sequence, it is not used:

### Example

```
# Vector with numerical decimals in a sequence
numbers1 <- 1.5:6.5
numbers1
```

```
# Vector with numerical decimals in a sequence where the last element is not used
numbers2 <- 1.5:6.3
numbers2
```

### Result:

```
[1] 1.5 2.5 3.5 4.5 5.5 6.5
[1] 1.5 2.5 3.5 4.5 5.5
```

In the example below, we create a vector of logical values:

### Example

```
# Vector of logical values
log_values <- c(TRUE, FALSE, TRUE, FALSE)
```

```
log_values
```

### Output:

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

### Vector Length

To find out how many items a vector has, use the length() function:

Example

```
fruits <- c("banana", "apple", "orange")
```

```
length(fruits)
```

output:

```
[1] 3
```

### Sort a Vector

To sort items in a vector alphabetically or numerically, use the sort() function:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
numbers <- c(13, 3, 5, 7, 20, 2)
```

```
sort(fruits) # Sort a string
```

```
sort(numbers) # Sort numbers
```

**output:**

```
[1] "apple" "banana" "lemon" "mango" "orange"
```

```
[1] 2 3 5 7 13 20
```

Access Vectors

You can access the vector items by referring to its index number inside brackets []. The first item has index 1, the second item has index 2, and so on:

Example

```
fruits <- c("banana", "apple", "orange")
```

```
# Access the first item (banana)
```

```
fruits[1]
```

output:

```
[1] "banana"
```

You can also access multiple elements by referring to different index positions with the c() function:

**Example**

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Access the first and third item (banana and orange)
```

```
fruits[c(1, 3)]
```

output:

```
[1] "banana" "orange"
```

You can also use negative index numbers to access all items except the ones specified:

### Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Access all items except for the first item
```

```
fruits[c(-1)]
```

output:

```
[1] "apple" "orange" "mango" "lemon"
```

### Change an Item

To change the value of a specific item, refer to the index number:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Change "banana" to "pear"
```

```
fruits[1] <- "pear"
```

```
# Print fruits
```

```
fruits
```

output:

```
[1] "pear" "apple" "orange" "mango" "lemon"
```

Repeat Vectors

To repeat vectors, use the rep() function:

### Example

Repeat each value:

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
repeat_each<- rep(c(1,2,3), each = 3)
```

```
repeat_each
```

```
output:
```

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

### Example

**Repeat the sequence of the vector:**

```
repeat_times<- rep(c(1,2,3), times = 3)
```

```
repeat_times
```

```
output:
```

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

### Example

Repeat each value independently:

```
repeat_indepent<- rep(c(1,2,3), times = c(5,2,1))
```

```
repeat_indepent
```

```
output:
```

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

### Generating Sequenced Vectors

One of the examples on top, showed you how to create a vector with numerical values in a sequence with the : operator:

### Example

```
numbers <- 1:10
```

```
numbers
```

```
output:
```

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

To make bigger or smaller steps in a sequence, use the seq() function:

### Example

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
numbers <- seq(from = 0, to = 100, by = 20)
```

```
numbers
```

### output:

```
[1] 0 20 40 60 80 100
```

Note: **The seq() function** has three parameters: from is where the sequence starts, to is where the sequence stops, and by is the interval of the sequence.

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.

### Vector Creation

#### Single Element Vector

Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

```
# Atomic vector of type character.
```

```
print("abc");
```

```
# Atomic vector of type double.
```

```
print(12.5)
```

```
# Atomic vector of type integer.
```

```
print(63L)
```

```
# Atomic vector of type logical.
```

```
print(TRUE)
```

```
# Atomic vector of type complex.
```

```
print(2+3i)
```

```
# Atomic vector of type raw.
```

```
print(charToRaw('hello'))
```

When we execute the above code, it produces the following result –

```
[1] "abc"  
[1] 12.5  
[1] 63  
[1] TRUE  
[1] 2+3i  
[1] 68 65 6c 6c 6f
```

### **Multiple Elements Vector**

#### **Using colon operator with numeric data**

```
# Creating a sequence from 5 to 13.
```

```
v <-5:13
```

```
print(v)
```

```
# Creating a sequence from 6.6 to 12.6.
```

```
v <-6.6:12.6
```

```
print(v)
```

```
# If the final element specified does not belong to the sequence then it is discarded.
```

```
v <-3.8:11.4
```

```
print(v)
```

When we execute the above code, it produces the following result –

```
[1] 5 6 7 8 9 10 11 12 13  
[1] 6.6 7.6 8.6 9.6 10.6 11.6 12.6  
[1] 3.8 4.8 5.8 6.8 7.8 8.8 9.8 10.8
```

#### **Using sequence (Seq.) operator**

```
# Create vector with elements from 5 to 9 incrementing by 0.4.
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
print(seq(5,9,by=0.4))
```

When we execute the above code, it produces the following result –

```
[1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
```

Using the c() function

The non-character values are coerced to character type if one of the elements is a character.

```
# The logical and numeric values are converted to characters.
```

```
s <- c('apple','red',5,TRUE)
```

```
print(s)
```

When we execute the above code, it produces the following result –

```
[1] "apple" "red" "5" "TRUE"
```

### Accessing Vector Elements

Elements of a Vector are accessed using indexing. The [ ] brackets are used for indexing. Indexing starts with position 1. Giving a negative value in the index drops that element from result.TRUE, FALSE or 0 and 1 can also be used for indexing.

```
# Accessing vector elements using position.
```

```
t <- c("Sun", "Mon", "Tue", "Wed", "Thurs", "Fri", "Sat")
```

```
u <- t[2,3,6]
```

```
print(u)
```

```
# Accessing vector elements using logical indexing.
```

```
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
```

```
print(v)
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

# Accessing vector elements using negative indexing.

```
x <- t[c(-2,-5)]
```

```
print(x)
```

# Accessing vector elements using 0/1 indexing.

```
y <- t[c(0,0,0,0,0,0,1)]
```

```
print(y)
```

When we execute the above code, it produces the following result –

```
[1] "Mon" "Tue" "Fri"
```

```
[1] "Sun" "Fri"
```

```
[1] "Sun" "Tue" "Wed" "Fri" "Sat"
```

```
[1] "Sun"
```

### Vector Manipulation

#### Vector arithmetic

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.
```

```
v1 <-c(3,8,4,5,0,11)
```

```
v2 <-c(4,11,0,8,1,2)
```

```
# Vector addition.
```

```
add.result<- v1+v2
```

```
print(add.result)
```

your roots to success...



## R-PROGRAMMING(DS3101PC/AM3101PC)

# Vector subtraction.

```
sub.result<- v1-v2
```

```
print(sub.result)
```

# Vector multiplication.

```
multi.result<- v1*v2
```

```
print(multi.result)
```

# Vector division.

```
divi.result<- v1/v2
```

```
print(divi.result)
```

When we execute the above code, it produces the following result –

```
[1] 7 19 4 13 1 13
```

```
[1] -1 -3 4 -3 -1 9
```

```
[1] 12 88 0 40 0 22
```

```
[1] 0.7500000 0.7272727      Inf 0.6250000 0.0000000 5.5000000
```

### Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

```
v1 <-c(3,8,4,5,0,11)
```

```
v2 <-c(4,11)
```

```
# V2 becomes c(4,11,4,11,4,11)
```

```
add.result<- v1+v2
```

```
print(add.result)
```

```
sub.result<- v1-v2
```

```
print(sub.result)
```

When we execute the above code, it produces the following result –

```
[1] 7 19 8 16 4 22
```

```
[1] -1 -3 0 -6 -4 0
```

### **Vector Element Sorting**

Elements in a vector can be sorted using the `sort()` function.

```
v <-c(3,8,4,5,0,11,-9,304)
```

```
# Sort the elements of the vector.
```

```
sort.result<- sort(v)
```

```
print(sort.result)
```

```
# Sort the elements in the reverse order.
```

```
revsort.result<- sort(v, decreasing = TRUE)
```

```
print(revsort.result)
```

```
# Sorting character vectors.
```

```
v <- c("Red","Blue","yellow","violet")
```

```
sort.result<- sort(v)
```

```
print(sort.result)
```

```
# Sorting character vectors in reverse order.
```

```
revsort.result<- sort(v, decreasing = TRUE)
```

```
print(revsort.result)
```

When we execute the above code, it produces the following result –

```
[1] -9 0 3 4 5 8 11 304
```

```
[1] 304 11 8 5 4 3 0 -9
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[1] "Blue" "Red" "violet" "yellow"
```

```
[1] "yellow" "violet" "Red" "Blue"
```

### UNIT-III

#### R – Lists

A list in R is a generic object consisting of an ordered collection of objects. Lists are one-dimensional, heterogeneous data structures. The list can be a list of vectors, a list of matrices, a list of characters and a list of functions, and so on.

A list is a vector but with heterogeneous data elements. A list in R is created with the use of list() function. R allows accessing elements of a list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0 like other programming languages.

#### Creating a List

To create a List in R you need to use the function called “list()”. In other words, a list is a generic vector containing other objects. To illustrate how a list looks, we take an example here. We want to build a list of employees with the details. So for this, we want attributes such as ID, employee name, and the number of employees.

#### Example:

R program to create a List

```
# The first attributes is a numeric vector
# containing the employee IDs which is created
# using the command here
empId = c(1, 2, 3, 4)

# The second attribute is the employee name
# which is created using this line of code here
# which is the character vector
empName = c("Debi", "Sandeep", "Subham", "Shiba")
```

```
# The third attribute is the number of employees
```

```
# which is a single numeric variable.
```

```
numberOfEmp = 4
```

```
# We can combine all these three different
```

```
# data types into a list
```

```
# containing the details of employees
```

```
# which can be done using a list command
```

```
empList = list(empId, empName, numberOfEmp)
```

```
print(empList)
```

Output:

```
[[1]]
```

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

```
[[2]]
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
[[3]]
```

```
[1] 4
```

### **Accessing components of a list**

We can access components of a list in two ways.

Access components by names: All the components of a list can be named and we can use those names to access the components of the list using the dollar command.

Example:

R

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# R program to access
# components of a list

# Creating a list by naming all its components
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
print(empList)

# Accessing components by names
cat("Accessing name components using $ command\n")
print(empList$Names)
```

Output:

```
$ID
```

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$`Total Staff`
```

```
[1] 4
```

### Accessing name components using \$ command

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

Access components by indices: We can also access the components of the list using indices. To access the top-level components of a list we have to use a double slicing operator “[ [ ] ]” which is two square brackets and if we want to access the lower or inner level components of a list we have to use another square bracket “[ ]” along with the double slicing operator “[ [ ] ]”.

Example:

R

```
# R program to access
```

```
# components of a list
```

```
# Creating a list by naming all its components
```

```
empId = c(1, 2, 3, 4)
```

```
empName = c("Debi", "Sandeep", "Subham", "Shiba")
```

```
numberOfEmp = 4
```

```
empList = list(
```

```
  "ID" = empId,
```

```
  "Names" = empName,
```

```
  "Total Staff" = numberOfEmp
```

```
)
```

```
print(empList)
```

```
# Accessing a top level components by indices
```

```
cat("Accessing name components using indices\n")
```

```
print(empList[[2]])
```

```
# Accessing a inner level components by indices
```

```
cat("Accessing Sandeep from name using indices\n")
```

```
print(empList[[2]][2])
```

```
# Accessing another inner level components by indices
```

```
cat("Accessing 4 from ID using indices\n")
```

```
print(empList[[1]][4])
```

Output:

```
$ID
```

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$`Total Staff`
```

```
[1] 4
```

### **Accessing name components using indices**

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

Accessing Sandeep from name using indices

```
[1] "Sandeep"
```

Accessing 4 from ID using indices

```
[1] 4
```

### **Modifying components of a list**

A list can also be modified by accessing the components and replacing them with the ones which you want.

Example:

R

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# R program to edit
# components of a list

# Creating a list by naming all its components
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
cat("Before modifying the list\n")
print(empList)

# Modifying the top-level component
empList$`Total Staff` = 5

# Modifying inner level component
empList[[1]][5] = 5
empList[[2]][5] = "Kamala"
```

```
cat("After modified the list\n")
```

```
print(empList)
```

Output:

Before modifying the list

\$ID



## R-PROGRAMMING(DS3101PC/AM3101PC)

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$`Total Staff`
```

```
[1] 4
```

### After modified the list

```
$ID
```

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba" "Kamala"
```

```
$`Total Staff`
```

```
[1] 5
```

### Concatenation of lists

Two lists can be concatenated using the concatenation function. So, when we want to concatenate two lists we have to use the concatenation operator.

Syntax:

```
list = c(list, list1)
```

```
list = the original list
```

```
list1 = the new list
```

Example:

```
R
```

```
# R program to edit
```

```
# components of a list
```

```
# Creating a list by naming all its components  
empId = c(1, 2, 3, 4)  
empName = c("Debi", "Sandeep", "Subham", "Shiba")  
numberOfEmp = 4  
empList = list(  
  "ID" = empId,  
  "Names" = empName,  
  "Total Staff" = numberOfEmp  
)  
cat("Before concatenation of the new list\n")  
print(empList)
```

```
# Creating another list  
empAge = c(34, 23, 18, 45)  
empAgeList = list(  
  "Age" = empAge  
)
```

```
# Concatenation of list using concatenation operator  
empList = c(empList, empAgeList)
```

```
cat("After concatenation of the new list\n")  
print(empList)
```

Output:

Before concatenation of the new list

\$ID

## R-PROGRAMMING(DS3101PC/AM3101PC)

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$`Total Staff`
```

```
[1] 4
```

After concatenation of the new list

```
$ID
```

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$`Total Staff`
```

```
[1] 4
```

```
$Age
```

```
[1] 34 23 18 45
```

### Deleting components of a list

To delete components of a list, first of all, we need to access those components and then insert a negative sign before those components. It indicates that we had to delete that component.

#### Example:

```
R
```

```
# R program to access
```

```
# components of a list
```

```
# Creating a list by naming all its components
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
cat("Before deletion the list is\n")
print(empList)
```

```
# Deleting a top level components
cat("After Deleting Total staff components\n")
print(empList[-3])
```

```
# Deleting a inner level components
cat("After Deleting sandeep from name\n")
print(empList[[2]][-2])
```

Output:

Before deletion the list is

```
$ID
[1] 1 2 3 4
```

```
$Names
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

```
$Total Staff
```

```
[1] 4
```

**After Deleting Total staff components**

```
$ID
```

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

```
$Names
```

```
[1] "Debi" "Sandeep" "Subham" "Shiba"
```

**After Deleting sandeep from name**

```
[1] "Debi" "Subham" "Shiba"
```

Merging list

We can merge the list by placing all the lists into a single list.

```
R
```

```
# Create two lists.
```

```
lst1 <- list(1,2,3)
```

```
lst2 <- list("Sun","Mon","Tue")
```

```
# Merge the two lists.
```

```
new_list<- c(lst1,lst2)
```

```
# Print the merged list.
```

```
print(new_list)
```

Output:

```
[[1]]
```

[1] 1

[[2]]

[1] 2

[[3]]

[1] 3

[[4]]

[1] "Sun"

[[5]]

[1] "Mon"

[[6]]

[1] "Tue"

### Converting List to Vector

Here we are going to convert the list to vector, for this we will create a list first and then unlist the list into the vector.

R

```
# Create lists.
```

```
lst<- list(1:5)
```

```
print(lst)
```

your roots to success...

```
# Convert the lists to vectors.
```

```
vec<- unlist(lst)
```

```
print(vec)
```

Output:

```
[[1]]
```

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

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

### R List to matrix

We will create matrices using `matrix()` function in R programming. Another function that will be used is `unlist()` function to convert the lists into a vector.

R

```
# Defining list
```

```
lst1 <- list(list(1, 2, 3),  
             list(4, 5, 6))
```

```
# Print list
```

```
cat("The list is:\n")
```

```
print(lst1)
```

```
cat("Class:", class(lst1), "\n")
```

```
# Convert list to matrix
```

```
mat <- matrix(unlist(lst1), nrow = 2, byrow = TRUE)
```

```
# Print matrix
```

```
cat("\nAfter conversion to matrix:\n")
```

```
print(mat)
```

```
cat("Class:", class(mat), "\n")
```

Output:

The list is:

```
[[1]]
```

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

```
[1] 1
```

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

```
[1] 2
```

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

```
[1] 3
```

```
[[2]]
```

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

```
[1] 4
```

```
[[2]][[2]]
```

```
[1] 5
```

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

```
[1] 6
```



your roots to success...

Class: list

After conversion to matrix:

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



[1,] 1 2 3

[2,] 4 5 6

Class: matrix

### List of Dataframes in R

DataFrames are generic data objects of R which are used to store the tabular data. They are two-dimensional, heterogeneous data structures. A list in R, however, comprises of elements, vectors, data frames, variables, or lists that may belong to different data types. In this article, we will study how to create a list consisting of data frames as its components and how to access, modify, and delete these data frames to lists. list() function in R creates a list of the specified arguments. The data frames specified as arguments in this function may have different lengths.

Operations that can be performed on a list of DataFrames are:

- Creating a list of Dataframes
- Accessing components of a list of Dataframes
- Modifying components of a list of Dataframes
- Concatenation of lists of Dataframes
- Deleting components of a list of Dataframes

### Creating a list of Dataframes

To create a list of Dataframes we use the list() function in R and then pass each of the data frame you have created as arguments to the function.

Example:

```
# R program to create list of data frames
```

```
# Create dataframe
```

```
df1 =data.frame( y1 =c(1, 2, 3), y2 =c(4, 5, 6))
```

```
# Create another dataframe
```

```
df2 =data.frame( y1 =c(7, 8, 9), y2 =c(1, 4, 6))
```

```
# Create list of data frame using list()
listOfDataframe =list(df1, df2)
print(listOfDataframe)
```

Output:

```
[[1]]
 y1 y2
1 1 4
2 2 5
3 3 6
```

```
[[2]]
 y1 y2
1 7 1
2 8 4
3 9 6
```

### Accessing components of a list of Dataframes

We can access components of a list of data frames in two ways.

Access components by names: All the components of a list of data frames can be named and we can use those names to access the components of the list using the dollar command.

Example:

your roots to success...

```
Python3
# R program to access components
# of a list of data frames
```

```
# Create dataframe
```

```
df1 =data.frame(  
y1 =c(1, 2, 3),  
y2 =c(4, 5, 6)  
)
```

```
# Create another dataframe
```

```
df2 =data.frame(  
y1 =c(7, 8, 9),  
y2 =c(1, 4, 6)  
)
```

```
# Creating a list of data frames
```

```
# by naming all its components
```

```
listOfDataframe =list(  
"Dataframe1"=df1,  
"Dataframe2"=df2  
)  
print(listOfDataframe)
```

```
# Accessing components by names
```

```
cat("Accessing Dataframe2 using $ command\n")
```

```
print(listOfDataframe$Dataframe2)
```

Output:

```
$Dataframe1
```

```
y1 y2
```

```
1 1 4
```

```
2 2 5
```

```
3 3 6
```

```
$Dataframe2
```

```
y1 y2
```

```
1 7 1
```

```
2 8 4
```

```
3 9 6
```

```
Accessing Dataframe2 using $ command
```

```
y1 y2
```

```
1 7 1
```

```
2 8 4
```

```
3 9 6
```

**Access components by indices:** We can also access the components of the list of data frames using indices. To access the top-level components of a list of data frames we have to use a double slicing operator “[ [ ] ]” which is two square brackets and if we want to access the lower or inner level components of a list we have to use another square bracket “[ ]” along with the double slicing operator “[ [ ] ]”.

Example:

```
# R program to access components
```

```
# of a list of data frames
```

```
# Create dataframe
```

```
df1 =data.frame(
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
y1 =c(1, 2, 3),
y2 =c(4, 5, 6)
)

# Create another dataframe
df2 =data.frame(
  y1 =c(7, 8, 9),
  y2 =c(1, 4, 6)
)

# Creating a list of data frames
# by naming all its components
listOfDataframe =list(
  "Dataframe1"=df1,
  "Dataframe2"=df2
)

print(listOfDataframe)

# Accessing a top level components by indices
cat("Accessing Dataframe2 using indices\n")
print(listOfDataframe[[2]])

# Accessing a inner level components by indices
cat("Accessing second column from Dataframe1 using indices\n")
print(listOfDataframe[[1]][2])
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
# Accessing another inner level components by indices
cat("Accessing 4 from Dataframe2 using indices\n")
# Here [2, 2] represents that I want
# to access element from second row and second column i.e 4 here
print(listOfDataframe[[2]][2, 2])
```

Output:

```
$Dataframe1
```

```
  y1 y2
1  1  4
2  2  5
3  3  6
```

```
$Dataframe2
```

```
  y1 y2
1  7  1
2  8  4
3  9  6
```

```
Accessing Dataframe2 using indices
```

```
  y1 y2
1  7  1
2  8  4
3  9  6
```

```
Accessing second column from Dataframe1 using indices
```

```
  y2
```



your roots to success...

1 4

2 5

3 6

Accessing 4 from Dataframe2 using indices

[1] 4

### Modifying components of a list of Dataframes

A list of data frames can also be modified by accessing the components and replacing them with the ones which you want.

Example:

```
# R program to modify components
```

```
# of a list of data frames
```

```
# Create dataframe
```

```
df1 =data.frame( y1 =c(1, 2, 3), y2 =c(4, 5, 6))
```

```
# Create another dataframe
```

```
df2 =data.frame( y1 =c(7, 8, 9), y2 =c(1, 4, 6))
```

```
# Creating a list of data frames
```

```
# by naming all its components
```

```
listOfDataframe =list( "Dataframe1"=df1, "Dataframe2"=df2)
```

```
cat("Before modifying the list of data frame\n")
```

```
print(listOfDataframe)
```

```
# Modifying the dataframe2
```

```
listOfDataframe$Dataframe2 =data.frame( y1 =c(70, 80, 9), y2 =c(14, 41, 63))
```

```
# Modifying second column from Dataframe1
```

```
listOfDataframe[[1]][2] =c(23, 45, 67)
```

```
# Modifying element 2 from dataframe1
```

```
listOfDataframe[[1]][2, 1] =15
```

```
cat("After modified the list of data frame\n")
```

```
print(listOfDataframe)
```

Output:

Before modifying the list of data frame

```
$Dataframe1
```

```
  y1 y2
```

```
1  1  4
```

```
2  2  5
```

```
3  3  6
```

```
$Dataframe2
```

```
  y1 y2
```

```
1  7  1
```

```
2  8  4
```

```
3  9  6
```

your roots to success...



## R-PROGRAMMING(DS3101PC/AM3101PC)

After modified the list of data frame

```
$Dataframe1
```

```
  y1 y2
```

```
1  1 23
```

```
2 15 45
```

```
3  3 67
```

```
$Dataframe2
```

```
  y1 y2
```

```
1 70 14
```

```
2 80 41
```

```
3  9 63
```

### Concatenation of lists of Dataframes

Two lists of data frames can be concatenated using the concatenation function. So, when we want to concatenate two lists of data frames we have to use the concatenation operator.

Syntax:

```
list = c(list, list1)
```

list = the original list of the data frame

list1 = the new list of the data frame

### Example:

```
Python3
```

```
# R program concatenation
```

```
# of lists of data frames
```

```
# Create dataframe
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
df1 =data.frame(y1 =c(1, 2, 3), y2 =c(4, 5, 6))  
  
# Create another dataframe  
  
df2 =data.frame( y1 =c(7, 8, 9), y2 =c(1, 4, 6))  
  
# Creating a list of data frames  
# by naming all its components  
listOfDataframe =list( "Dataframe1"=df1, "Dataframe2"=df2)  
cat("Before concatenation of the new list of data frame\n")  
print(listOfDataframe)  
  
# Creating another one list of data frame  
df3 =data.frame(  
  y1 =c(7, 8, 98),  
  y2 =c(10, 44, 6)  
)  
newListOfDataframe =list(  
  "Dataframe3"=df3  
)  
  
# Concatenation of list of data frames  
# using concatenation operator  
listOfDataframe =c(listOfDataframe, newListOfDataframe)
```

```
cat("After concatenation of the new list of data frame\n")  
print(listOfDataframe)
```

Output:

## R-PROGRAMMING(DS3101PC/AM3101PC)

Before concatenation of the new list of data frame

```
$Dataframe1
```

```
  y1 y2
```

```
1 1 4
```

```
2 2 5
```

```
3 3 6
```

```
$Dataframe2
```

```
  y1 y2
```

```
1 7 1
```

```
2 8 4
```

```
3 9 6
```

After concatenation of the new list of data frame

```
$Dataframe1
```

```
  y1 y2
```

```
1 1 4
```

```
2 2 5
```

```
3 3 6
```

```
$Dataframe2
```

```
  y1 y2
```

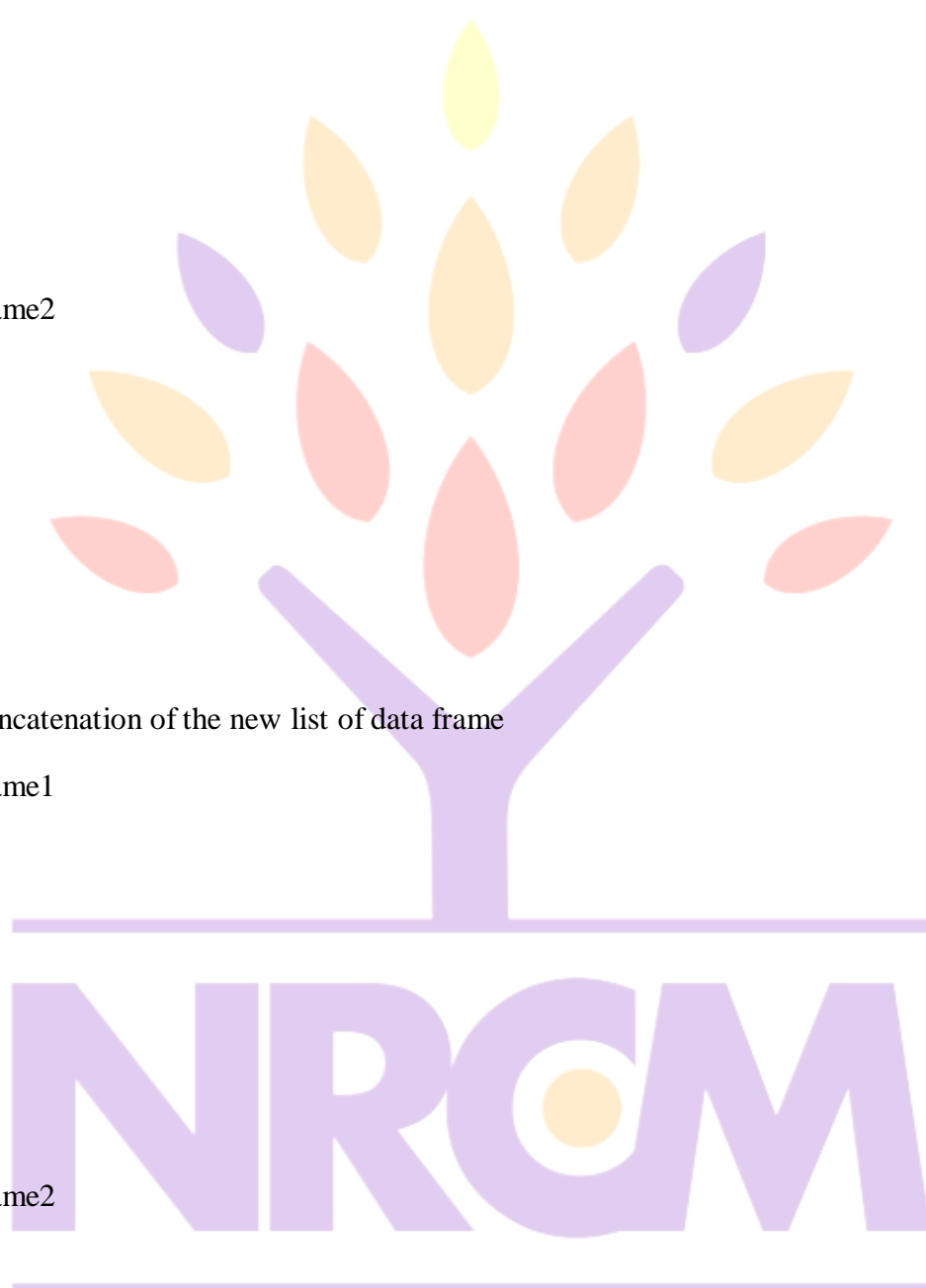
```
1 7 1
```

```
2 8 4
```

```
3 9 6
```

```
$Dataframe3
```

```
  y1 y2
```



1 7 10

2 8 44

3 98 6

### Deleting components from a list of Dataframes

To delete components of a list of data frames, first of all, we need to access those components and then insert a negative sign before those components. It indicates that we had to delete that component.

#### Example:

```
# R program to delete components  
# of a list of data frames
```

```
# Create dataframe
```

```
df1 =data.frame(  
y1 =c(1, 2, 3),
```

```
y2 =c(4, 5, 6)
```

```
)
```

```
# Create another dataframe
```

```
df2 =data.frame(  
y1 =c(7, 8, 9),
```

```
y2 =c(1, 4, 6)
```

```
)
```

```
# Creating a list of data frames
```

```
# by naming all its components
```

```
listOfDataframe =list(  
"Dataframe1"=df1,
```

```
"Dataframe2"=df2,
```

```
"Dataframe3"=df3)
```

```
"Dataframe2"=df2
)
cat("Before deletion the list is\n")
print(listOfDataframe)

# Deleting a top level components
cat("After Deleting Dataframe1\n")
print(listOfDataframe[[-1]])

# Deleting a inner level components
cat("After Deleting first column from Dataframe2\n")
print(listOfDataframe[[2]][-1])
```

Output:

Before deletion the list is

```
$Dataframe1
```

```
  y1 y2
```

```
1  1  4
```

```
2  2  5
```

```
3  3  6
```

```
$Dataframe2
```

```
  y1 y2
```

```
1  7  1
```

```
2  8  4
```

```
3  9  6
```

your roots to success...

After Deleting Dataframe1

```
y1 y2
```

```
1 7 1
```

```
2 8 4
```

```
3 9 6
```

After Deleting first column from Dataframe2

```
y2
```

```
1 1
```

```
2 4
```

```
3 6
```

### MATRIX

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

A Matrix is created using the `matrix()` function.

#### Syntax

The basic syntax for creating a matrix in R is –

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Following is the description of the parameters used –

**data** is the input vector which becomes the data elements of the matrix.

**nrow** is the number of rows to be created.

**ncol** is the number of columns to be created.

**byrow** is a logical clue. If **TRUE** then the input vector elements are arranged by row.

**dimname** is the names assigned to the rows and columns.

Example

Create a matrix taking a vector of numbers as input.

```
# Elements are arranged sequentially by row.
```

```
M <- matrix(c(3:14),nrow=4,byrow= TRUE)
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
print(M)
```

```
# Elements are arranged sequentially by column.
```

```
N <- matrix(c(3:14),nrow=4,byrow= FALSE)
```

```
print(N)
```

```
# Define the column and row names.
```

```
rownames=c("row1","row2","row3","row4")
```

```
colnames=c("col1","col2","col3")
```

```
P <- matrix(c(3:14),nrow=4,byrow= TRUE,dimnames= list(rownames,colnames))
```

```
print(P)
```

When we execute the above code, it produces the following result –

```
  [,1] [,2] [,3]
[1,]  3  4  5
[2,]  6  7  8
[3,]  9 10 11
[4,] 12 13 14
```

```
  [,1] [,2] [,3]
[1,]  3  7 11
[2,]  4  8 12
[3,]  5  9 13
[4,]  6 10 14
```

```
  col1 col2 col3
row1  3  4  5
row2  6  7  8
row3  9 10 11
row4 12 13 14
```

### Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below.

```
# Define the column and row names.
```

```
rownames=c("row1","row2","row3","row4")
```

```
colnames=c("col1","col2","col3")
```

```
# Create the matrix.
```

```
P <- matrix(c(3:14),nrow=4,byrow= TRUE,dimnames= list(rownames,colnames))
```

```
# Access the element at 3rd column and 1st row.
```

```
print(P[1,3])
```

```
# Access the element at 2nd column and 4th row.
```

```
print(P[4,2])
```

```
# Access only the 2nd row.
```

```
print(P[2,])
```

```
# Access only the 3rd column.
```

```
print(P[,3])
```

When we execute the above code, it produces the following result –

```
[1] 5
```

```
[1] 13
```

```
col1 col2 col3
```

```
6 7 8
```

```
row1 row2 row3 row4
```

```
5 8 11 14
```



## Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.

The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

### Matrix Addition & Subtraction

# Create two 2x3 matrices.

```
matrix1 <-matrix(c(3,9,-1,4,2,6),nrow=2)
```

```
print(matrix1)
```

```
matrix2 <-matrix(c(5,2,0,9,3,4),nrow=2)
```

```
print(matrix2)
```

# Add the matrices.

```
result <- matrix1 + matrix2
```

```
cat("Result of addition","\n")
```

```
print(result)
```

# Subtract the matrices

```
result <- matrix1 - matrix2
```

```
cat("Result of subtraction","\n")
```

```
print(result)
```

When we execute the above code, it produces the following result –

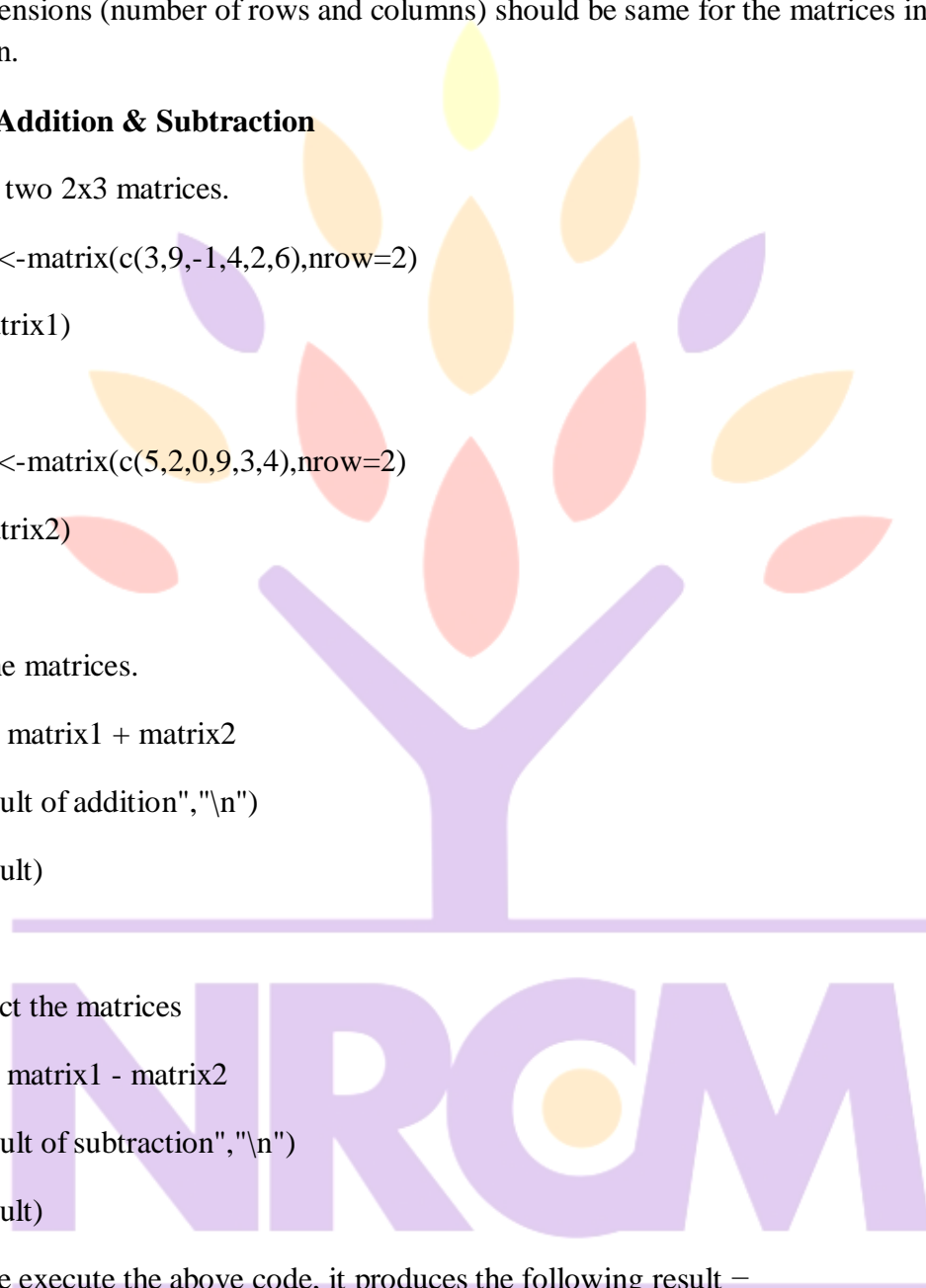
```
[,1] [,2] [,3]
```

```
[1,] 3 -1 2
```

```
[2,] 9 4 6
```

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

```
[1,] 5 0 3
```



your roots to success...

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
[2,] 2 9 4
```

Result of addition

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

```
[1,] 8 -1 5
```

```
[2,] 11 13 10
```

Result of subtraction

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

```
[1,] -2 -1 -1
```

```
[2,] 7 -5 2
```

Matrix Multiplication & Division

[Live Demo](#)

```
# Create two 2x3 matrices.
```

```
matrix1 <-matrix(c(3,9,-1,4,2,6),nrow=2)
```

```
print(matrix1)
```

```
matrix2 <-matrix(c(5,2,0,9,3,4),nrow=2)
```

```
print(matrix2)
```

```
# Multiply the matrices.
```

```
result <- matrix1 * matrix2
```

```
cat("Result of multiplication","\n")
```

```
print(result)
```

```
# Divide the matrices
```

```
result <- matrix1 / matrix2
```

```
cat("Result of division","\n")
```

```
print(result)
```

When we execute the above code, it produces the following result –

## R-PROGRAMMING(DS3101PC/AM3101PC)

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

```
[1,] 3 -1 2
```

```
[2,] 9 4 6
```

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

```
[1,] 5 0 3
```

```
[2,] 2 9 4
```

Result of multiplication

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

```
[1,] 15 0 6
```

```
[2,] 18 36 24
```

Result of division

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

```
[1,] 0.6 -Inf 0.6666667
```

```
[2,] 4.5 0.4444444 1.5000000
```

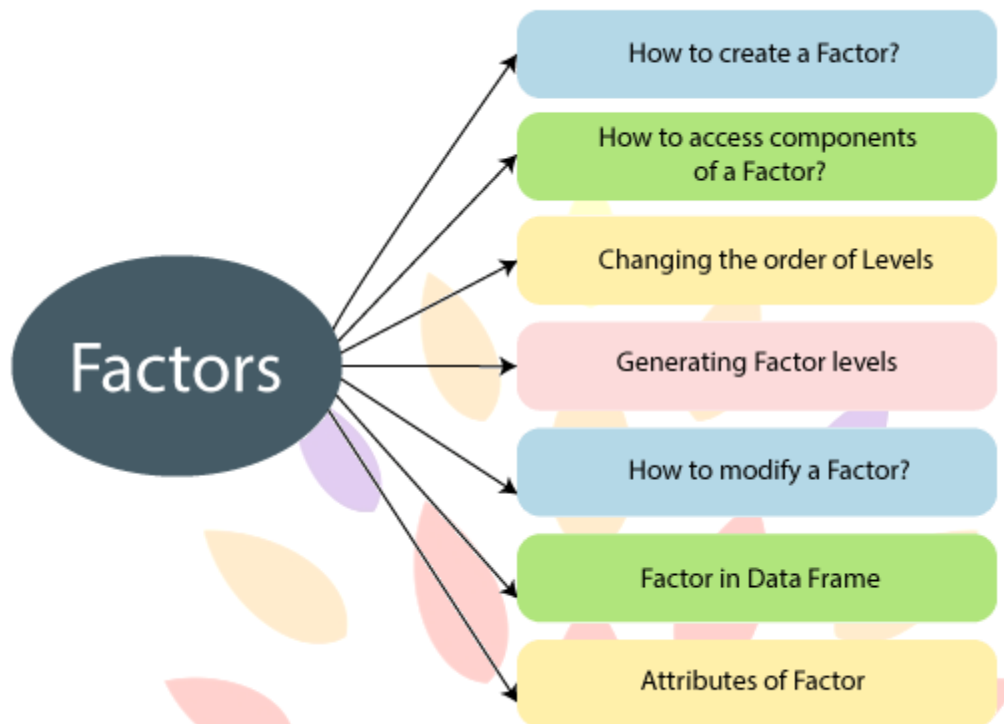
---

### UNIT-IV

#### R factors

The factor is a data structure which is used for fields which take only predefined finite number of values. These are the variable which takes a limited number of different values. These are the data objects which are used to categorize the data and to store it on multiple levels. It can store both integers and strings values, and are useful in the column that has a limited number of unique values.

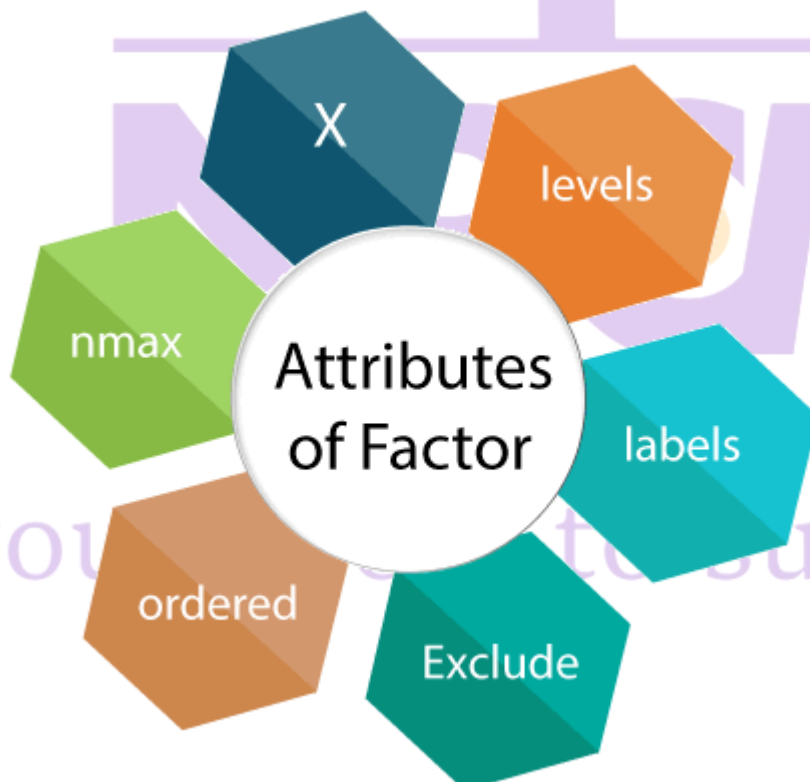
your roots to success...



Factors have labels which are associated with the unique integers stored in it. It contains predefined set value known as levels and by default R always sorts levels in alphabetical order.

#### Attributes of a factor

There are the following attributes of a factor in R



### **X**

It is the input vector which is to be transformed into a factor.

### **levels**

It is an input vector that represents a set of unique values which are taken by x.

### **labels**

It is a character vector which corresponds to the number of labels.

### **Exclude**

It is used to specify the value which we want to be excluded,

### **ordered**

It is a logical attribute which determines if the levels are ordered.

### **nmax**

It is used to specify the upper bound for the maximum number of level.

### **How to create a factor?**

In R, it is quite simple to create a factor. A factor is created in two steps

1. In the first step, we create a vector.
2. Next step is to convert the vector into a factor,

R provides factor() function to convert the vector into factor. There is the following syntax of factor() function

```
factor_data<- factor(vector)
```

Let's see an example to understand how factor function is used.

### **Example**

```
# Creating a vector as input.
```

```
data <-
```

```
c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","Arpita","Sumit")
```

```
print(data)
```

```
print(is.factor(data))
```

```
# Applying the factor function.
```

```
factor_data<- factor(data)
```

```
print(factor_data)
print(is.factor(factor_data))
```

### Output

```
[1] "Shubham" "Nishka" "Arpita" "Nishka" "Shubham" "Sumit" "Nishka"
 [8] "Shubham" "Sumit" "Arpita" "Sumit"
[1] FALSE
 [1] Shubham Nishka Arpita Nishka Shubham SumitNishka Shubham Sumit
[10] Arpita Sumit
Levels: Arpita Nishka Shubham Sumit
[1] TRUE
```

### Accessing components of factor

Like vectors, we can access the components of factors. The process of accessing components of factor is much more similar to the vectors. We can access the element with the help of the indexing method or using logical vectors. Let's see an example in which we understand the different-different ways of accessing the components.

Example

```
# Creating a vector as input.
```

```
data <-
c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","A
rpita","Sumit")
```

```
# Applying the factor function.
```

```
factor_data<- factor(data)
```

```
#Printing all elements of factor
```

```
print(factor_data)
```

```
#Accessing 4th element of factor
```

```
print(factor_data[4])
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

#Accessing 5th and 7th element

```
print(factor_data[c(5,7)])
```

#Accessing all element except 4th one

```
print(factor_data[-4])
```

#Accessing elements using logical vector

```
print(factor_data[c(TRUE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,TRUE)])
```

### Output

```
[1] Shubham Nishka Arpita Nishka Shubham SumitNishka Shubham Sumit
```

```
[10] Arpita Sumit
```

Levels: Arpita Nishka Shubham Sumit

```
[1] Nishka
```

Levels: Arpita Nishka Shubham Sumit

```
[1] Shubham Nishka
```

Levels: Arpita Nishka Shubham Sumit

```
[1] Shubham Nishka Arpita Shubham SumitNishka Shubham Sumit Arpita
```

```
[10] Sumit
```

Levels: Arpita Nishka Shubham Sumit

your roots to success...

```
[1] Shubham ShubhamSumitNishkaSumit
```

Levels: Arpita Nishka Shubham Sumit

### Modification of factor

## R-PROGRAMMING(DS3101PC/AM3101PC)

Like data frames, R allows us to modify the factor. We can modify the value of a factor by simply re-assigning it. In R, we cannot choose values outside of its predefined levels means we cannot insert value if it's level is not present on it. For this purpose, we have to create a level of that value, and then we can add it to our factor.

Let's see an example to understand how the modification is done in factors.

Example

```
# Creating a vector as input.
```

```
data <- c("Shubham","Nishka","Arpita","Nishka","Shubham")
```

```
# Applying the factor function.
```

```
factor_data<- factor(data)
```

```
#Printing all elements of factor
```

```
print(factor_data)
```

```
#Change 4th element of factor with sumit
```

```
factor_data[4] <- "Arpita"
```

```
print(factor_data)
```

```
#change 4th element of factor with "Gunjan"
```

```
factor_data[4] <- "Gunjan" # cannot assign values outside levels
```

```
print(factor_data)
```

```
#Adding the value to the level
```

```
levels(factor_data) <- c(levels(factor_data),"Gunjan")#Adding new level
```

```
factor_data[4] <- "Gunjan"
```

```
print(factor_data)
```

### Output

```
[1] Shubham Nishka Arpita Nishka Shubham
```



## R-PROGRAMMING(DS3101PC/AM3101PC)

Levels: Arpita Nishka Shubham

```
[1] Shubham Nishka Arpita Arpita Shubham
```

Levels: Arpita Nishka Shubham

Warning message:

```
In `[<-.factor`(`*tmp*`, 4, value = "Gunjan") :
```

```
  invalid factor level, NA generated
```

```
[1] Shubham NishkaArpita Shubham
```

Levels: Arpita Nishka Shubham

```
[1] Shubham Nishka Arpita Gunjan Shubham
```

Levels: Arpita Nishka Shubham Gunjan

### Factor in Data Frame

When we create a frame with a column of text data, R treats this text column as categorical data and creates factor on it.

Example

```
# Creating the vectors for data frame.
```

```
height <- c(132,162,152,166,139,147,122)
```

```
weight <- c(40,49,48,40,67,52,53)
```

```
gender <- c("male","male","female","female","male","female","male")
```

```
# Creating the data frame.
```

```
input_data<- data.frame(height,weight,gender)
```

```
print(input_data)
```

```
# Testing if the gender column is a factor.
```

```
print(is.factor(input_data$gender))
```

```
# Printing the gender column to see the levels.
```

```
print(input_data$gender)
```

**Output**

height weight gender

1 132 40 male

2 162 49 male

3 152 48 female

4 166 40 female

5 139 67 male

6 147 52 female

7 122 53 male

[1] TRUE

[1] male male female female male female male

Levels: female male

**Changing order of the levels**

In R, we can change the order of the levels in the factor with the help of the factor function.

Example

```
data <- c("Nishka","Gunjan","Shubham","Arpita","Arpita","Sumit","Gunjan","Shubham")
```

```
# Creating the factors
```

```
factor_data<- factor(data)
```

```
print(factor_data)
```

```
# Apply the factor function with the required order of the level.
```

```
new_order_factor<-
```

```
factor(factor_data,levels = c("Gunjan","Nishka","Arpita","Shubham","Sumit"))
```

```
print(new_order_factor)
```

Output

[1] Nishka Gunjan Shubham Arpita ArpitaSumit Gunjan Shubham

Levels: Arpita Gunjan Nishka Shubham Sumit

[1] Nishka Gunjan Shubham Arpita ArpitaSumit Gunjan Shubham

## R-PROGRAMMING(DS3101PC/AM3101PC)

Levels: Gunjan Nishka Arpita Shubham Sumit

### Generating Factor Levels

R provides `gl()` function to generate factor levels. This function takes three arguments i.e., `n`, `k`, and `labels`. Here, `n` and `k` are the integers which indicate how many levels we want and how many times each level is required.

There is the following syntax of `gl()` function which is as follows

```
gl(n, k, labels)
```

`n` indicates the number of levels.

`k` indicates the number of replications.

`labels` is a vector of labels for the resulting factor levels.

Example

```
gen_factor<- gl(3,5,labels=c("BCA","MCA","B.Tech"))
```

```
gen_factor
```

Output

```
[1] BCA BCABCABCABCA MCA MCAMCAMCAMCA
```

```
[11] B.TechB.TechB.TechB.TechB.Tech
```

```
Levels: BCA MCA B.Tech
```

### R Functions

A set of statements which are organized together to perform a specific task is known as a function. R provides a series of in-built functions, and it allows the user to create their own functions. Functions are used to perform tasks in the modular approach.

Functions are used to avoid repeating the same task and to reduce complexity. To understand and maintain our code, we logically break it into smaller parts using the function. A function should be

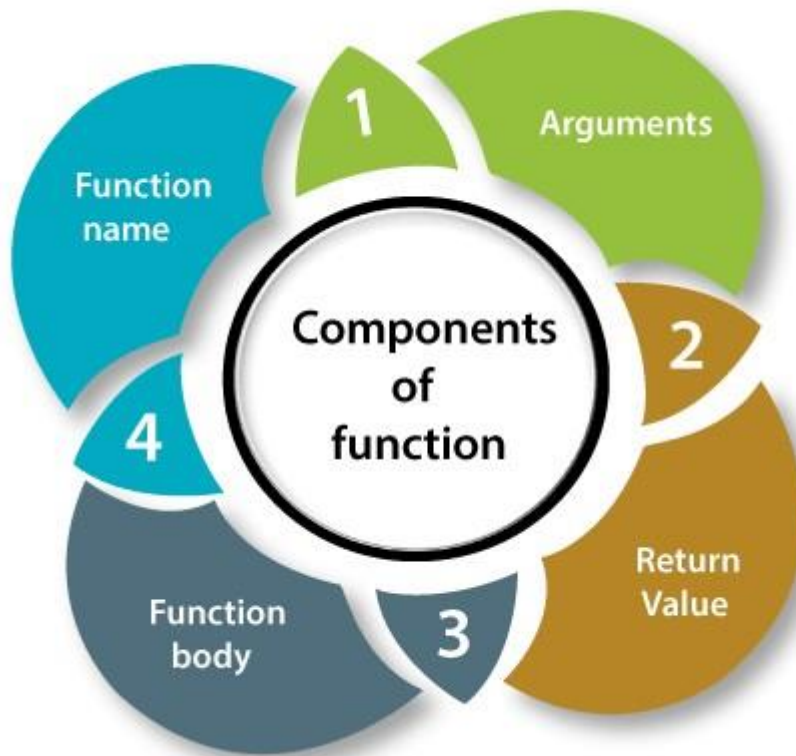
- Written to carry out a specified task.
- May or may not have arguments
- Contain a body in which our code is written.
- May or may not return one or more output values.

"An R function is created by using the keyword `function`." There is the following syntax of R function:

```
func_name <- function(arg_1, arg_2, ...) {  
  Function body  
}
```

### Components of Functions

There are four components of function, which are as follows:



#### Function Name

The function name is the actual name of the function. In R, the function is stored as an object with its name.

#### Arguments

In R, an argument is a placeholder. In function, arguments are optional means a function may or may not contain arguments, and these arguments can have default values also. We pass a value to the argument when a function is invoked.

#### Function Body

The function body contains a set of statements which defines what the function does.

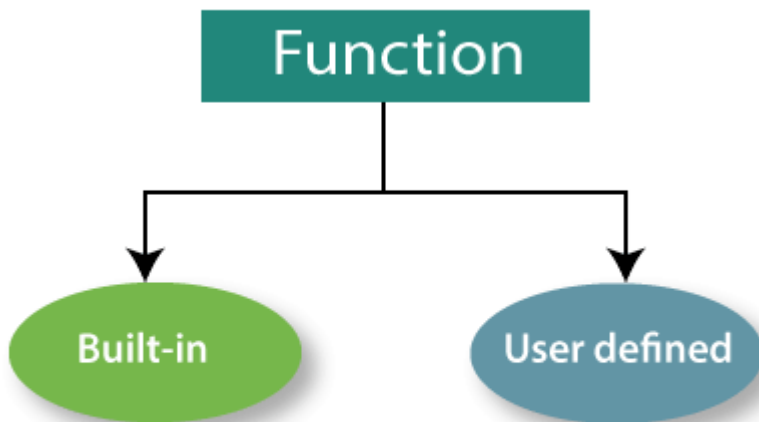
#### Return value

It is the last expression in the function body which is to be evaluated.

#### Function Types

## R-PROGRAMMING(DS3101PC/AM3101PC)

Similar to the other languages, R also has two types of function, i.e. Built-in Function and User-defined Function. In R, there are lots of built-in functions which we can directly call in the program without defining them. R also allows us to create our own functions.



### **Built-in function**

The functions which are already created or defined in the programming framework are known as built-in functions. User doesn't need to create these types of functions, and these functions are built into an application. End-users can access these functions by simply calling it. R have different types of built-in functions such as `seq()`, `mean()`, `max()`, and `sum(x)` etc.

```
# Creating sequence of numbers from 32 to 46.
```

```
print(seq(32,46))
```

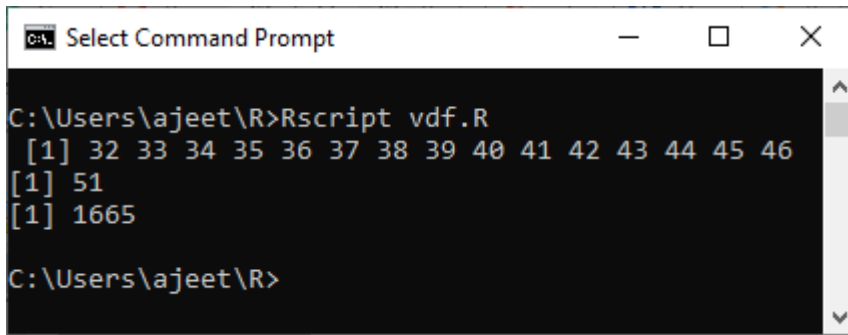
```
# Finding the mean of numbers from 22 to 80.
```

```
print(mean(22:80))
```

```
# Finding the sum of numbers from 41 to 70.
```

```
print(sum(41:70))
```

Output:



```
C:\Users\ajeet\R>Rscript vdf.R
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
[1] 51
[1] 1665
C:\Users\ajeet\R>
```

### User-defined function

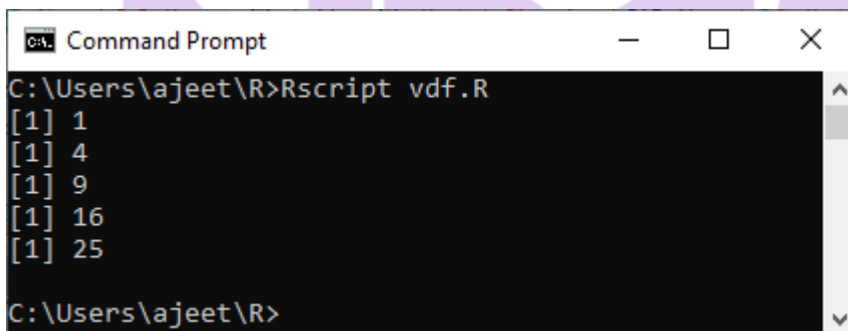
R allows us to create our own function in our program. A user defines a user-defined function to fulfill the requirement of user. Once these functions are created, we can use these functions like in-built function.

# Creating a function without an argument.

```
new.function <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
```

new.function()

Output:



```
C:\Users\ajeet\R>Rscript vdf.R
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
C:\Users\ajeet\R>
```

### Function calling with an argument

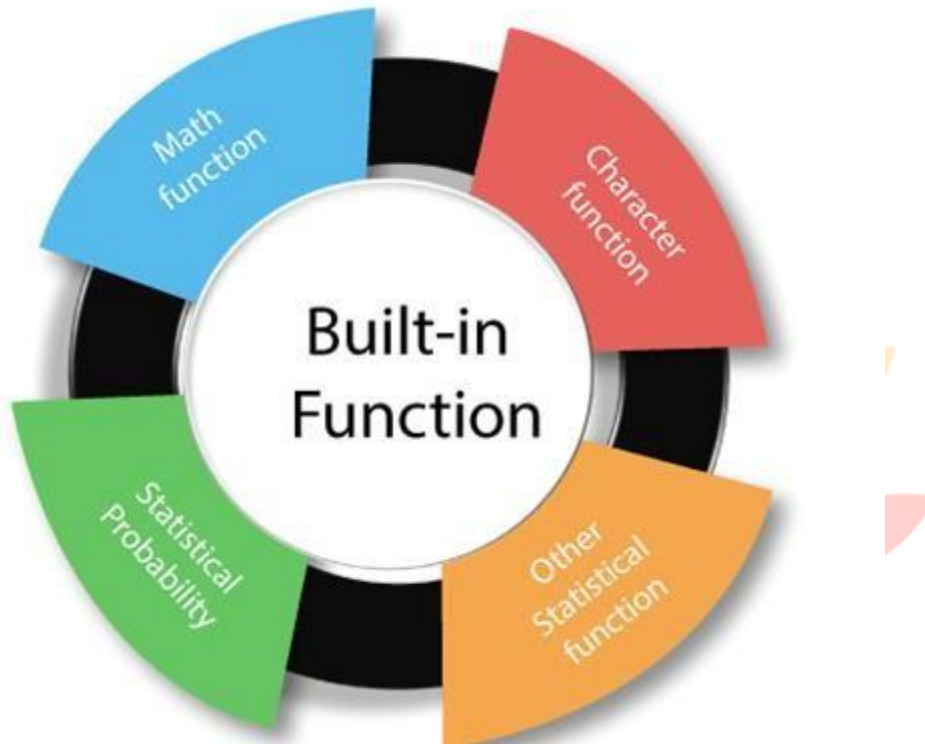
We can easily call a function by passing an appropriate argument in the function. Let see an example to see how a function is called.

# Creating a function to print squares of numbers in sequence.

R Built-in Functions

## R-PROGRAMMING(DS3101PC/AM3101PC)

The functions which are already created or defined in the programming framework are known as a built-in function. R has a rich set of functions that can be used to perform almost every task for the user. These built-in functions are divided into the following categories based on their functionality.



### Math Functions

R provides the various mathematical functions to perform the mathematical calculation. These mathematical functions are very

helpful to find absolute value, square value and much more calculations. In R, there are the following functions which are used:

S. No	Function	Description	Example
1.	abs(x)	It returns the absolute value of input x.	<pre>x&lt;- -4 print(abs(x))</pre> <p>Output [1] 4</p>
2.	sqrt(x)	It returns the square root of input x.	<pre>x&lt;- 4</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

			<pre>print(sqrt(x))</pre> <p>Output</p> <pre>[1] 2</pre>
3.	ceiling(x)	It returns the smallest integer which is larger than or equal to x.	<pre>x&lt;- 4.5</pre> <pre>print(ceiling(x))</pre> <p>Output</p> <pre>[1] 5</pre>
4.	floor(x)	It returns the largest integer, which is smaller than or equal to x.	<pre>x&lt;- 2.5</pre> <pre>print(floor(x))</pre> <p>Output</p> <pre>[1] 2</pre>
5.	trunc(x)	It returns the truncate value of input x.	<pre>x&lt;- c(1.2,2.5,8.1)</pre> <pre>print(trunc(x))</pre> <p>Output</p> <pre>[1] 1 2 8</pre>
6.	round(x, digits=n)	It returns round value of input x.	<pre>x&lt;- -4</pre> <pre>print(abs(x))</pre> <p>Output</p> <pre>4</pre>
7.	cos(x), sin(x), tan(x)	It returns cos(x), sin(x) value of input x.	<pre>x&lt;- 4</pre> <pre>print(cos(x))</pre> <pre>print(sin(x))</pre> <pre>print(tan(x))</pre> <p>Output</p>



## R-PROGRAMMING(DS3101PC/AM3101PC)

			<pre>[1] -06536436 [2] -0.7568025 [3] 1.157821</pre>
8.	log(x)	It returns natural logarithm of input x.	<pre>x&lt;- 4 print(log(x)) Output [1] 1.386294</pre>
9.	log10(x)	It returns common logarithm of input x.	<pre>x&lt;- 4 print(log10(x)) Output [1] 0.60206</pre>
10.	exp(x)	It returns exponent.	<pre>x&lt;- 4 print(exp(x)) Output [1] 54.59815</pre>

### String Function

R provides various string functions to perform tasks. These string functions allow us to extract sub string from string, search pattern etc.

There are the following string functions in R:

S. No	Function	Description	Example
1.	substr(x, start=n1, stop=n2)	It is used to extract substrings in a character vector.	<pre>a &lt;- "987654321" substr(a, 3, 3) Output</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

			[1] "3"
2.	<code>grep(pattern, x , ignore.case=FALSE, fixed=FALSE)</code>	It searches for pattern in x.	<pre>st1 &lt;- c('abcd','bdcd','abcdabcd') pattern &lt;- '^abc' print(grep(pattern, st1))</pre> <p>Output</p> <p>[1] 1 3</p>
3.	<code>sub(pattern, replacement, x, ignore.case =FALSE, fixed=FALSE)</code>	It finds pattern in x and replaces it with replacement (new) text.	<pre>st1 &lt;- "England is beautiful but no the part of EU" sub("England", "UK", st1)</pre> <p>Output</p> <p>[1] "UK is beautiful but not a part of EU"</p>
4.	<code>paste(..., sep="")</code>	It concatenates strings after using sep string to separate them.	<pre>paste('one',2,'three',4,'five')</pre> <p>Output</p> <p>[1] one 2 three 4 five</p>
5.	<code>strsplit(x, split)</code>	It splits the elements of character vector x at split point.	<pre>a &lt;- "Split all the character" print(strsplit(a, ""))</pre> <p>Output</p> <p>[[1]]</p> <p>[1] "split" "all" "the" "character"</p>
6.	<code>tolower(x)</code>	It is used to convert the string into lower case.	<pre>st1 &lt;- "shuBHAm" print(tolower(st1))</pre> <p>Output</p> <p>[1] shubham</p>

## R-PROGRAMMING(DS3101PC/AM3101PC)

7.	<code>toupper(x)</code>	It is used to convert the string into upper case.	<pre>st1&lt;- "shuBHAm" print(toupper(st1))</pre> <p>Output</p> <p>[1] SHUBHAM</p>
----	-------------------------	---	--

### Statistical Probability Functions

R provides various statistical probability functions to perform statistical task. These statistical functions are very helpful to

find normal density,

normal quantile and many more calculation. In R, there are following functions which are used:

795.3K

One in three children in the UK lie about their age on social media, Ofcom says

S. No	Function	Description	Example
1.	<code>dnorm(x, m=0, sd=1, log=False)</code>	It is used to find the height of the probability distribution at each point to a given mean and standard deviation	<pre>a &lt;- seq(-7, 7, by=0.1) b &lt;- dnorm(a, mean=2.5, sd=0.5) png(file="dnorm.png") plot(x,y) dev.off()</pre>
2.	<code>pnorm(q, m=0, sd=1, lower.tail=TRUE, log.p=FALSE)</code>	it is used to find the probability of a normally distributed random numbers which are less than the value of a given number.	<pre>a &lt;- seq(-7, 7, by=0.2) b &lt;- dnorm(a, mean=2.5, sd=2) png(file="pnorm.png") plot(x,y) dev.off()</pre>

### R-PROGRAMMING(DS3101PC/AM3101PC)

3.	qnorm(p, m=0, sd=1)	It is used to find a number whose cumulative value matches with the probability value.	<pre>a &lt;- seq(1, 2, by=0.02) b &lt;- qnorm(a, mean=2.5, sd=0.5) png(file="qnorm.png") plot(x,y) dev.off()</pre>
4.	rnorm(n, m=0, sd=1)	It is used to generate random numbers whose distribution is normal.	<pre>y &lt;- rnorm(40) png(file="rnorm.png") hist(y, main="Normal Distribution") dev.off()</pre>
5.	dbinom(x, size, prob)	It is used to find the probability density distribution at each point.	<pre>a&lt;-seq(0, 40, by=1) b&lt;- dbinom(a, 40, 0.5) png(file="pnorm.png") plot(x,y) dev.off()</pre>
6.	pbinom(q, size, prob)	It is used to find the cumulative probability (a single value representing the probability) of an event.	<pre>a &lt;- pbinom(25, 40,0.5) print(a) Output [1] 0.9596548</pre>
7.	qbinom(p, size, prob)	It is used to find a number whose cumulative value matches the probability value.	<pre>a &lt;- qbinom(0.25, 40,0.5) print(a) Output [1] 18</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

8.	<code>rbinom(n, size, prob)</code>	It is used to generate required number of random values of a given probability from a given sample.	<pre>a &lt;- rbinom(6, 140,0.4)  print(a)  Output  [1] 55 61 46 56 58 49</pre>
9.	<code>dpois(x, lamba)</code>	it is the probability of x successes in a period when the expected number of events is lambda ( $\lambda$ )	<pre>dpois(a=2, lambda=3)+dpois(a=3, lambda=3)+dpois(z=4, labda=4)  Output  [1] 0.616115</pre>
10.	<code>ppois(q, lamba)</code>	It is a cumulative probability of less than or equal to q successes.	<pre>ppois(q=4, lambda=3, lower.tail=TRUE)- ppois(q=1, lambda=3, lower.tail=TRUE)  Output  [1] 0.6434504</pre>
11.	<code>rpois(n, lamba)</code>	It is used to generate random numbers from the poisson distribution.	<pre>rpois(10, 10)  [1] 6 10 11 3 10 7 7 8 14 12</pre>
12.	<code>dunif(x, min=0, max=1)</code>	This function provide information about the uniform distribution on the interval from min to max. It gives the density.	<pre>dunif(x, min=0, max=1, log=FALSE)</pre>
13.	<code>punif(q, min=0, max=1)</code>	It gives the distributed function	<pre>punif(q, min=0, max=1, lower.tail=TRUE, log.p=FALSE)</pre>
14.	<code>qunif(p, min=0,</code>	It gives the quantile function.	<pre>qunif(p, min=0,</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

	max=1)		max=1, lower.tail=TRUE, log.p=FALSE)
15.	runif(x, min=0, max=1)	It generates random deviates.	runif(x, min=0, max=1)

### Other Statistical Function

Apart from the functions mentioned above, there are some other useful functions which helps for statistical purpose.

There are the following functions:

S. No	Function	Description	Example
1.	mean(x, trim=0, na.rm=FALSE)	It is used to find the mean for x object	a<-c(0:10, 40)  xm<-mean(a)  print(xm)  Output [1] 7.916667
2.	sd(x)	It returns standard deviation of an object.	a<-c(0:10, 40)  xm<-sd(a)  print(xm)  Output [1] 10.58694
3.	median(x)	It returns median.	a<-c(0:10, 40)  xm<-meadian(a)  print(xm)  Output

## R-PROGRAMMING(DS3101PC/AM3101PC)

			[1] 5.5
4.	quantile(x, probs)	It returns quantile where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0, 1]	
5.	range(x)	It returns range.	<pre>a&lt;-c(0:10, 40) xm&lt;-range(a) print(xm) Output [1] 0 40</pre>
6.	sum(x)	It returns sum.	<pre>a&lt;-c(0:10, 40) xm&lt;-sum(a) print(xm) Output [1] 95</pre>
7.	diff(x, lag=1)	It returns differences with lag indicating which lag to use.	<pre>a&lt;-c(0:10, 40) xm&lt;-diff(a) print(xm) Output [1] 1 1 1 1 1 1 1 1 1 1 30</pre>
8.	min(x)	It returns minimum value.	<pre>a&lt;-c(0:10, 40) xm&lt;-min(a) print(xm) Output [1] 0</pre>

## R-PROGRAMMING(DS3101PC/AM3101PC)

9.	<code>max(x)</code>	It returns maximum value	<pre>a&lt;-c(0:10, 40) xm&lt;-max(a) print(xm) Output [1] 40</pre>
10.	<code>scale(x, center=TRUE, scale=TRUE)</code>	Column center or standardize a matrix.	<pre>a &lt;- matrix(1:9,3,3) scale(x) Output [,1] [1,] -0.747776547 [2,] -0.653320562 [3,] -0.558864577 [4,] -0.464408592 [5,] -0.369952608 [6,] -0.275496623 [7,] -0.181040638 [8,] -0.086584653 [9,] 0.007871332 [10,] 0.102327317 [11,] 0.196783302 [12,] 3.030462849 attr("scaled:center") [1] 7.916667 attr("scaled:scale") [1] 10.58694</pre>

`new.function <- function(a) {`



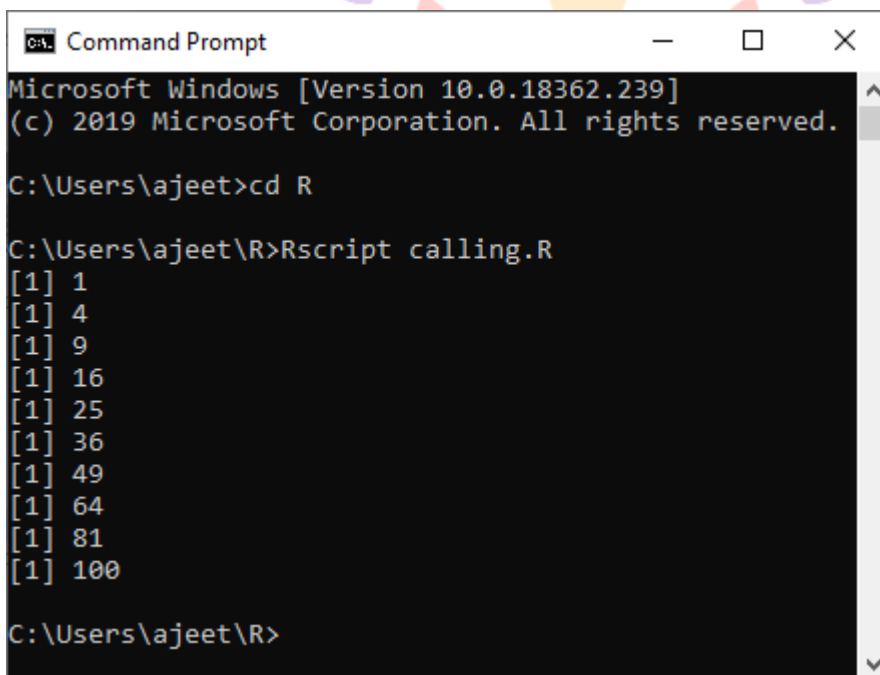
## R-PROGRAMMING(DS3101PC/AM3101PC)

```
for(i in 1:a) {  
  b <- i^2  
  print(b)  
}
```

# Calling the function new.function supplying 10 as an argument.

```
new.function(10)
```

Output:



```
Command Prompt  
Microsoft Windows [Version 10.0.18362.239]  
(c) 2019 Microsoft Corporation. All rights reserved.  
C:\Users\ajeet>cd R  
C:\Users\ajeet\R>Rscript calling.R  
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25  
[1] 36  
[1] 49  
[1] 64  
[1] 81  
[1] 100  
C:\Users\ajeet\R>
```

### Function calling with no argument

In R, we can call a function without an argument in the following way

# Creating a function to print squares of numbers in sequence.

```
new.function <- function() {
```

```
  for(i in 1:5) {
```

```
    a <- i^2
```

```
    print(a)
```

```
  }
```

```
}
```

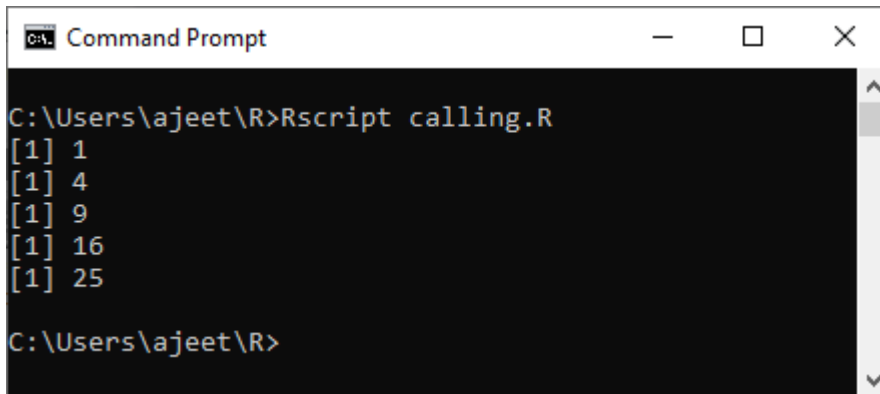
your roots to success...

## R-PROGRAMMING(DS3101PC/AM3101PC)

# Calling the function new.function with no argument.

```
new.function()
```

Output:



```
Command Prompt
C:\Users\ajeet\R>Rscript calling.R
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
C:\Users\ajeet\R>
```

### Function calling with Argument Values

We can supply the arguments to a function call in the same sequence as defined in the function or can supply in a different sequence but assigned them to the names of the arguments.

# Creating a function with arguments.

```
new.function <- function(x,y,z) {
  result <- x * y + z
  print(result)
}
```

# Calling the function by position of arguments.

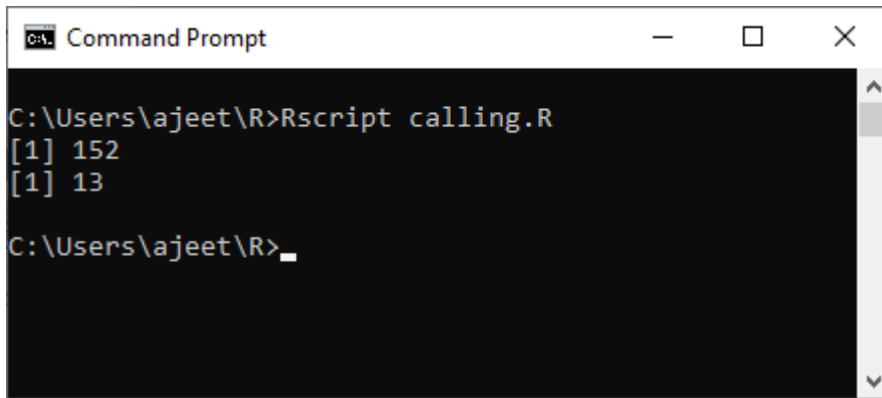
```
new.function(11,13,9)
```

# Calling the function by names of the arguments.

```
new.function(x = 2, y = 5, z = 3)
```

Output:

your roots to success...



```
Command Prompt
C:\Users\ajeet\R>Rscript calling.R
[1] 152
[1] 13
C:\Users\ajeet\R>_
```

### Function calling with default arguments

To get the default result, we assign the value to the arguments in the function definition, and then we call the function without supplying argument. If we pass any argument in the function call, then it will get replaced with the default value of the argument in the function definition.

# Creating a function with arguments.

```
new.function <- function(x = 11, y = 24) {
  result <- x * y
  print(result)
}
```

# Calling the function without giving any argument.

```
new.function()
```

# Calling the function with giving new values of the argument.

```
new.function(4,6)
```

Output:

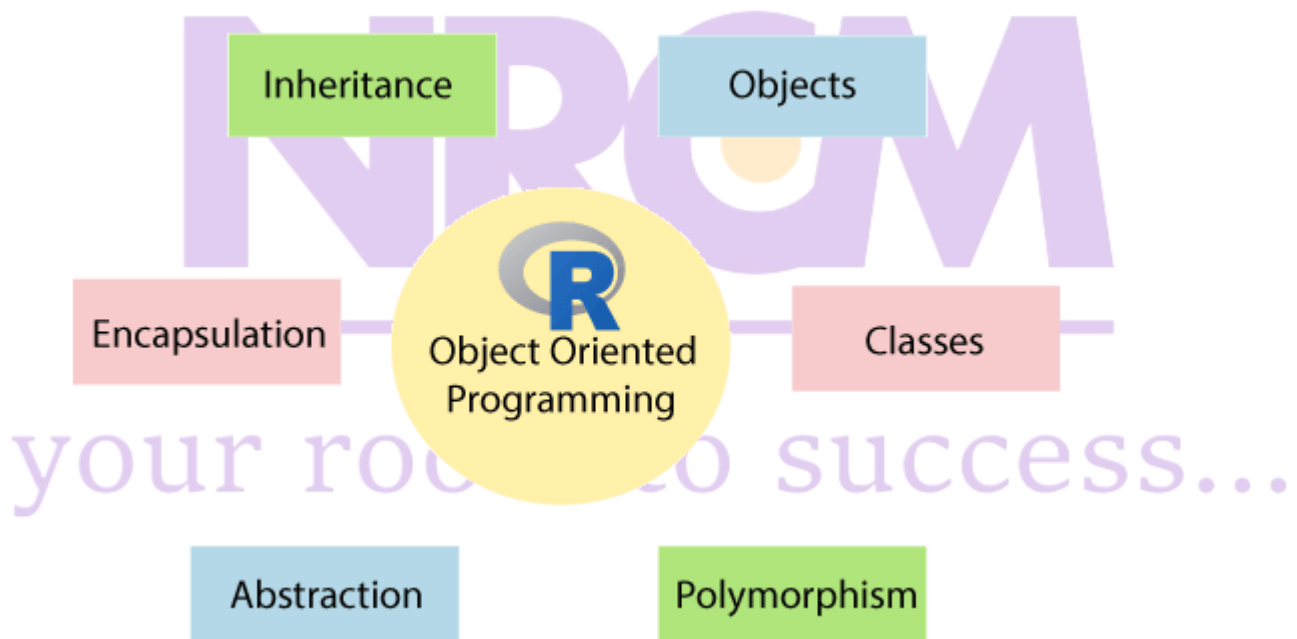
your roots to success...

```
Command Prompt
C:\Users\ajeet\R>Rscript calling.R
[1] 264
[1] 24
C:\Users\ajeet\R>_
```

## UNIT-V

### What is Object-Oriented Programming in R?

Object-Oriented Programming (OOP) is the most popular programming language. With the help of oops concepts, we can construct the modular pieces of code which are used to build blocks for large systems. R is a functional language, and we can do programming in oops style. In R, oops is a great tool to manage the complexity of larger programs.



In Object-Oriented Programming, S3 and S4 are the two important systems.

### S3

In oops, the S3 is used to overload any function. So that we can call the functions with different names and it depends on the type of input parameter or the number of parameters.

#### Play Videox

### S4

S4 is the most important characteristic of oops. However, this is a limitation, as it is quite difficult to debug. There is an optional reference class for S4.

#### Objects and Classes in R

In R, everything is an object. Therefore, programmers perform OOPS concept when they write code in R. An object is a data structure which has some methods that can act upon its attributes.

In R, classes are the outline or design for the object. Classes encapsulate the data members, along with the functions. In R, there are two most important classes, i.e., S3 and S4, which play an important role in performing OOPs concepts.

Let's discuss both the classes one by one with their examples for better understanding.

#### 1) S3 Class

With the help of the S3 class, we can take advantage of the ability to implement the generic function OO. Furthermore, using only the first argument, S3 is capable of dispatching. S3 differs from traditional programming languages such as Java, C ++, and C #, which implement OO passing messages. This makes S3 easy to implement. In the S3 class, the generic function calls the method. S3 is very casual and has no formal definition of classes.

**S3 requires very little knowledge from the programmer.**

#### Creating an S3 class

In R, we define a function which will create a class and return the object of the created class. A list is made with relevant members, class of the list is determined, and a copy of the list is returned. There is the following syntax to create a class

```
variable_name <- list(member1, member2, member3. ....memberN)
```

Example

```
s <- list(name = "Ram", age = 29, GPA = 4.0)
```

```
class(s) <- "Faculty"
```

```
s
```

Output

```

Command Prompt
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet>cd R

C:\Users\ajeet\R>Rscript class.R
$name
[1] "Ram"

$sage
[1] 29

$GPA
[1] 4

attr(,"class")
[1] "student"

C:\Users\ajeet\R>
    
```

There is the following way in which we define our generic function print.

print

function(x, ...)

UseMethod("Print")

When we execute or run the above code, it will give us the following output:

```

Rterm (64-bit)
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]

> print
function (x, ...)
  UseMethod("print")
<bytecode: 0x0000000014d3c578>
<environment: namespace:base>
> function (x, ...)
+ UseMethod("print")
function (x, ...)
  UseMethod("print")
>
    
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

Like print function, we will make a generic function GPA to assign a new value to our GPA member. In the following way we will make the generic function GPA

```
GPA <- function(obj1){  
  UseMethod("GPA")  
}
```

Once our generic function GPA is created, we will implement a default function for it

```
GPA.default <- function(obj){  
  cat("We are entering in generic function\n")  
}
```

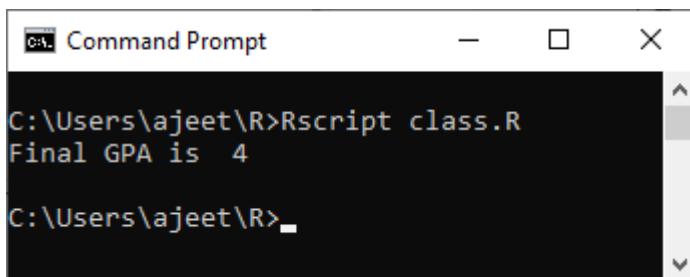
After that we will make a new method for our GPA function in the following way

```
GPA.faculty <- function(obj1){  
  cat("Final GPA is ",obj1$GPA,"\n")  
}
```

And at last we will run the method GPA as

GPA(s)

Output



```
Command Prompt  
C:\Users\ajet\R>Rscript class.R  
Final GPA is 4  
C:\Users\ajet\R>_
```

### Inheritance in S3

Inheritance means extracting the features of one class into another class. In the S3 class of R, inheritance is achieved by applying the class attribute in a vector.

For inheritance, we first create a function which creates new object of class faculty in the following way

```
faculty<- function(n,a,g) {  
  value <- list(nname=n, aage=a, GPA=g)
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

```
attr(value, "class") <- "faculty"
```

```
value
```

```
}
```

After that we will define a method for generic function print() as

```
print.student <- function(obj1) {
```

```
cat(1obj$name, "\n")
```

```
cat(1obj$age, "years old\n")
```

```
cat("GPA:", obj1$GPA, "\n")
```

```
}
```

Now, we will create an object of class InternationalFaculty which will inherit from faculty class. This process will be done by assigning a character vector of class name as:

```
class(Objet) <- c(child, parent)
```

so,

```
# create a list
```

```
fac <- list(name="Shubham", age=22, GPA=3.5, country="India")
```

```
# make it of the class InternationalFaculty which is derived from the class Faculty
```

```
class(fac) <- c("InternationalFaculty","Faculty")
```

```
# print it out
```

```
fac
```

When we run the above code which we have discussed, it will generate the following output:

your roots to success...



```
Command Prompt
C:\Users\ajeet\R>Rscript class.R
$name
[1] "Shubham"

$age
[1] 22

$GPA
[1] 3.5

$country
[1] "India"

attr(,"class")
[1] "InternationalFaculty" "Faculty"

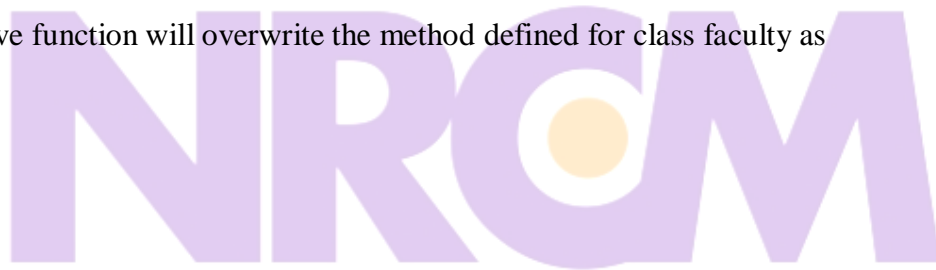
C:\Users\ajeet\R>
```

We can see above that, we have not defined any method of form `print.InternationalFaculty()`, the method called `print.Faculty()`. This method of class `Faculty` was inherited.

So our next step is to defined `print.InternationalFaculty()` in the following way:

```
print.InternationalFaculty<- function(obj1) {
cat(obj1$name, "is from", obj1$country, "\n")
}
```

The above function will overwrite the method defined for class `faculty` as



your roots to success...

Fac

```
Command Prompt
C:\Users\ajeet\R>Rscript class.R
$name
[1] "Shubham"

$age
[1] 22

$GPA
[1] 3.5

$country
[1] "India"

attr(,"class")
[1] "InternationalFaculty" "Faculty"
Shubham is from India

C:\Users\ajeet\R>
```

### getS3method and getAnywhere function

There are the two most common and popular S3 method functions which are used in R. The first method is getS3method() and the second one is getAnywhere().

S3 finds the appropriate method associated with a class, and it is useful to see how a method is implemented. Sometimes, the methods are non-visible, because they are hidden in a namespace. We use getS3method or getAnywhere to solve this problem.

The logo for NRCM (National Research Centre for Mathematics) features the letters 'NRCM' in a bold, purple, sans-serif font. The letter 'O' is stylized with a yellow circle in the center. The logo is flanked by two horizontal purple lines, one above and one below.

your roots to success...

getS3method

```

Rterm (64-bit)
> exists("predict.ppr")
[1] FALSE
> getS3method("predict","ppr")
function (object, newdata, ...)
{
  if (missing(newdata))
    return(fitted(object))
  if (!is.null(object$terms)) {
    newdata <- as.data.frame(newdata)
    rn <- row.names(newdata)
    Terms <- delete.response(object$terms)
    m <- model.frame(Terms, newdata, na.action = na.omit,
      xlev = object$xlevels)
    if (!is.null(cl <- attr(Terms, "dataClasses")))
      .checkMFClasses(cl, m)
    keep <- match(row.names(m), rn)
    x <- model.matrix(Terms, m, contrasts.arg = object$contrasts)
  }
  else {
    x <- as.matrix(newdata)
    keep <- seq_len(nrow(x))
    rn <- dimnames(x)[[1L]]
  }
  if (ncol(x) != object$p)
    stop("wrong number of columns in 'x'")
  res <- matrix(NA, length(keep), object$q, dimnames = list(rn,
    object$ynames))
  res[keep, ] <- matrix(.Fortran(C_pppred, as.integer(nrow(x)),
    as.double(x), as.double(object$smod), y = double(nrow(x)) *
      object$q), double(2 * object$smod[4L]))$y, ncol = object$q)
  drop(res)
}
<bytecode: 0x0000000004ae3608>
<environment: namespace:stats>
>

```

## getAnywhere function

getAnywhere("simpleloess")

## 2) S4 Class

The S4 class is similar to the S3 but is more formal than the latter one. It differs from S3 in two different ways. First, in S4, there are formal class definitions which provide a description and representation of classes. In addition, it has special auxiliary functions for defining methods and generics. The S4 also offers multiple dispatches. This means that common functions are capable of taking methods based on multiple arguments which are based on class.

### Creating an S4 class

In R, we use setClass() command for creating S4 class. In S4 class, we will specify a function for verifying the data consistency and also specify the default value. In R, member variables are called slots.

## R-PROGRAMMING(DS3101PC/AM3101PC)

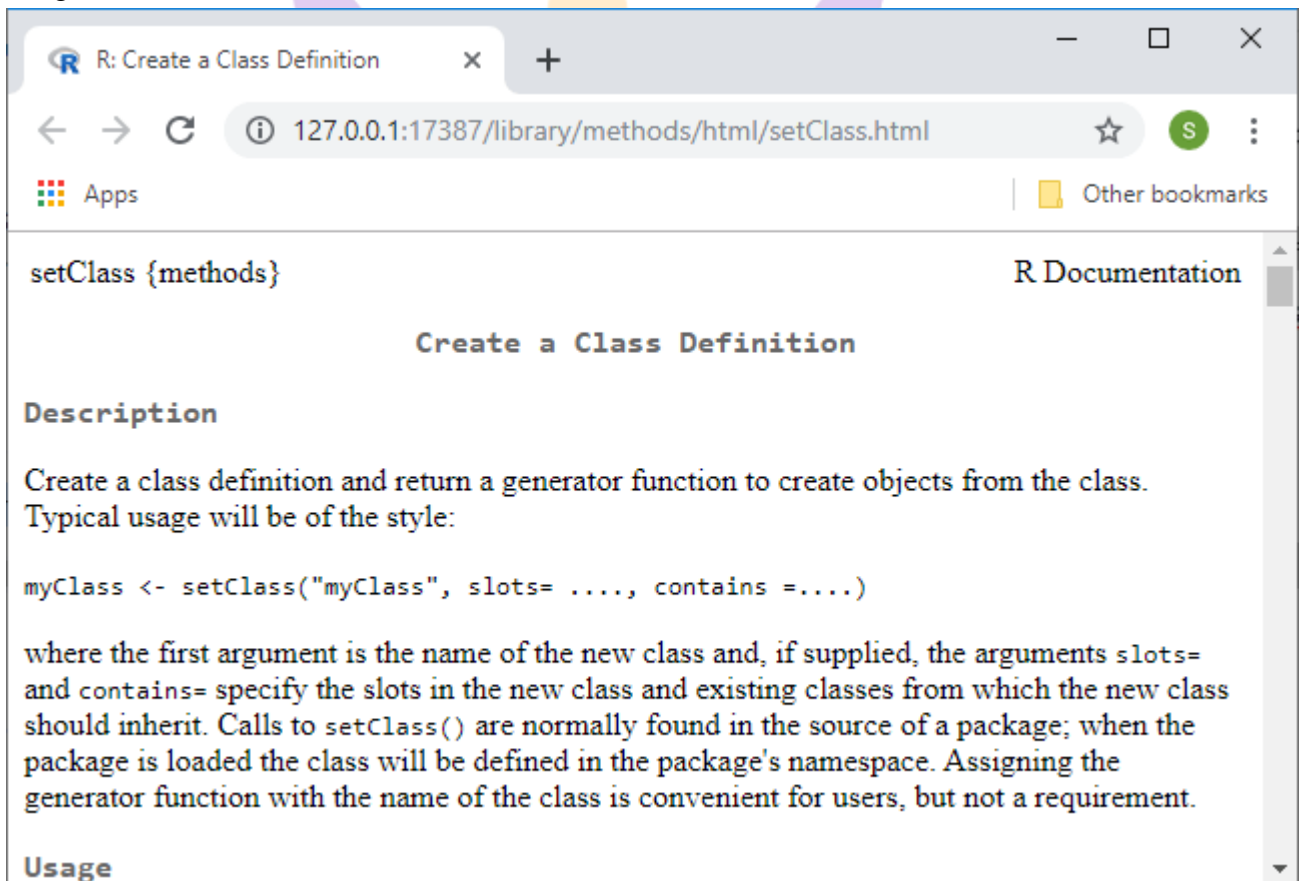
To create an S3 class, we have to define the class and its slots. There are the following steps to create an S4 class

### Step 1:

In the first step, we will create a new class called faculty with three slots name, age, and GPA.

```
setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
```

There are many other optional arguments of setClass() function which we can explore by using ?setClass command.



The screenshot shows a web browser window displaying the R Documentation for the setClass function. The browser tab is titled "R: Create a Class Definition" and the address bar shows "127.0.0.1:17387/library/methods/html/setClass.html". The page content includes the title "setClass {methods}" and "R Documentation". Below the title is the heading "Create a Class Definition". The "Description" section states: "Create a class definition and return a generator function to create objects from the class. Typical usage will be of the style:" followed by the code snippet: 

```
myClass <- setClass("myClass", slots= ....., contains =.....)
```

 The text explains that the first argument is the name of the new class, and slots= and contains= specify slots and existing classes for inheritance. The "Usage" section is partially visible at the bottom.

### Step 2:

In the next step, we will create the object of S4 class. R provides new() function to create an object of S4 class. In this new function we pass the class name and the values for the slots in the following way:

```
setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
```

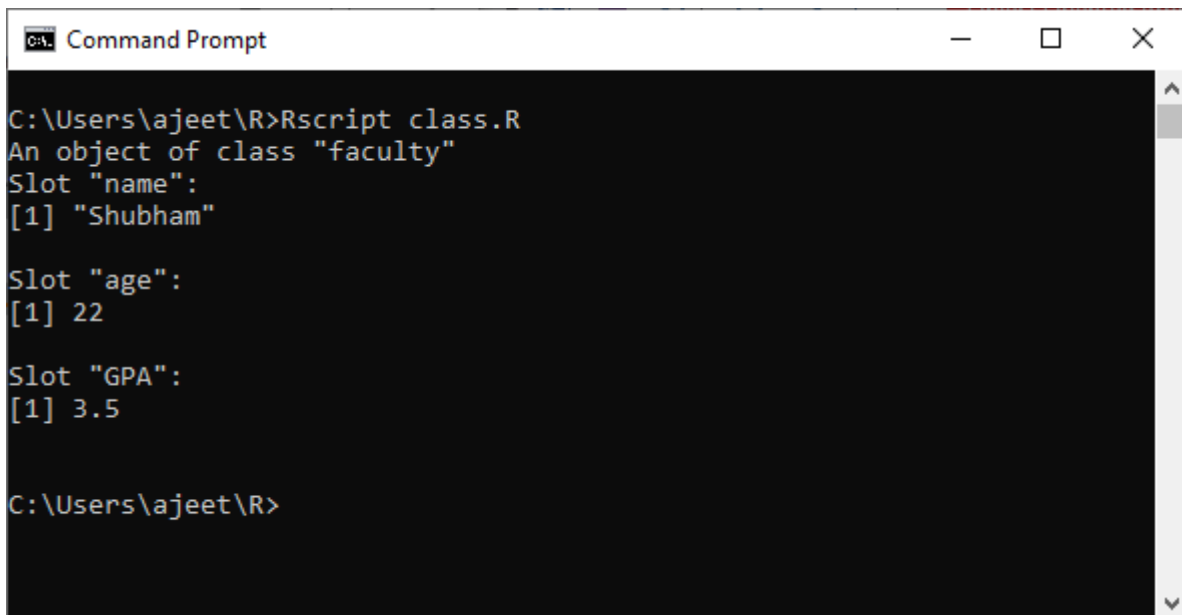
```
# creating an object using new()
```

```
# providing the class name and value for slots
```

```
s <- new("faculty", name="Shubham", age=22, GPA=3.5)
```

S

It will generate the following output



```
Command Prompt
C:\Users\ajeet\R>Rscript class.R
An object of class "faculty"
Slot "name":
[1] "Shubham"

Slot "age":
[1] 22

Slot "GPA":
[1] 3.5

C:\Users\ajeet\R>
```

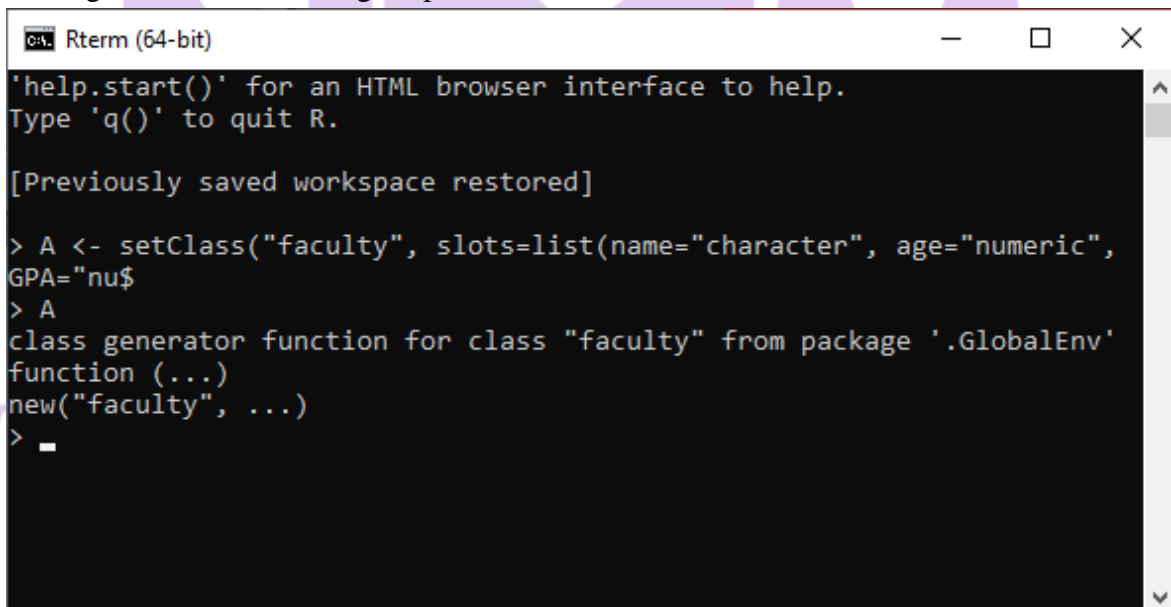
### Creating S4 objects using a generator function

The `setClass()` function returns a generator function. This generator function helps in creating new objects. And it acts as a constructor.

```
A <- setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
```

A

It will generate the following output:



```
Rterm (64-bit)
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> A <- setClass("faculty", slots=list(name="character", age="numeric",
GPA="nu$
> A
class generator function for class "faculty" from package '.GlobalEnv'
function (...)
new("faculty", ...)
> _
```

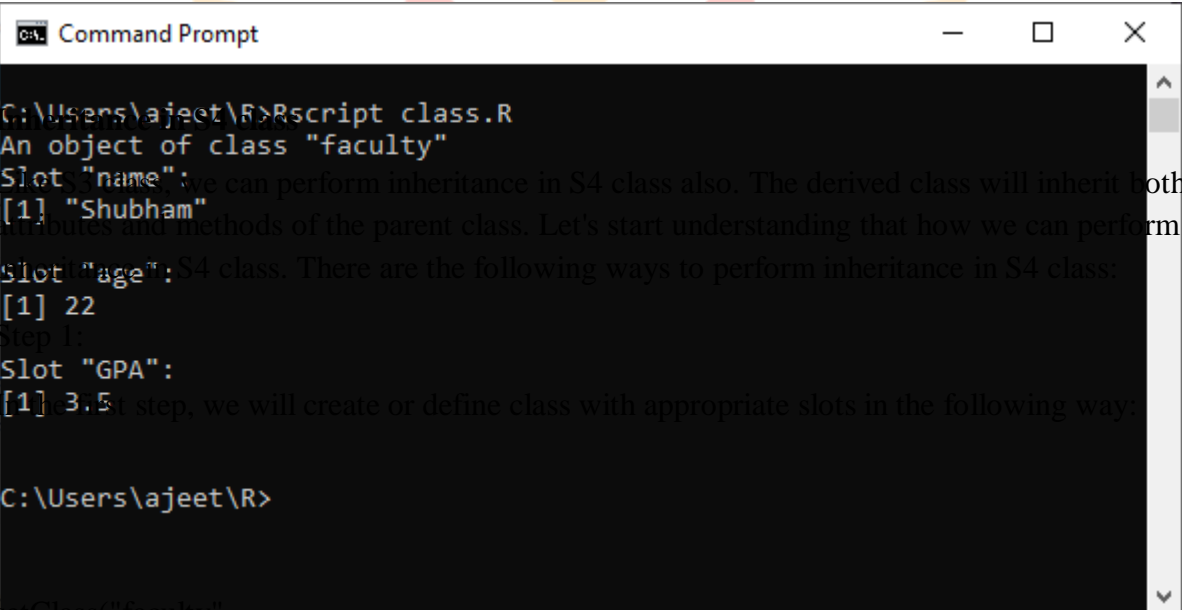
## R-PROGRAMMING(DS3101PC/AM3101PC)

Now we can use the above constructor function to create new objects. The constructor in turn uses the new() function to create objects. It is just a wrap around. Let's see an example to understand how S4 object is created with the help of generator function.

Example

```
faculty<-setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))  
  
# creating an object using generator() function  
  
# providing the class name and value for slots  
faculty(name="Shubham", age=22, GPA=3.5)
```

Output



```
Command Prompt  
C:\Users\ajeet\R>Rscript class.R  
An object of class "faculty"  
Slot "name":  
[1] "Shubham"  
Slot "age":  
[1] 22  
Slot "GPA":  
[1] 3.5  
C:\Users\ajeet\R>
```

```
setClass("faculty",  
slots=list(name="character", age="numeric", GPA="numeric")  
)
```

**NRCM**

your roots to success...

## R-PROGRAMMING(DS3101PC/AM3101PC)

### Step 2:

After defining class, our next step is to define class method for the display() generic function. This will be done in the following manner:

```
setMethod("show",  
"faculty",  
function(obj) {  
  cat(obj@name, "\n")  
  cat(obj@age, "years old\n")  
  cat("GPA:", obj@GPA, "\n")  
}  
)
```

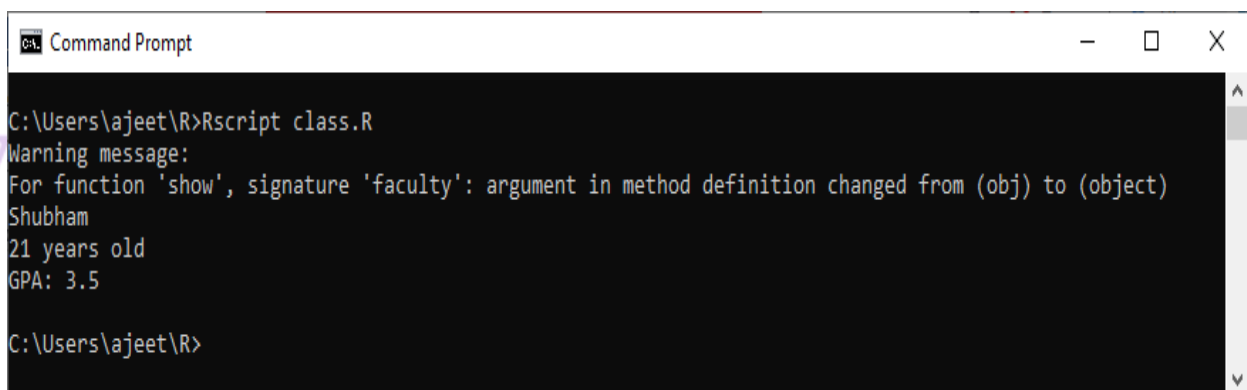
### Step 3:

In the next step, we will define the derived class with the argument contains. The derived class is defined in the following way

```
setClass("Internationalfaculty",  
slots=list(country="character"),  
contains="faculty"  
)
```

In our derived class we have defined only one attribute i.e. country. Other attributes will be inherited from its parent class.

```
s <- new("Internationalfaculty",name="John", age=21, GPA=3.5, country="India")  
show(s)
```



```
Command Prompt  
C:\Users\ajeet\R>Rscript class.R  
Warning message:  
For function 'show', signature 'faculty': argument in method definition changed from (obj) to (object)  
Shubham  
21 years old  
GPA: 3.5  
C:\Users\ajeet\R>
```

## R-PROGRAMMING(DS3101PC/AM3101PC)

When we did show(s), the method defines for class faculty gets called. We can also define methods for the derived class

of the base class as in the case of the S3 system.



your roots to success...