

R Programming Language – Introduction

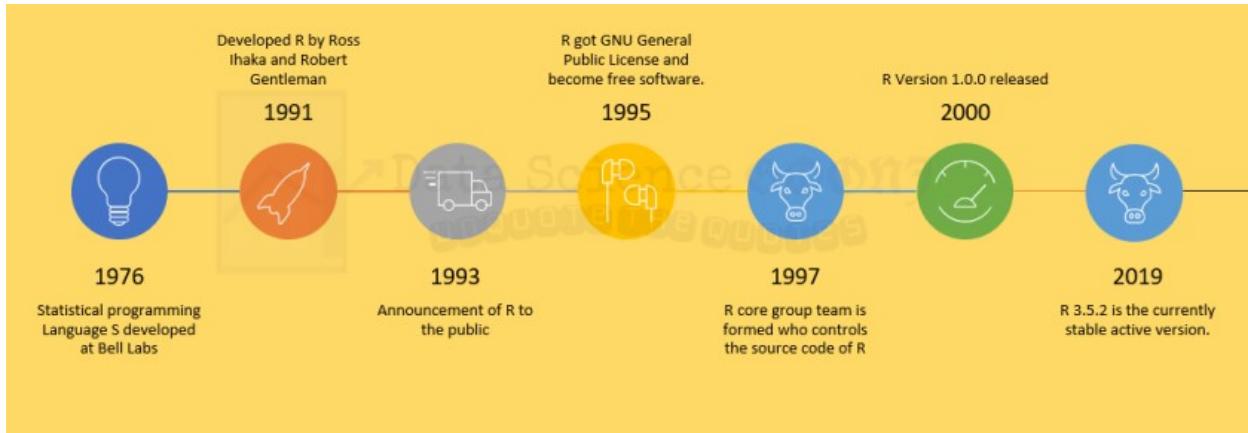
R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

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.

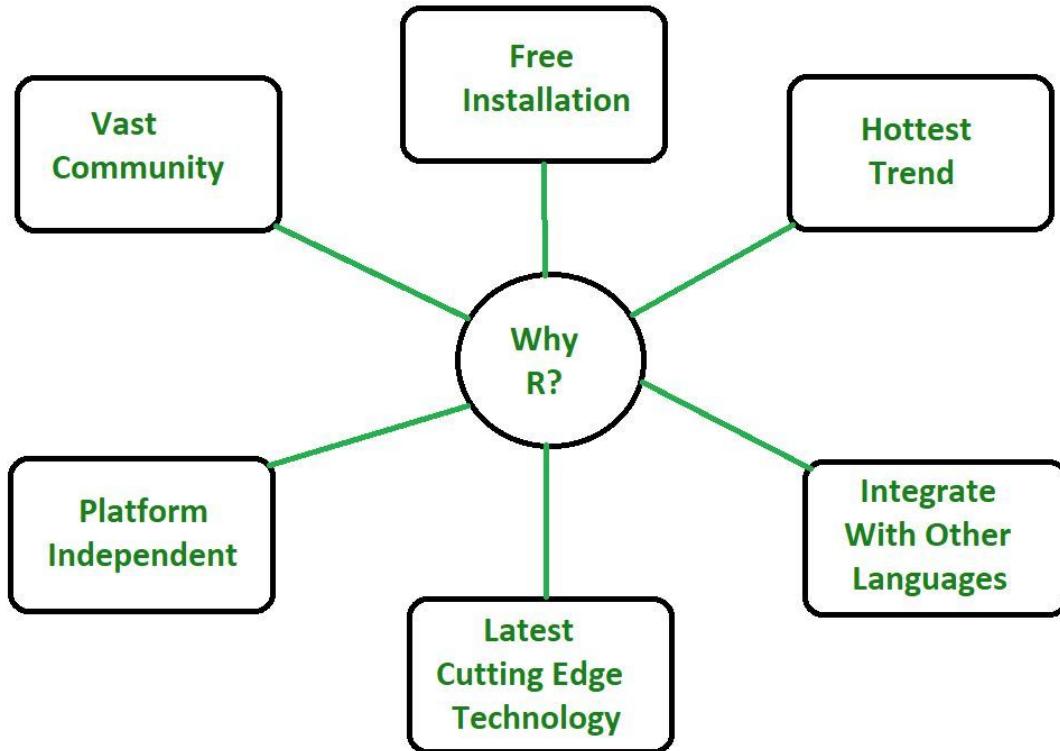
History of R

R is a dialect of S language. S language was developed by John Chambers, Rick Becker and others at the **Bell Labs** for which he got “**ACM Software System Award**” in 1998. He donated his prize money (US\$10,000) to the *American Statistical Association*. S language was started in 1976 as an internal Statistical analysis environment — implemented originally in Fortran Libraries. It was re-written in C language in 1988. S Language Version 4 was released in 1998 which is still in use. Following key notes written by John Chambers dictates the philosophy behind S language:

Key idea behind creation of S Language and later R language was to provide a language which is suitable for Interactive Data Analysis as well as for writing longer programs.



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 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

- 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`

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.

Interesting Facts about R Programming Language

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.

INTERESTING FACTS



Here are some interesting facts about the R programming language:

- R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. It is named partly after the first names of the first two R authors and partly as a play on the name of S.
- R supports both procedural programming and object-oriented programming. Procedural programming includes the procedure, records, modules, and procedure calls. While object-oriented programming language includes class, objects, and generic functions.
- R language is an interpreted language instead of a compiled language. Therefore, it doesn't need a compiler to compile code into an executable program. This makes running an R script much less time-consuming.
- The number of R packages available either through CRAN or GitHub is 1, 00, 000 and they do epic stuff with just one line of code. It could range from Regression to Bayesian analysis.
- R is growing faster than any other data science language. It's the most-used data science language after SQL. It is used by 70% of data miners.
- One of the packages in R namely rmarkdown package helps you create reproducible Word documents and reproducible Powerpoint Presentations from your R markdown code just by changing one line in the YAML! ("YAML Ain't Markup Language!")
- It is really very easy in R to connect to almost any database using the dbplyr package. This makes possible for an R user to work independently and pulling data from almost all common database types. You can also use packages like bigquery to work directly with BigQuery and other high-performance data stores.
- You can build and host interactive web apps in just a few lines of code in R. Using the flexdashboard package in R you can create interactive web apps with a few lines of code. And using the rsconnect package you can also host your web apps on your own server or, even easier, host them on a cloud server.

- You can not only deploy web apps but also can make them into awesome video games in R. The nessy package helps you create NES(The Nintendo Entertainment System) looking Shiny apps and deploy them just like you would any other Shiny app.
- You can build APIs and serve them from R. The plumber package in R helps you convert R functions to web APIs that can be integrated into downstream applications.
- According to [PYPL PopularitY of Programming Language](#) R is #7 of all programming languages. R is the #1 Google Search for Advanced Analytics software. It has more than 3 million users worldwide make a huge community for R programming language.
- The origin of R programming language can be traced back to 1993 when Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand introduced it.
- R is an open-source language and it is available for free for everyone to use for statistical and graphical purposes.
- The R programming language has a supportive and enthusiastic user community, providing ample resources and assistance to users.
- The widespread usage of R in fields such as data science, machine learning, and statistical modeling has made it one of the most sought-after programming languages.
- R has a wealth of packages and libraries, allowing users to perform complex tasks easily and extend its functionality.
- Industries such as finance, healthcare, pharmaceuticals, and marketing make use of R for data analysis and modeling.
- In academic research, R has become a crucial tool across various disciplines such as biology, psychology, and economics.
- R operates seamlessly on different platforms like Windows, macOS, and Linux, making it easily accessible to users regardless of the operating system they use.

Introduction to R Studio

R Studio is an integrated development environment(IDE) for R. IDE is a GUI, where you can write your quotes, see the results and also see the variables that are generated during the course of programming.

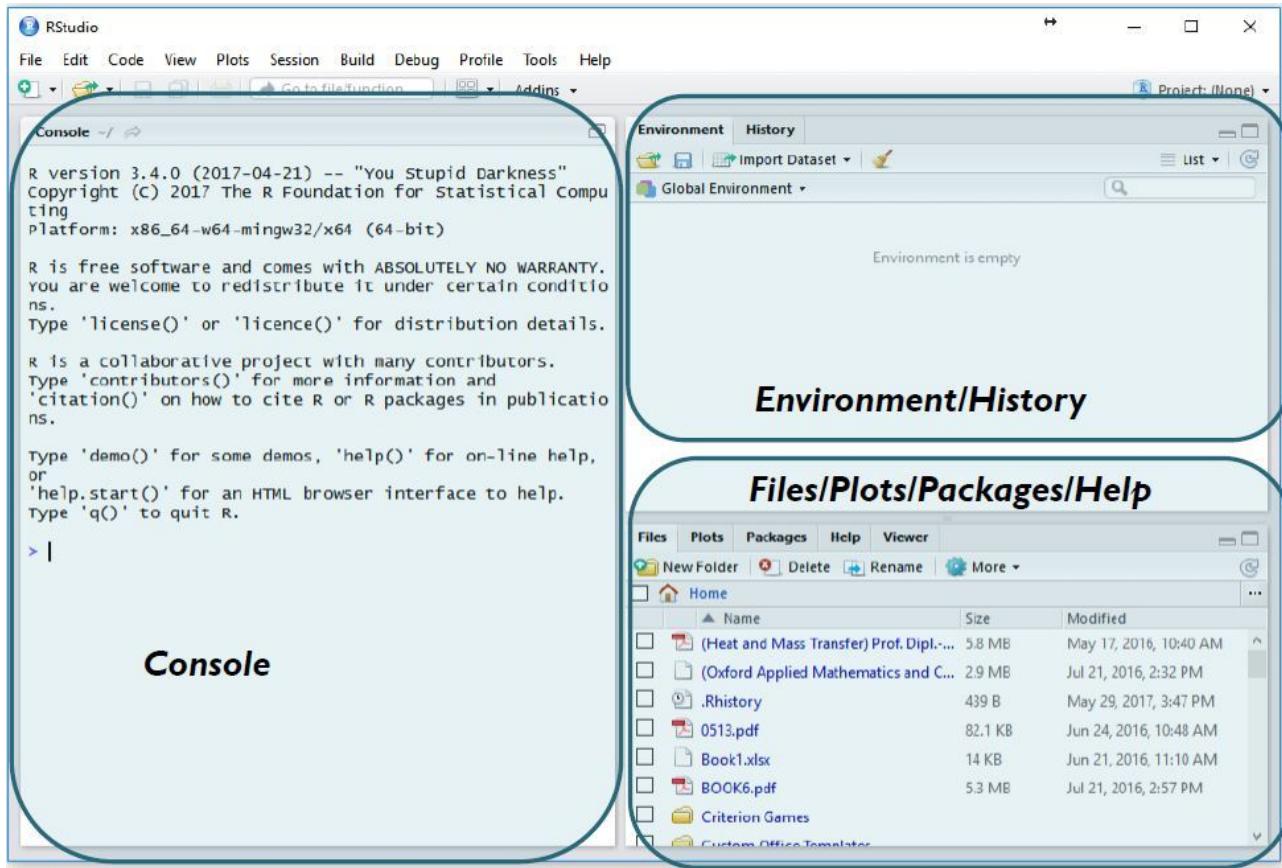
- R Studio is available as both Open source and Commercial software.
- R Studio is also available as both Desktop and Server versions.
- R Studio is also available for various platforms such as Windows, Linux, and macOS.

Introduction to R studio for beginners:

Rstudio is an open-source tool that provides IDE to use R language, and enterprise-ready professional software for data science teams to develop share the work with their team.

R Studio can be downloaded from its official Website (<https://rstudio.com/>) and instructions for installation are available on [How to Install RStudio for R programming in Windows?](#)

After the installation process is over, the R Studio interface looks like:



- The console panel(left panel) is the place where R is waiting for you to tell it what to do, and see the results that are generated when you type in the commands.
- To the top right, you have the Environmental/History panel. It contains 2 tabs:
 - **Environment tab:** It shows the variables that are generated during the course of programming in a workspace that is temporary.
 - **History tab:** In this tab, you'll see all the commands that are used till now from the start of usage of R Studio.
- To the right bottom, you have another panel, which contains multiple tabs, such as files, plots, packages, help, and viewer.

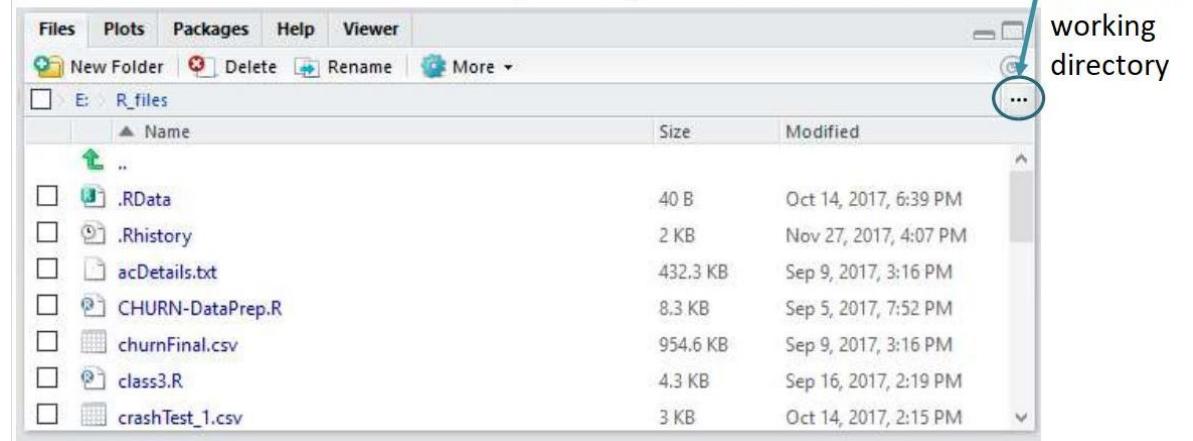
- The **Files tab** shows the files and directories that are available within the default workspace of R.
- The **Plots tab** shows the plots that are generated during the course of programming.
- The **Packages tab** helps you to look at what are the packages that are already installed in the R Studio and it also gives a user interface to install new packages.
- The **Help tab** is the most important one where you can get help from the R Documentation on the functions that are in built-in R.
- The final and last tab is that the **Viewer tab** which can be used to see the local web content that's generated using R.

Set the working directory in R Studio

R is always pointed at a directory on our computer. We can find out which directory by running the `getwd()` function. Note: this function has no arguments. We can set the working directory manually in two ways:

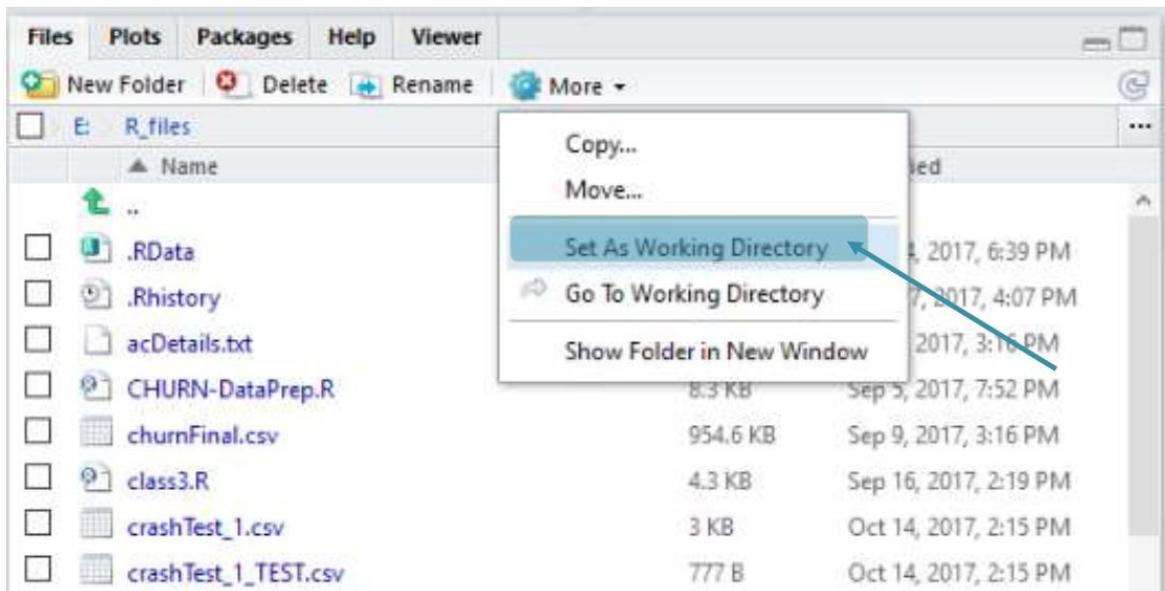
- **The first way is to use the console and using the command `setwd("directorypath")`.**
You can use this function `setwd()` and give the path of the directory which you want to be the working directory for R studio, in the double codes.
- **The second way is to set the working directory from the GUI.**
To set the working directory from the GUI you have to click on this 3 dots button. When you click this, this will open up a file browser, which will help you to choose your working directory.

STEP 1: Choose a suitable location by clicking on the indicated icon



- Once you choose your working directory, you need to use this setting button in the more tab and click it and then you get a popup menu, where you need to select “Set as working directory”.

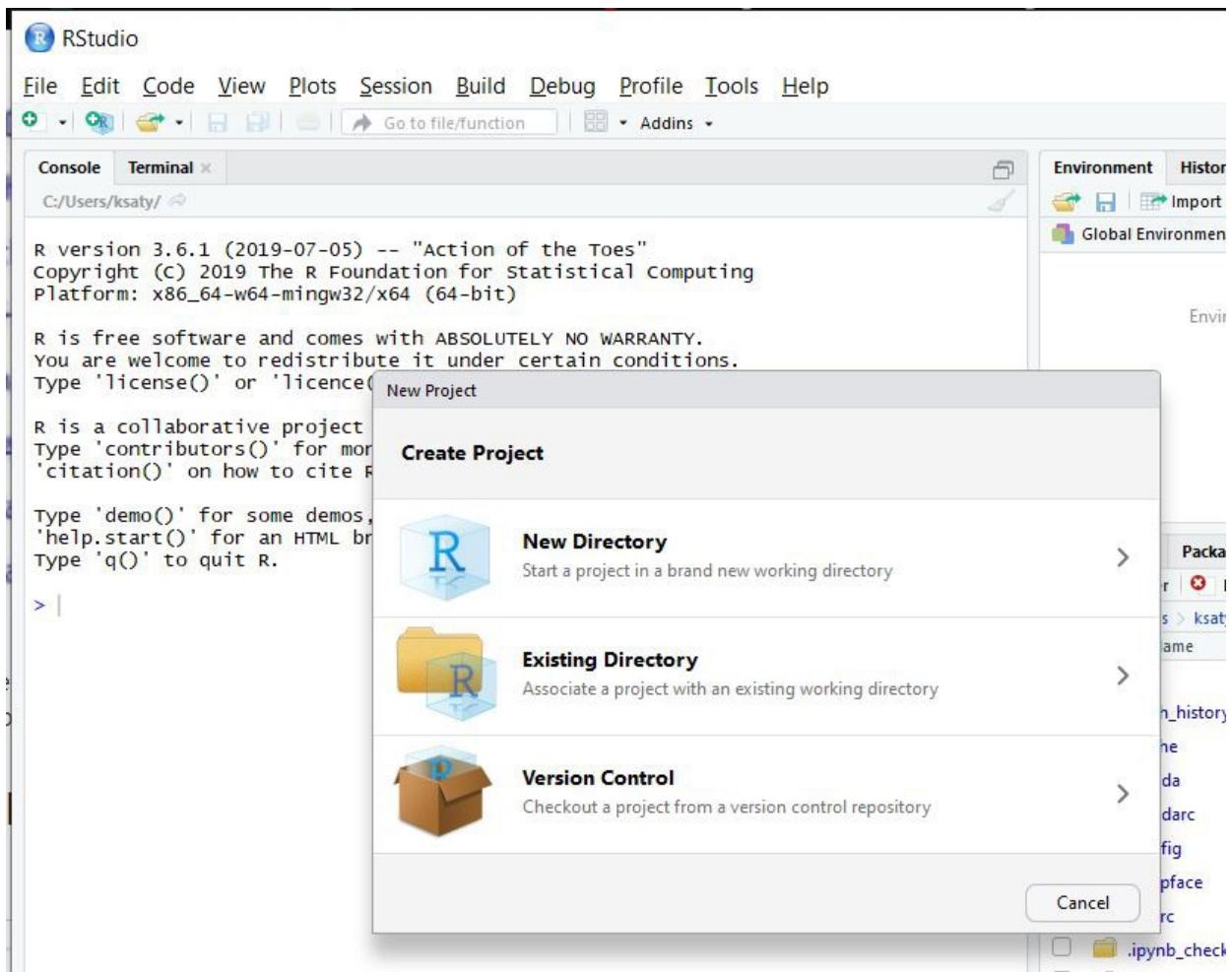
STEP 2: Once directory is chosen, select the more icon and choose “Set as Working Directory”



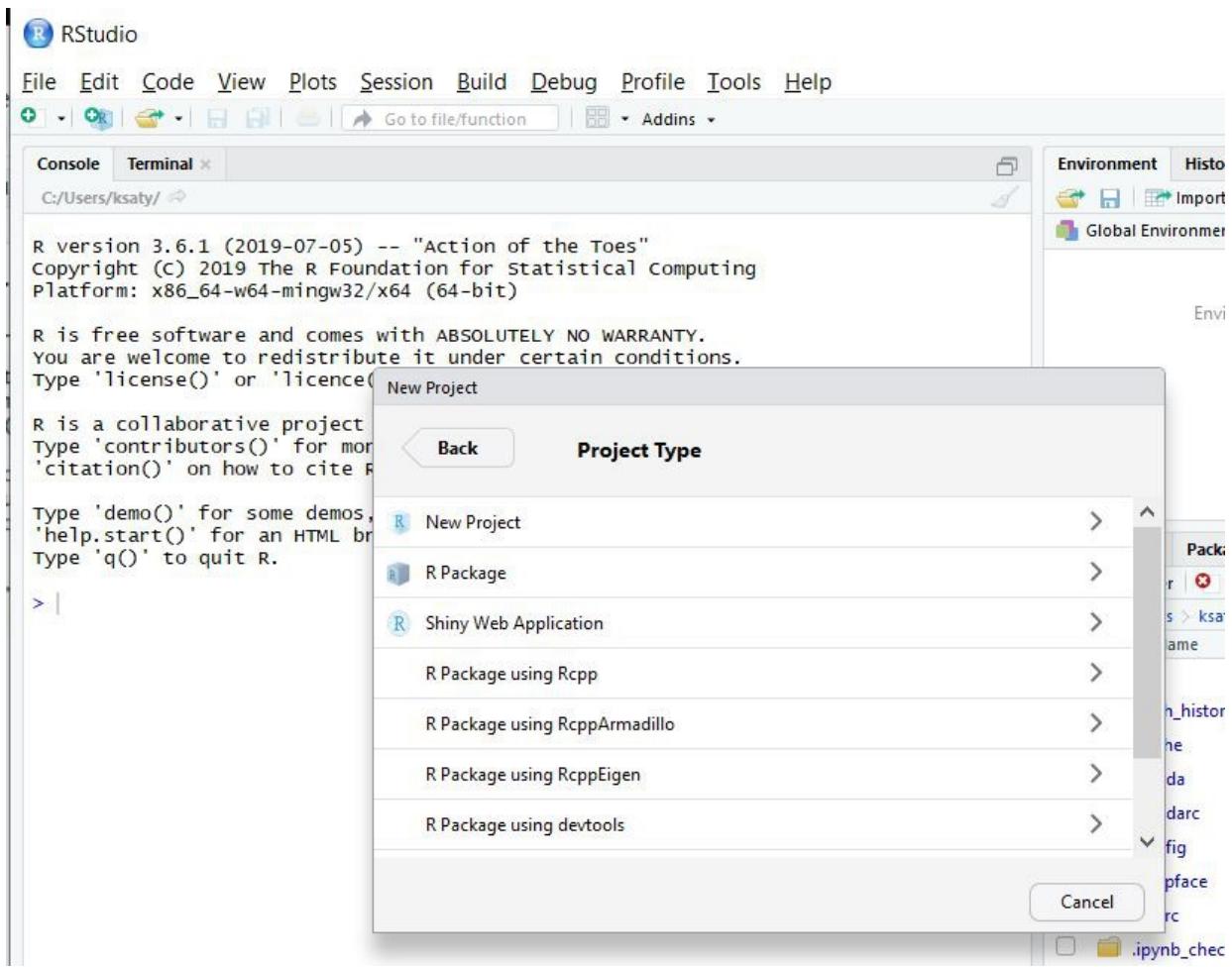
This will select the current directory, which you have chosen using this file browser as your working directory. Once you set the working directory, you are ready to program in R Studio.

Create an RStudio project

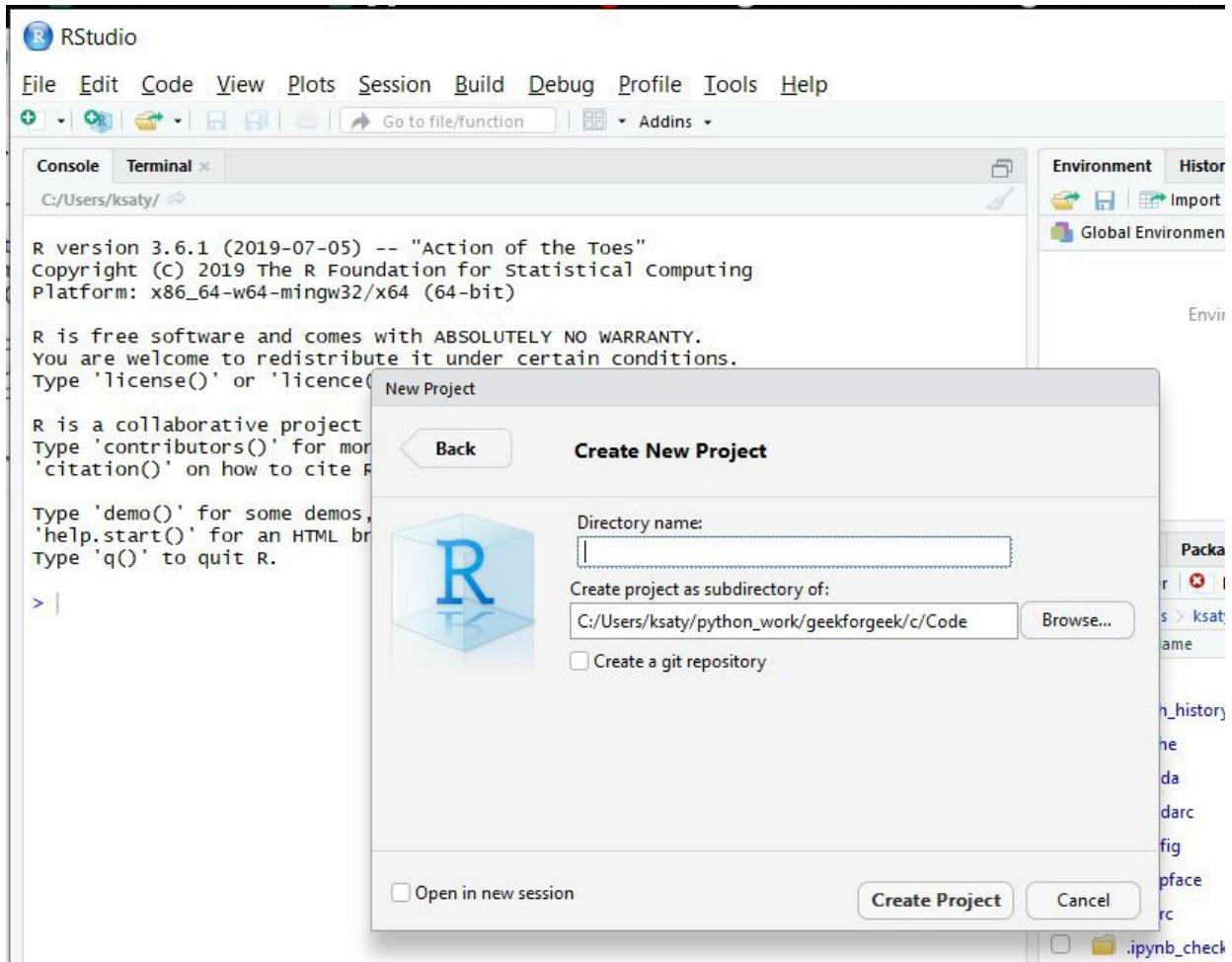
Step 1: Select the FILE option and select create option.



Step 2: Then select the New Project option.



Step 3: Then choose the path and directory name.



Finally, project are created in a specific location:

The screenshot shows the RStudio interface with the R console tab selected. The console window displays the standard R startup message, which includes the version number (R version 3.6.1), copyright information, platform details, and various usage instructions. The path 'C:/Users/ksaty/python_work/geekforgeek/c/Code/Code' is highlighted in yellow at the top of the console window. The right side of the interface features the Environment, Files, and Plots panes.

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (c) 2019 The R Foundation for Statistical computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

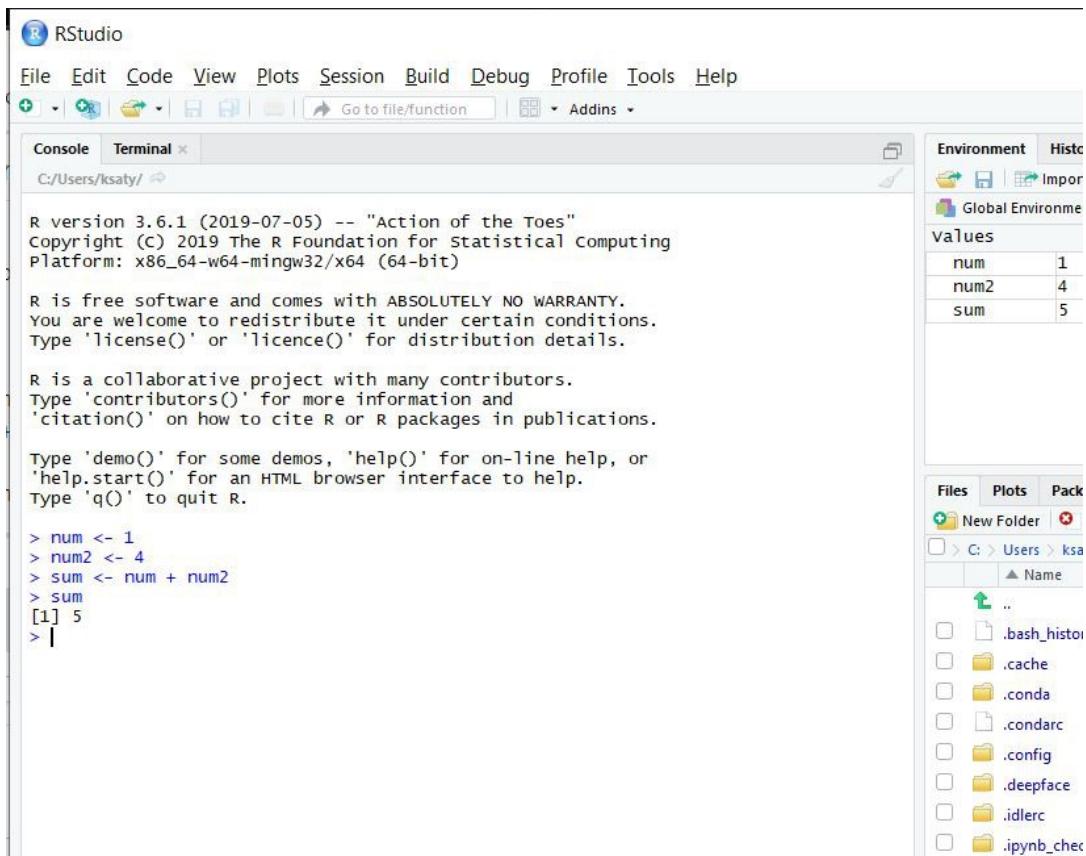
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

> |
```

Creating your first R script

Here we are adding two numbers in R studio.



Navigating directories in R studio

- **getwd()**: Returns the current working directory.
- **setwd()**: Set the working directory.
- **dir()**: Return the list of the directory.
- **sessionInfo()**: Return the session of the windows.
- **date()**: Return the current date.

How to Install R and R Studio?

[R Programming Language](#) is a language and free software environment, available under GNU license, supported by the R Foundation for Statistical Computing. Install R and R Studio The language is most widely known for its powerful statistical and data interpretation capabilities.

To use R language, you need the [R environment](#) to be Install R and R Studio on your machine, and an IDE (Integrated development environment) to run the language (can also be run using CMD on Windows or Terminal on Linux).

Why use R Studio?

- It is a powerful IDE, specifically used for the R language.
- Provides literate programming tools, which allow the use of R scripts, outputs, text, and images in reports, Word documents, and even HTML files.
- The use of Shiny (open-source R package) allows us to create interactive content in reports and presentations.

How to Install R and R Studio

To Install R and R Studio on windows we will follow the following steps.

Step 1: First, you need to set up an R environment in your local machine. You can download the same from [r-project.org](#).

The screenshot shows the Posit website with a navigation bar at the top. Below the navigation, there is a promotional message about using RStudio on the cloud. The main content area is divided into two sections: '1: Install R' on the left and '2: Install RStudio' on the right. Both sections contain sub-instructions and a 'DOWNLOAD RSTUDIO DESKTOP' button.

Don't want to download or install anything? Get started with RStudio on [Posit Cloud for free](#). If you're a professional data scientist looking to download RStudio and also need common enterprise features, don't hesitate to [book a call with us](#).

1: Install R

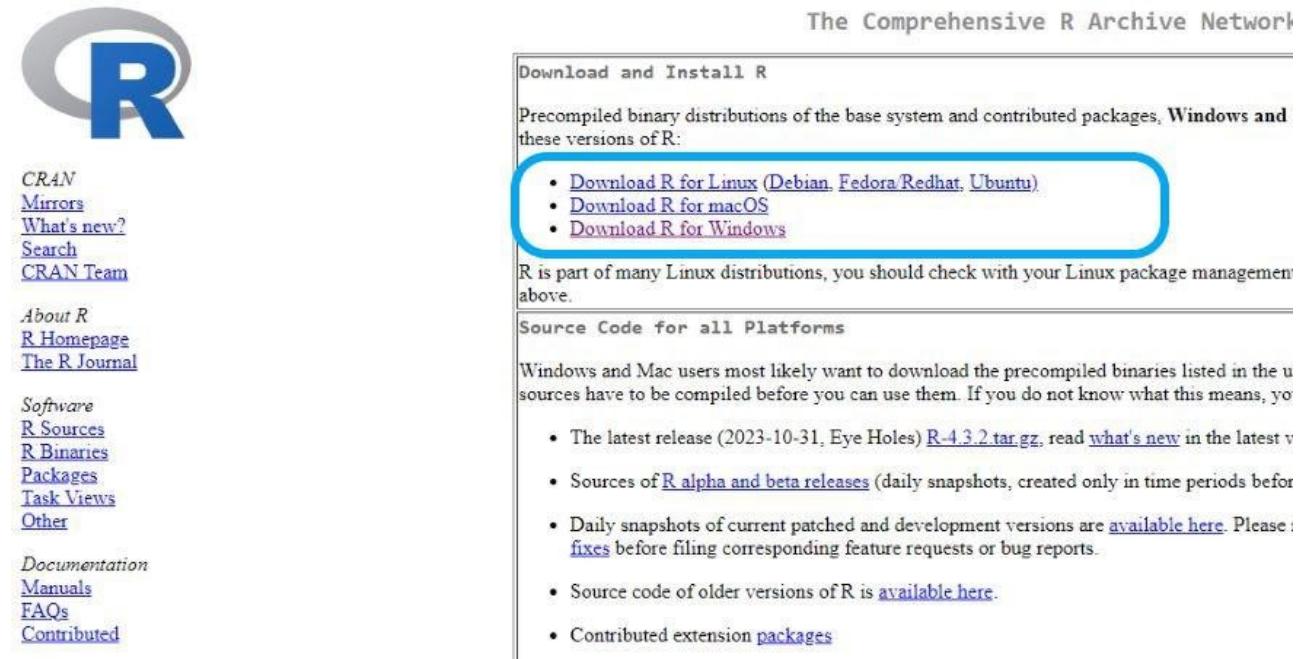
RStudio requires R 3.3.0+. Choose a version of R that matches your computer's operating system.

2: Install RStudio

[DOWNLOAD RSTUDIO DESKTOP](#)

Install R and R Studio

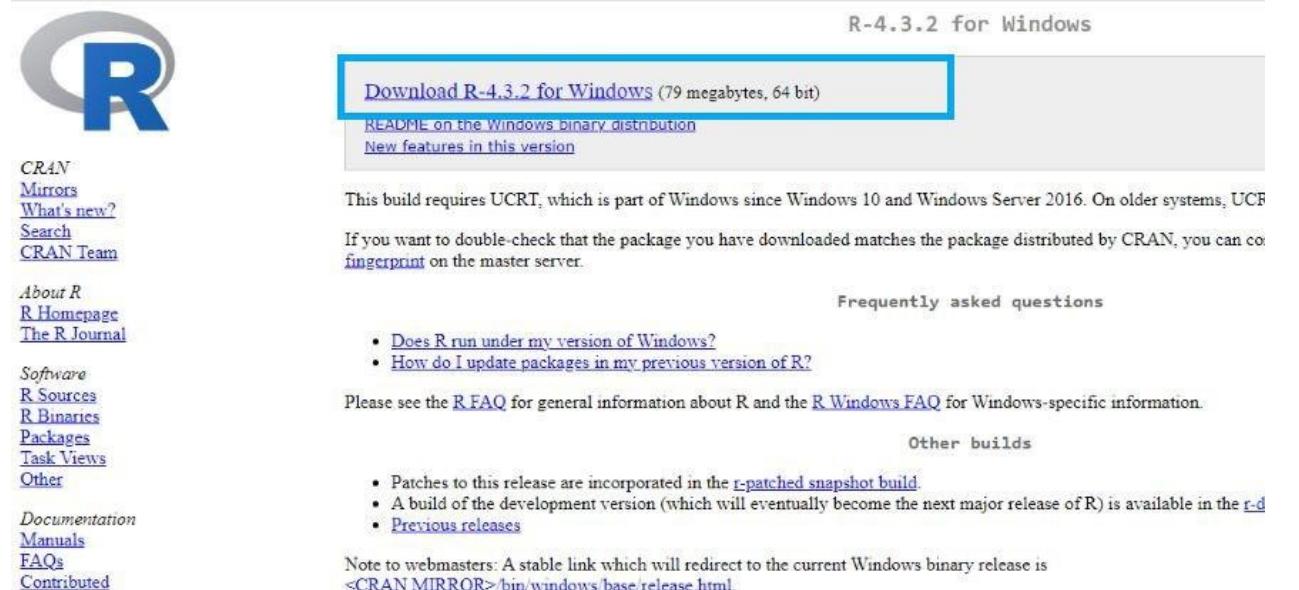
You have to download both the applications first go with R Base and then install RStudio. after click on install R you will get a new page like this.



The screenshot shows the CRAN homepage with a large R logo at the top. Below it, there are several navigation links: CRAN, Mirrors, What's new?, Search, CRAN Team, About R, R Homepage, and The R Journal. Under Software, there are links for R Sources, R Binaries, Packages, Task Views, and Other. Under Documentation, there are links for Manuals, FAQs, and Contributed. The main content area is titled "The Comprehensive R Archive Network". It features a section titled "Download and Install R" which says "Precompiled binary distributions of the base system and contributed packages, Windows and these versions of R:". Below this, a blue-bordered box contains three links: "Download R for Linux (Debian, Fedora/Redhat, Ubuntu)", "Download R for macOS", and "Download R for Windows". A note below states: "R is part of many Linux distributions, you should check with your Linux package manager above." Another section titled "Source Code for all Platforms" provides instructions for Windows and Mac users, listing the latest release, alpha/beta releases, daily snapshots, and source code for older versions.

Install R and R Studio

Here we can select the linux,mac or windows any one according to users system. you have to click on for which you want to install.

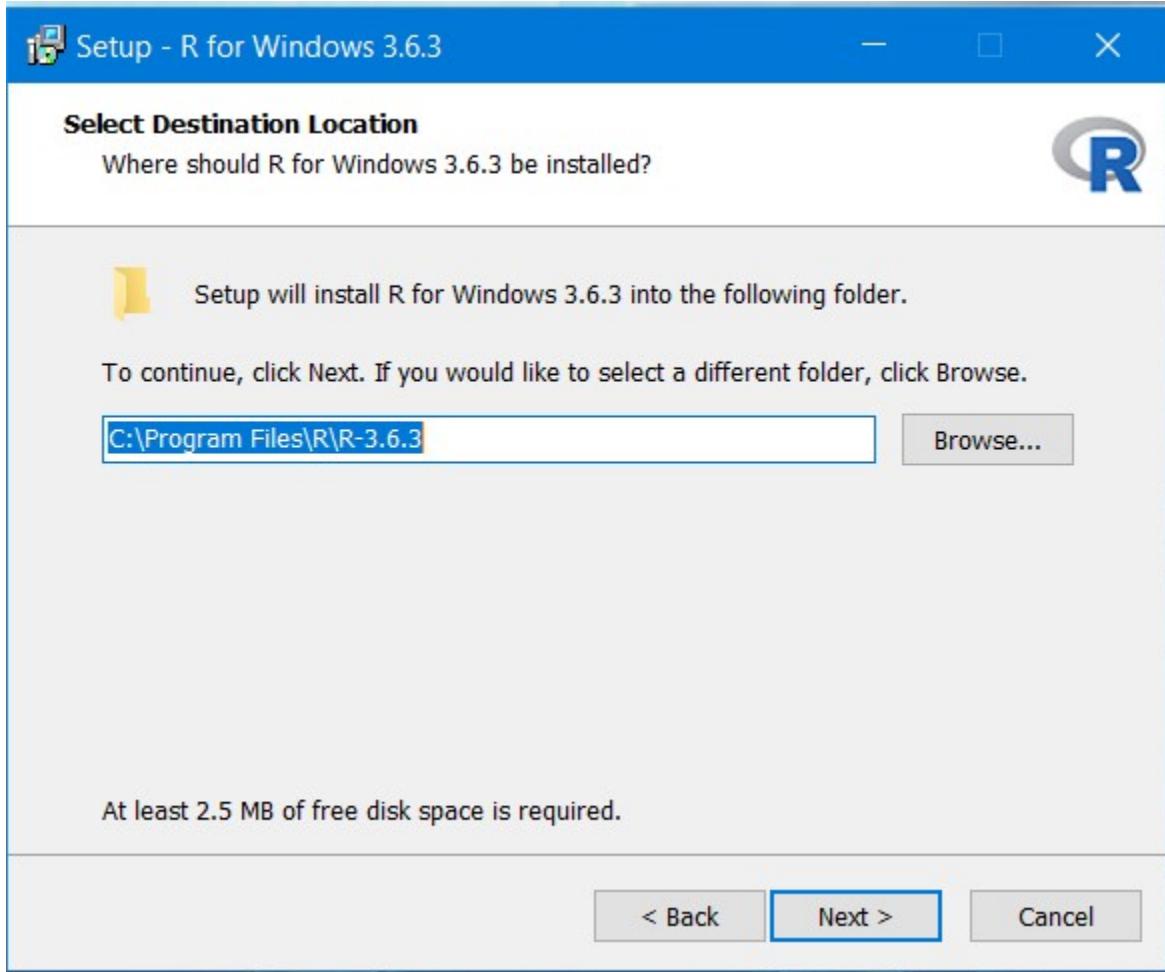


The screenshot shows the CRAN website for the R-4.3.2 for Windows download. It features the same navigation bar as the previous page. The main content is titled "R-4.3.2 for Windows". It has a prominent blue-bordered box containing the link "Download R-4.3.2 for Windows (79 megabytes, 64 bit)". Below this are links for "README on the Windows binary distribution" and "New features in this version". A note states: "This build requires UCRT, which is part of Windows since Windows 10 and Windows Server 2016. On older systems, UCRT is available via the Windows Feature Experience Pack." It also mentions: "If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare its SHA256 [fingerprint](#) on the master server." To the right, there are sections for "Frequently asked questions" (with links to "Does R run under my version of Windows?" and "How do I update packages in my previous version of R?"), "Other builds" (with links to "r-patched snapshot build", "r-devel", and "Previous releases"), and a note for webmasters about a stable link redirecting to the current Windows binary release. At the bottom, there is a link to "CRAN MIRROR>/bin/windows/base/release.html".

Install R and R Studio

now click on the link show above in image so R base start downloading and after again go to main page and download and click on Install RStudio.

Step 2: After downloading R for the Windows platform, install it by double-clicking it.



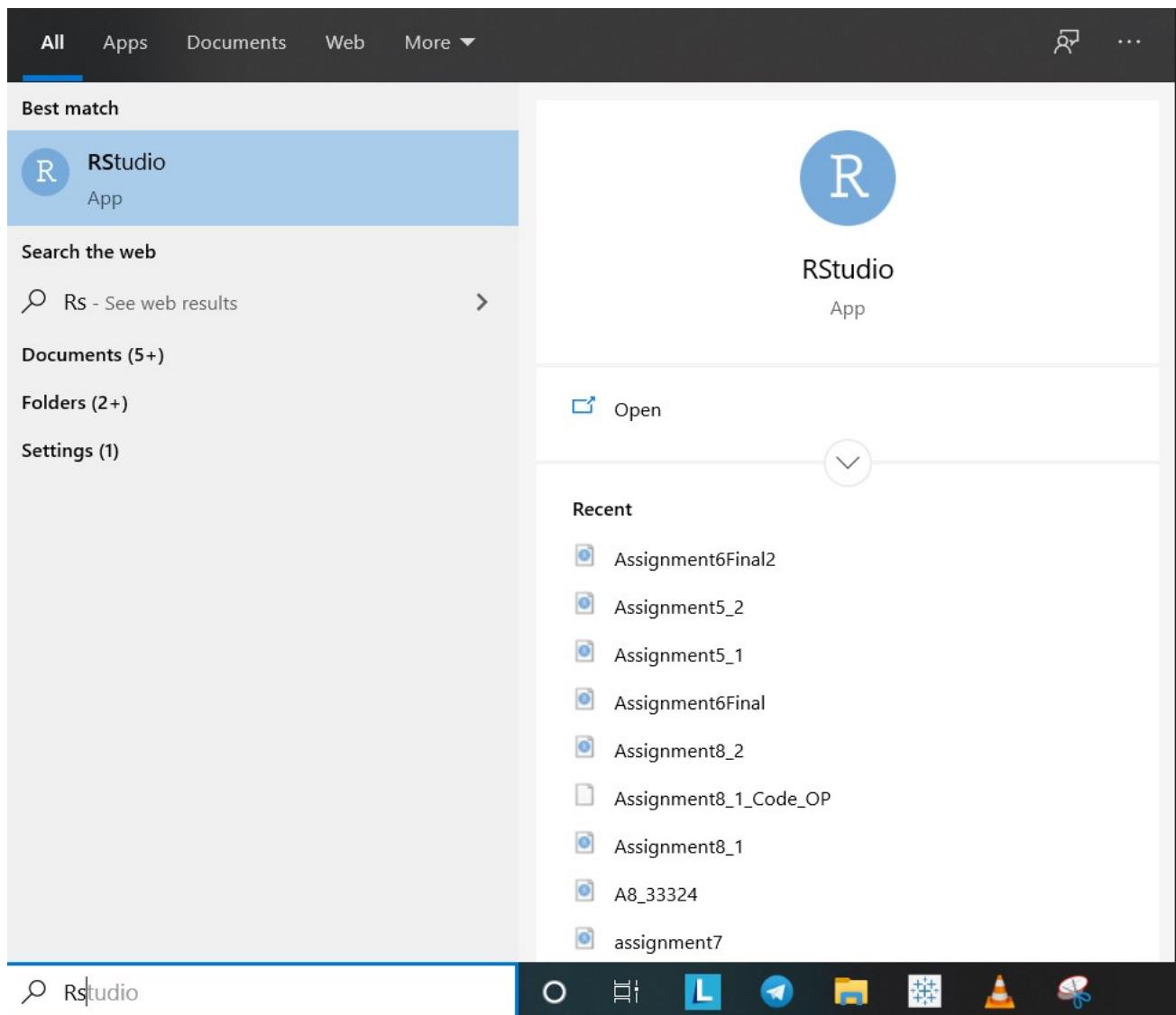
Step 3: Download R Studio from their official page. Note: It is free of cost (under AGPL licensing).

Step 4: After downloading, you will get a file named "RStudio-1.x.xxxx.exe" in your Downloads folder.

Step 5: Double-click the installer, and install the software.

Step 6: Test the R Studio installation

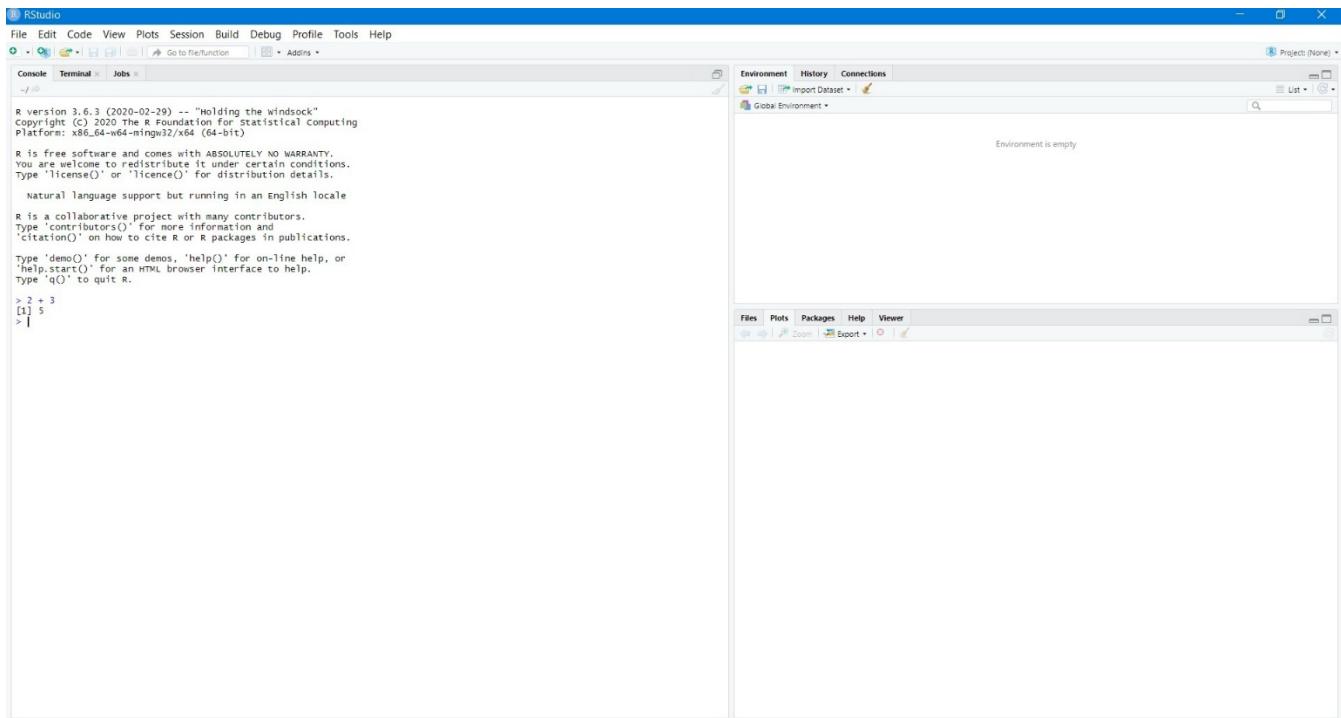
- Search for RStudio in the Window search bar on Taskbar.



- Start the application.
- Insert the following code in the console.

Input : print('Hello world!')

Output : [1] "Hello world!"



Step 7: Your installation is successful.

Installing R Studio on Ubuntu

Installing R Studio on Ubuntu has steps similar to Windows:

Through Terminal

Step 1: Open terminal (Ctrl+Alt+T) in Ubuntu.

Step 2: Update the package's cache.

```
sudo apt-get update
```

Step 3: Install R environment.

```
sudo apt -y install r-base
```

Check for the version of R package using

```
R --version
```

Step 4: Check R installation by using the following command.

```
user@Ubuntu:~$ R
```

(Note that R version should be 3.6+ to be able to install all packages like tm, e1071, etc.). If there is issue with R version, see the end of the post.

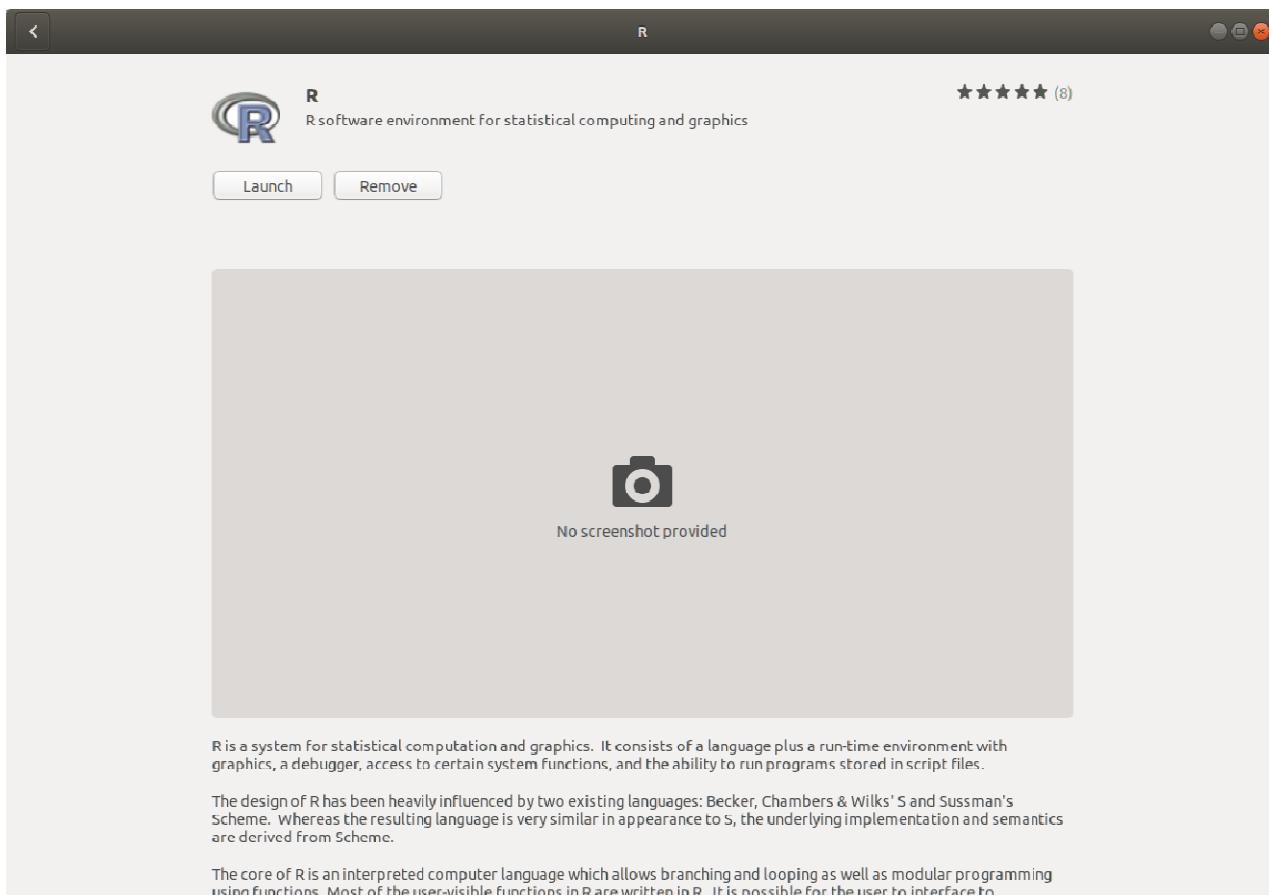
Step 5: Exit the terminal.

Through Ubuntu software Center

Step 1: Open Ubuntu Software Center.

Step 2: Search for r-base.

Step 3: Click install.



Install Rstudio on Ubuntu

Step 1: Install gdebi package to install .deb packages easily.

```
sudo add-apt-repository universe  
sudo apt-get install gdebi-core
```

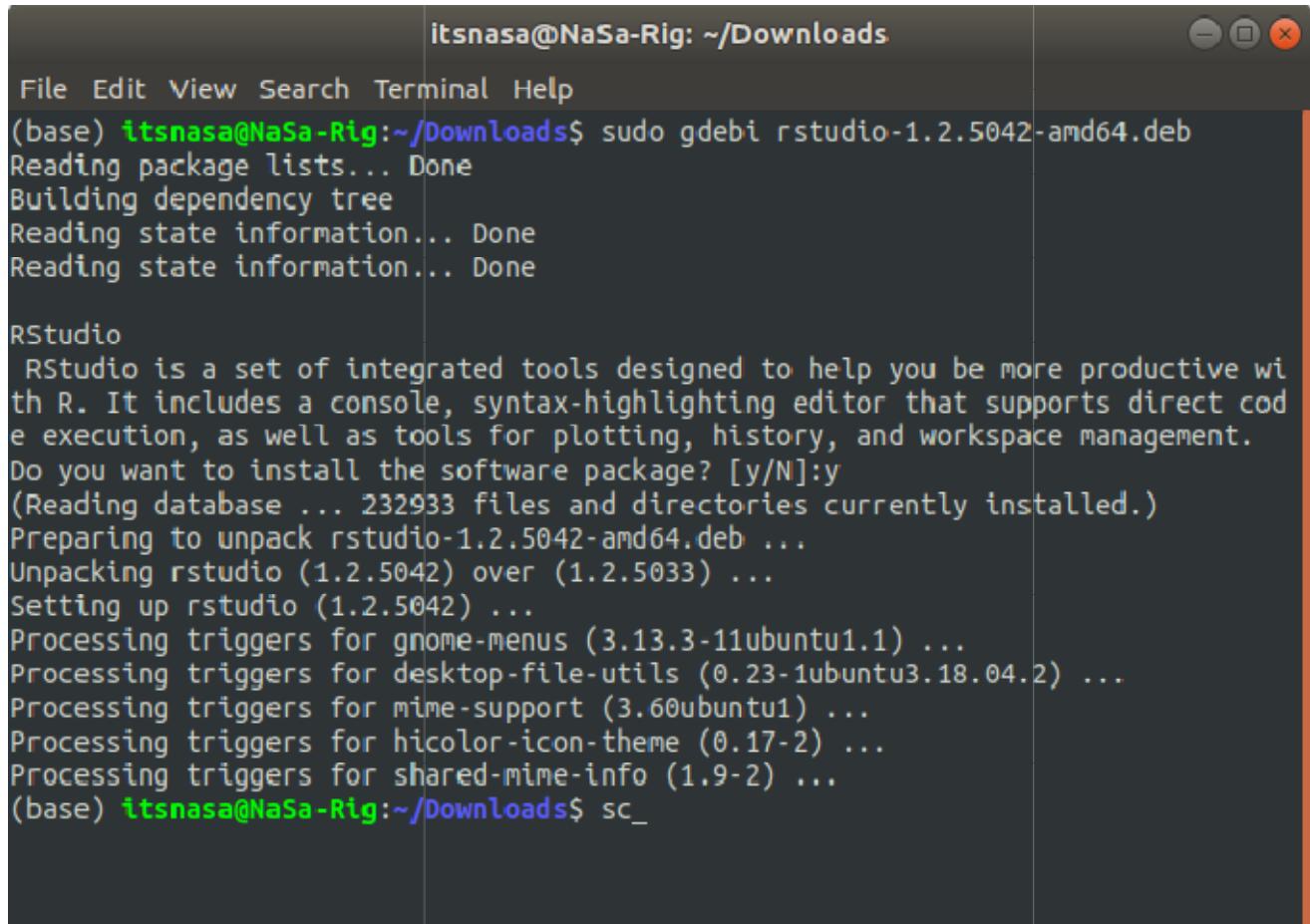
Step 2: Go to R Studio downloads and select the latest *.deb package available under Ubuntu 18/Debian 10.

Step 3: Navigate to the Downloads folder in the local machine.

```
$ cd Downloads/  
$ ls  
rstudio-1.2.5042-amd64.deb
```

Step 4: Install using gdebi package.

```
sudo gdebi rstudio-1.2.5042-amd64.deb
```



```
itsnasa@NaSa-Rig: ~/Downloads  
File Edit View Search Terminal Help  
(base) itsnasa@NaSa-Rig:~/Downloads$ sudo gdebi rstudio-1.2.5042-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 ... 232933 files and directories currently installed.)  
Preparing to unpack rstudio-1.2.5042-amd64.deb ...  
Unpacking rstudio (1.2.5042) over (1.2.5033) ...  
Setting up rstudio (1.2.5042) ...  
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...  
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...  
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) ...  
(base) itsnasa@NaSa-Rig:~/Downloads$ sc_
```

Step 5: Run the RStudio using Terminal

```
user@Ubuntu:~/Downloads/ $ rstudio
```

Alternatively, use the menu to search for Rstudio.

Step 6: Test the R Studio using the basic “Hello world!” command and exit.

Input : print('Hello world!')
Output : [1] "Hello world!"

Alternatively, RStudio can be installed through **Ubuntu Software** as well, but using the above approach generally guarantees the latest version is installed.

If there are issues with the R version getting downloaded or the previously installed version is older, check R version with

```
R --version
```

Now, Run the following commands in Terminal (Ctrl + Alt + T)

Add the key to secure APT from the CRAN package list:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys  
E298A3A825C0D65DFD57CBB651716619E084DAB9
```

Add the latest CRAN repository to the repository list. (This is for Ubuntu 18.04 specifically):

```
sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu bionic-cran35/'
```

Update the package cache:

```
sudo apt update
```

Install the r-base package:

```
sudo apt install r-base
```

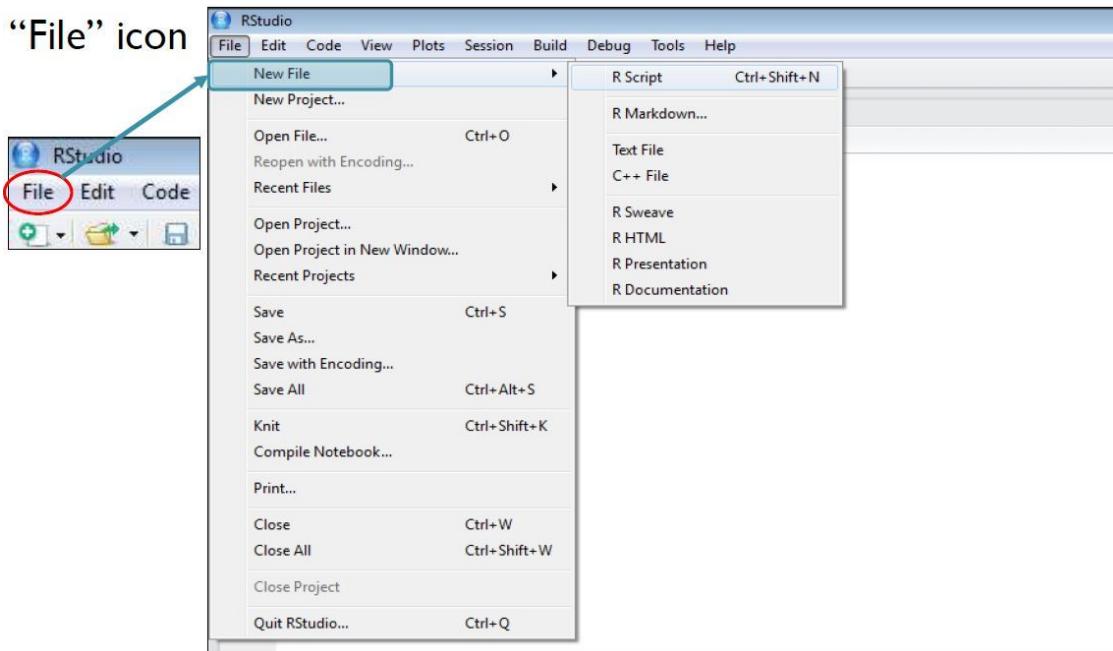
Creation and Execution of R File in R Studio

R Studio is an integrated development environment(IDE) for R. IDE is a GUI, where you can write your quotes, see the results and also see the variables that are generated during the course of programming. R is available as an Open Source software for Client as well as Server Versions.

Creating an R file

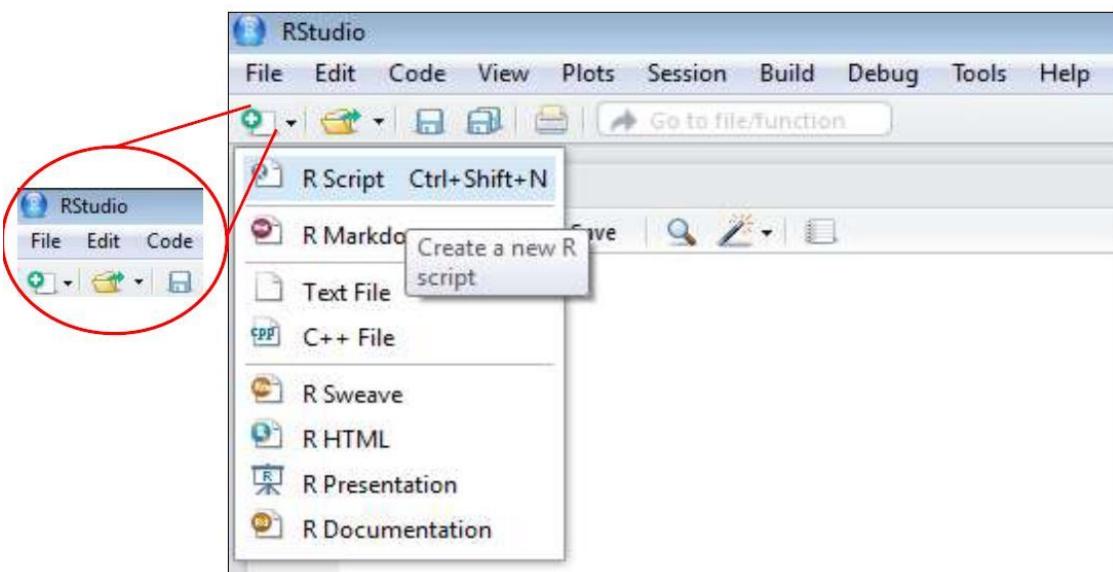
There are two ways to create an R file in R studio:

- You can click on the File tab, from there when you click it will give a drop-down menu, where you can select the new file and then R script, so that, you will get a new file open.

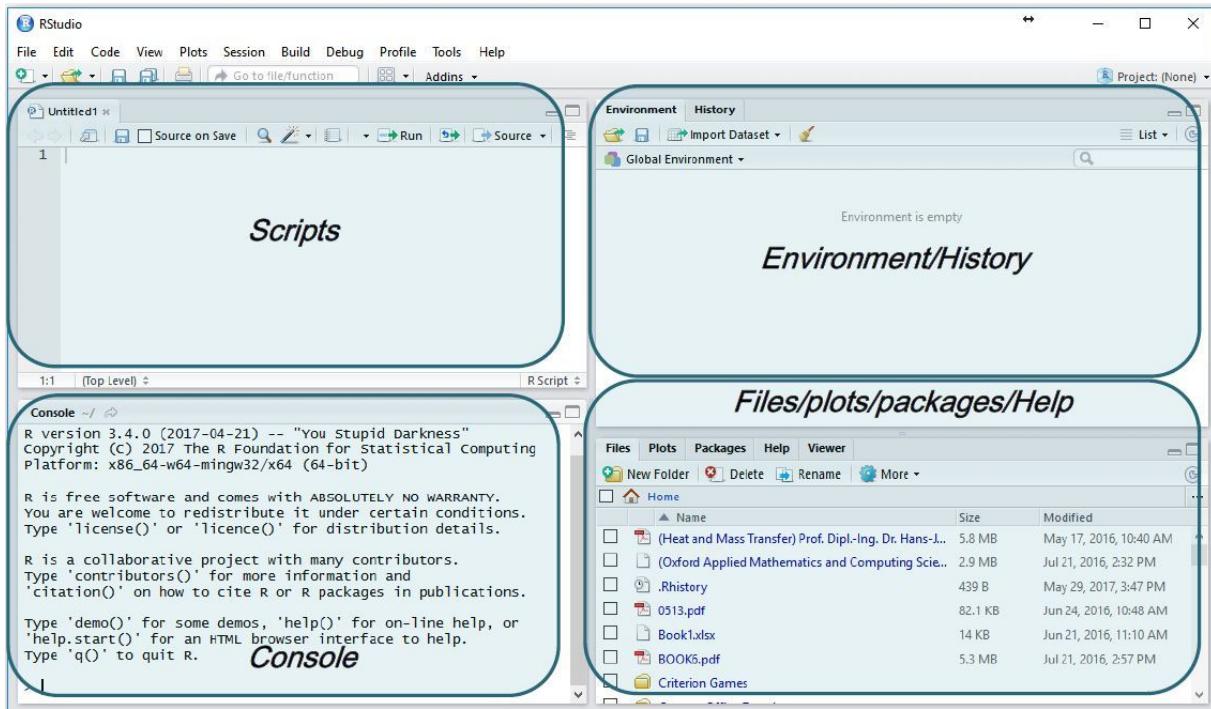


- Use the plus button, which is just below the file tab and you can choose R script, from there, to open a new R script file.

By clicking the icon “ ”below the toolbar



Once you open an R script file, this is how an R Studio with the script file open looks like.



So, 3 panels console, environment/history and file/plots panels are there. On top left you have a new window, which is now being opened as a script file. Now you are ready to write a script file or some program in R Studio.

Writing Scripts in an R File

Writing scripts to an R file is demonstrated here with an example:

The screenshot shows the RStudio interface. On the left, the 'Script Editor' window titled 'Untitled1*' contains the following R code:

```
1 a = 11
2 b = a*10
3 print(c(a,b))
```

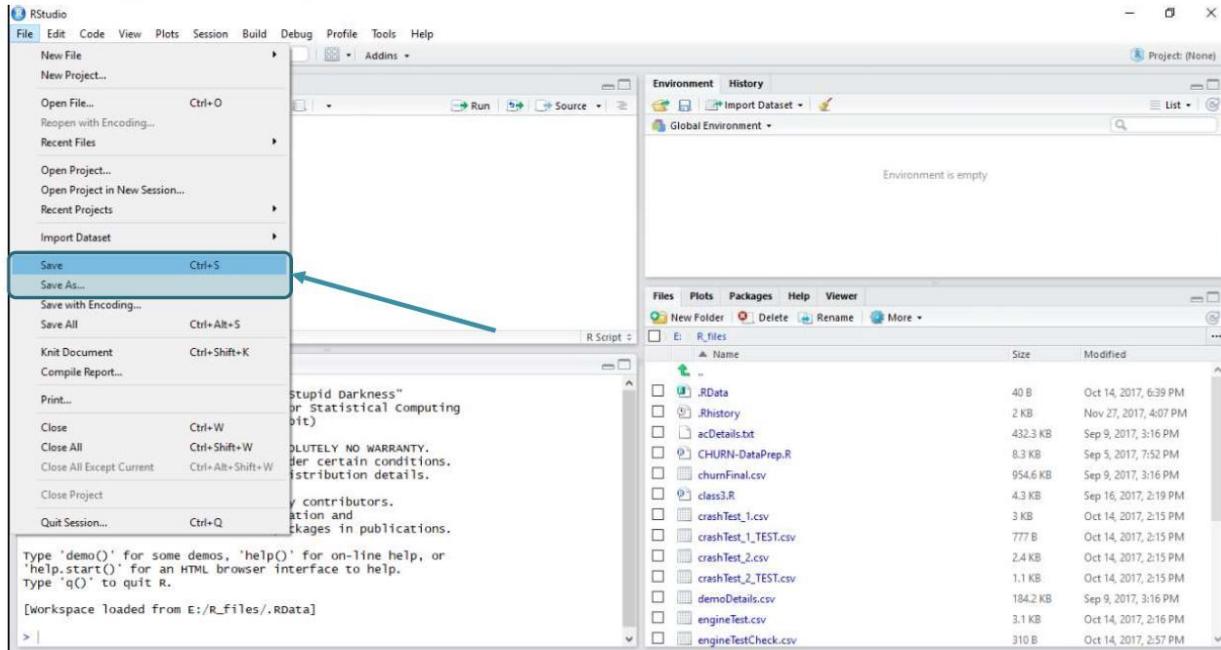
In the center, the 'Environment' viewer shows 'Global Environment' with the message 'Environment is empty'. At the bottom, the 'File Browser' shows a directory structure under 'E:/R_files' with the following files:

Name	Size	Modified
..		
.RData	40 B	Oct 14, 2017, 6:39 PM
.Rhistory	2 KB	Nov 27, 2017, 4:07 PM
acDetails.txt	432.3 KB	Sep 9, 2017, 3:16 PM
CHURN-DataPrep.R	8.3 KB	Sep 5, 2017, 7:52 PM
churnFinal.csv	954.6 KB	Sep 9, 2017, 3:16 PM
class3.R	4.3 KB	Sep 16, 2017, 2:19 PM
crashTest_1.csv	3 KB	Oct 14, 2017, 2:15 PM
crashTest_1 TEST.csv	777 B	Oct 14, 2017, 2:15 PM

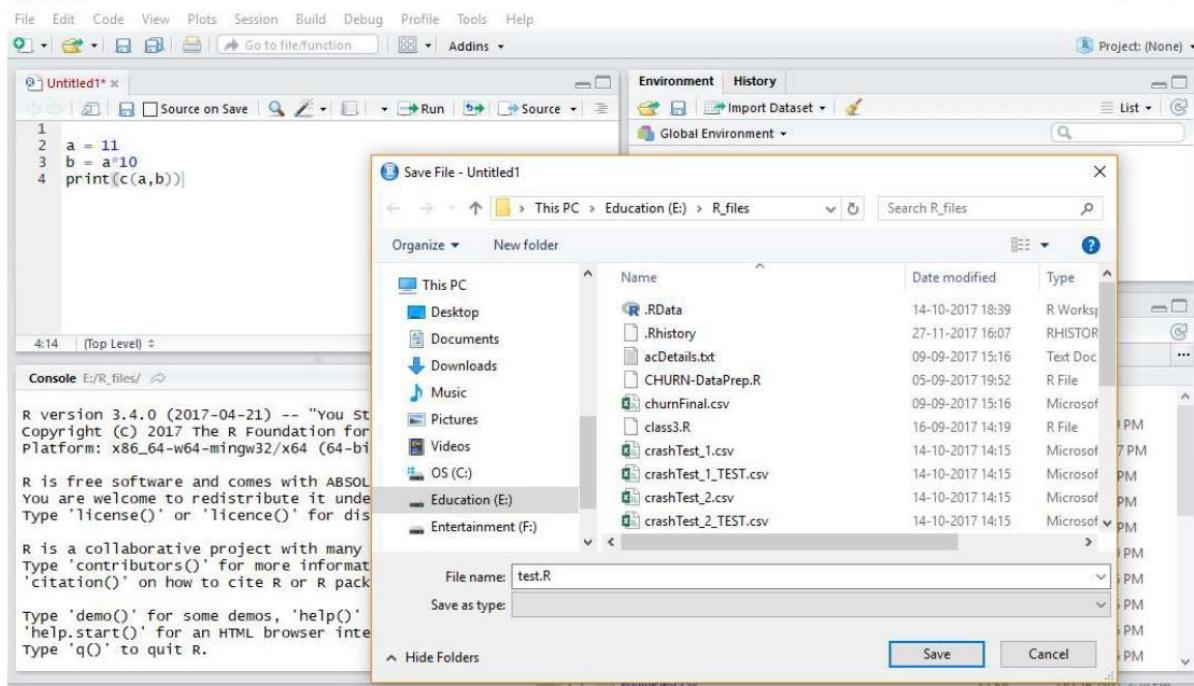
In the above example, a variable 'a' is assigned with a value 11, in the first line of the code and there is b which is 'a' times 10, that is the second command. Here, the code is evaluating the value of a times 10 and assign the value to the b and the third statement, which is `print(c(a, b))` means concatenates this a and b and print the result. So, this is how a script file is written in R. After writing a script file, there is a need to save this file before execution.

Saving an R File

Let us see, how to save the R file. From the file menu if you click the file tab you can either save or save as button. When you want to save the file if you click the save button, it will automatically save the file has untitled x. So, this x can be 1 or 2 depending upon how many R scripts you have already opened.



Or, it is a nice idea to use the Save as button, just below the Save one, so that, you can rename the script file according to your wish. Let us suppose we have clicked the Save as button. This will pop out a window like this, where you can rename the script file as test.R. Once you rename, then by clicking the save button you can save the script file.

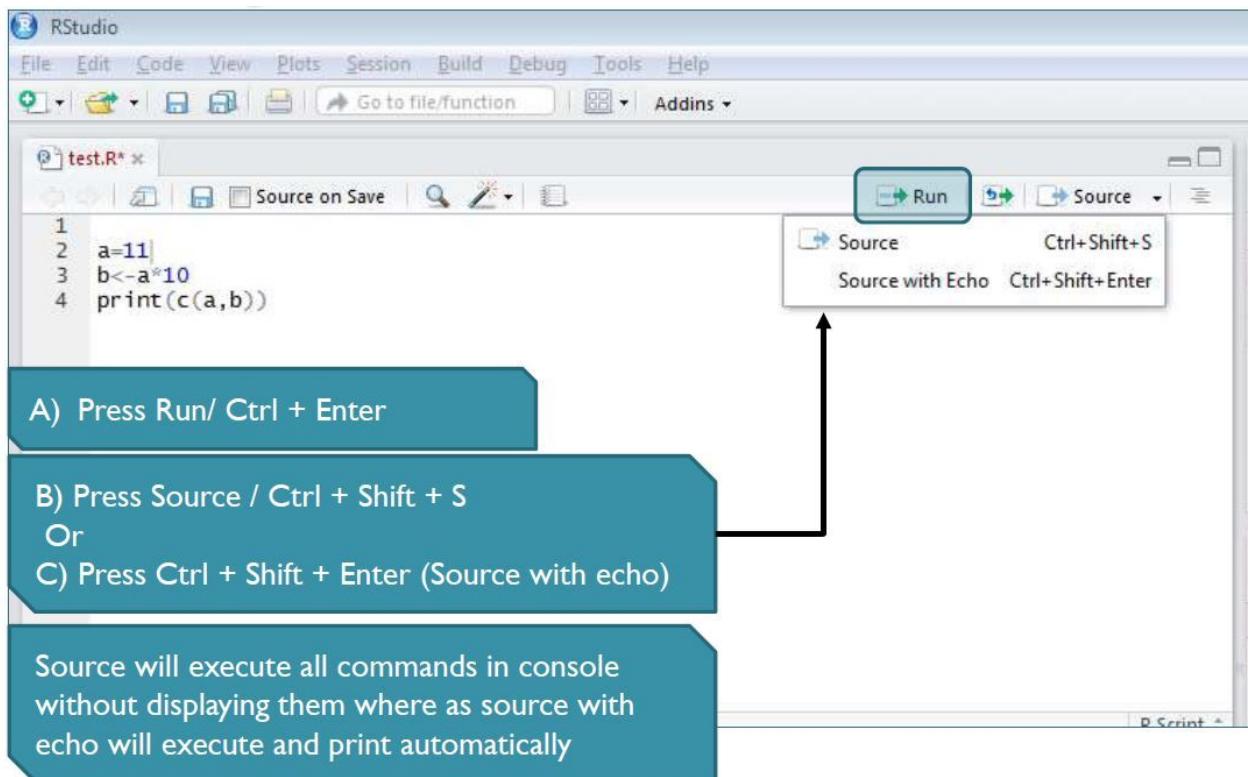


So now, we have seen how to open an R script and how to write some code in the R script file and save the file.

The next task is to execute the R file.

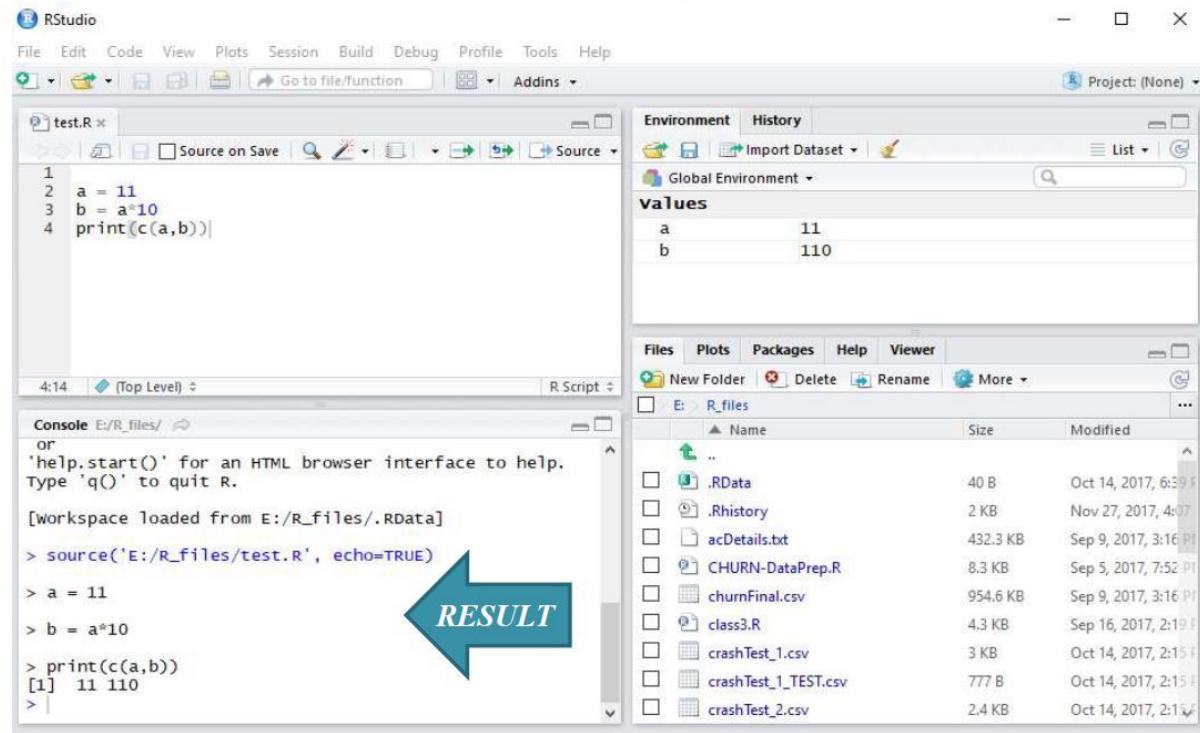
Execution of an R file

There are several ways in which the execution of the commands that are available in the R file is done.



- **Using the run command:** This “run” command can be executed using the GUI, by pressing the run button there, or you can use the Shortcut key control + enter.
What does it do?
It will execute the line in which the cursor is there.
- **Using the source command:**
This “source” command can be executed using the GUI, by pressing the source button there, or you can use the Shortcut key control + shift + S.
What does it do?
It will execute the whole R file and only print the output which you wanted to print.
- **Using the source with echo command:**
This “source with echo” command can be executed using the GUI, by pressing the source with echo button there, or you can use the Shortcut key control + shift + enter.
What does it do?
It will print the commands also, along with the output you are printing.

So, this is an example, where R file is executed, using the source with echo command.



```
test.R x
1 a = 11
2 b = a*10
3 print(c(a,b))

4:14 (Top Level) R Script

Console E:/R_files/
or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from E:/R_files/.RData]
> source('E:/R_files/test.R', echo=TRUE)
> a = 11
> b = a*10
> print(c(a,b))
[1] 11 110
> |
```

RESULT

Environment History
Import Dataset Global Environment
values
a 11
b 110

Files Plots Packages Help Viewer
New Folder Delete Rename More
E: R_files
Name Size Modified
..
.RData 40 B Oct 14, 2017, 6:39 PM
.Rhistory 2 KB Nov 27, 2017, 4:07 PM
acDetails.txt 432.3 KB Sep 9, 2017, 3:16 PM
CHURN-DataPrep.R 8.3 KB Sep 5, 2017, 7:52 PM
churnFinal.csv 954.6 KB Sep 9, 2017, 3:16 PM
class3.R 4.3 KB Sep 16, 2017, 2:19 PM
crashTest_1.csv 3 KB Oct 14, 2017, 2:15 PM
crashTest_1_TEST.csv 777 B Oct 14, 2017, 2:15 PM
crashTest_2.csv 2.4 KB Oct 14, 2017, 2:15 PM

It can be seen in the console, that it printed the command `a = 11` and the command `b = a*10` and also the output `print(c(a, b))` with the values.

So, `a` is 11 and `b` is 11 times 10, this is 110. This is how the output will be printed in the console.

Values of `a` and `b` are also shown in the environment panel.

Run command over Source command:

- Run can be used to execute the selected lines of R code.
- Source and Source with echo can be used to run the whole file.
- The advantage of using Run is, you can troubleshoot or debug the program when something is not behaving according to your expectations.
- The disadvantages of using run command are, it populates the console and makes it messy unnecessarily.

Clear the Console and the Environment in R Studio

[R Studio](#) is an integrated development environment(IDE) for R. IDE is a GUI, where you can write your quotes, see the results and also see the variables that are generated during the course of programming.

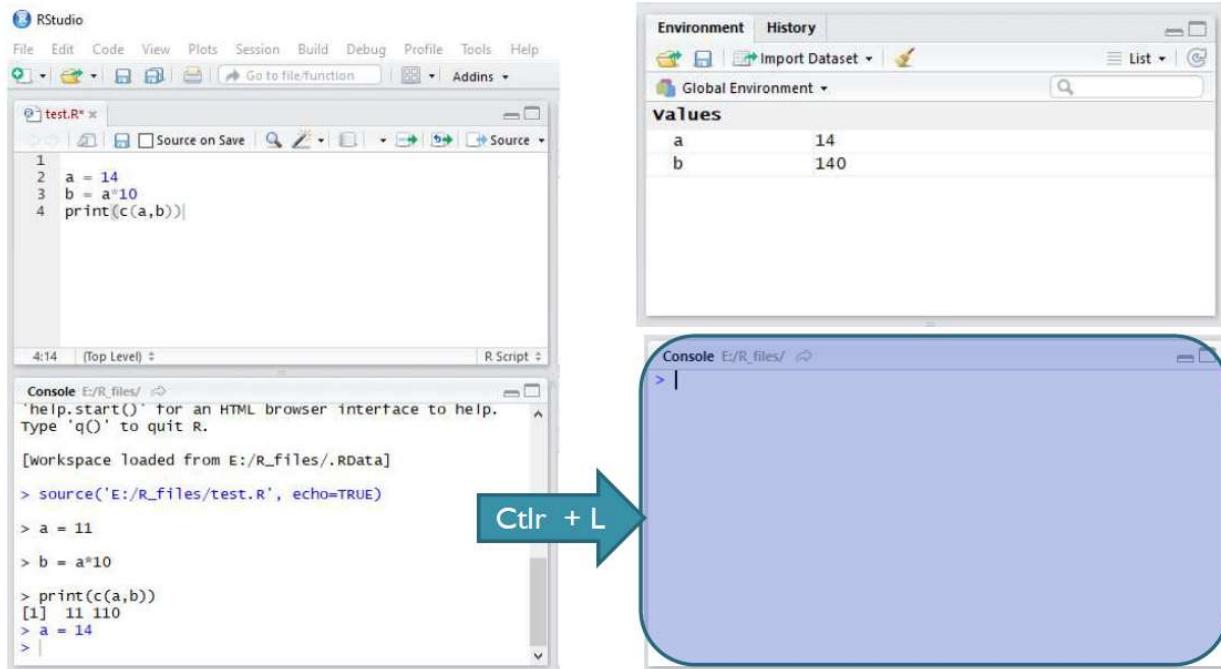
Clearing the Console

We Clear console in R and RStudio, In some cases when you run the codes using “**source**” and “**source with echo**” your console will become messy. And it is needed to clear the console. So let’s now look at how to clear the console. The console can be cleared using the shortcut key “**ctrl + L**”.

Example:

In this below screenshot, an R code is written in the script tab defined **a** and calculated **b** and printed **a**, **b**. When this code is executed using “**source with echo**” all the commands will get printed in the console tab. Now, to clear this console click on the console tab and enter the key combination “**ctrl + L**”. Once it is done the console will get cleared.

“control +L”



Note: Remember that clearing the console will not delete the variables that are there in the workspace. You can see that in the environment tab even though we have cleared the console in the workspace we still have the variables that are created earlier.

Clearing the Environment

Variables on the R environment can be cleared in two ways:

Using **rm()** command:

When you want to clear a single variable from the R environment you can use the “**rm()**” command followed by the variable you want to remove.

-> `rm(variable)`

variable: that variable name you want to remove.

If you want to delete all the variables that are there in the environment what you can do is you can use the “rm” with an argument “list” is equal to “ls” followed by a parenthesis.

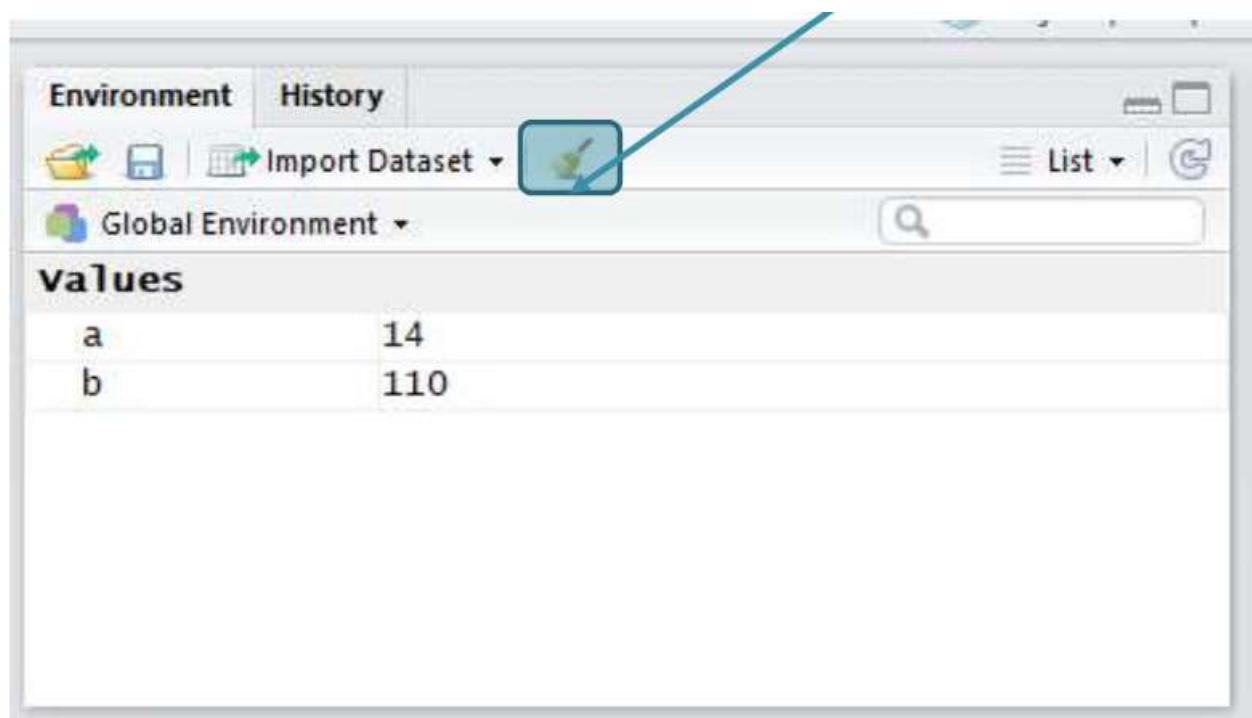
-> `rm(list=ls())`

Using the GUI:

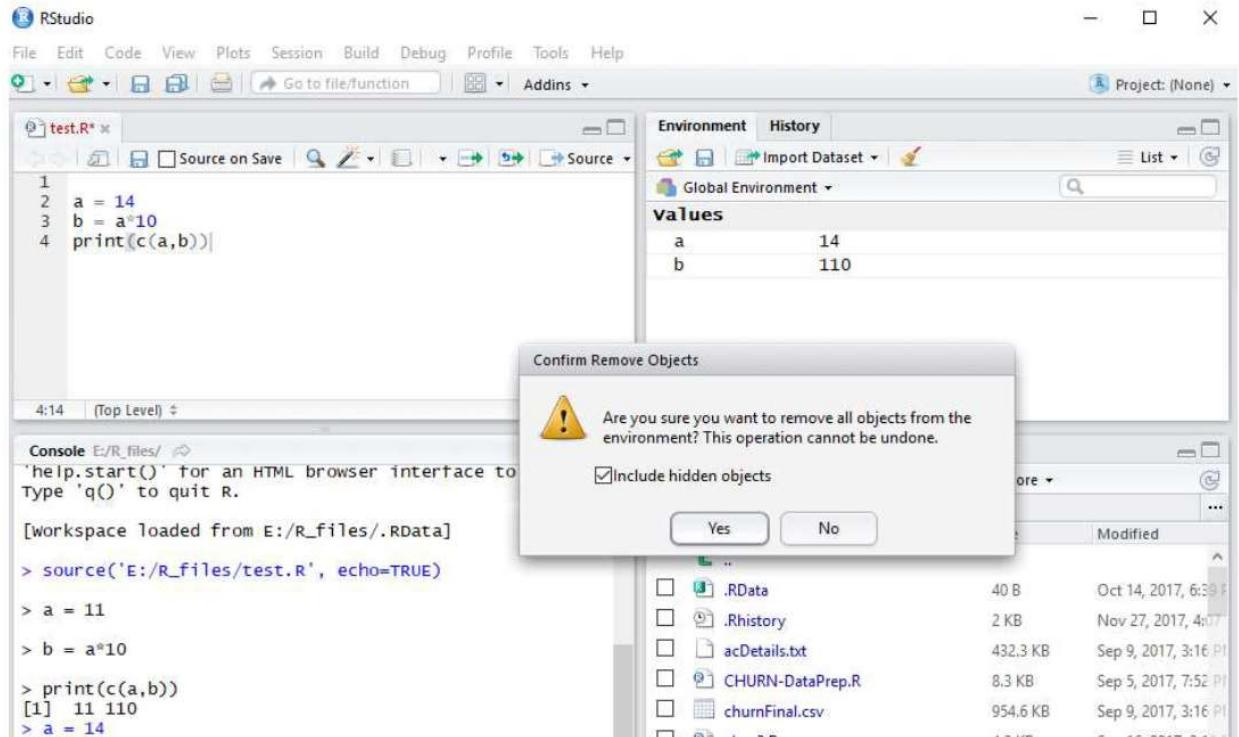
We can also clear all the variables in the environment using the GUI in the environment pane.

How does it works?

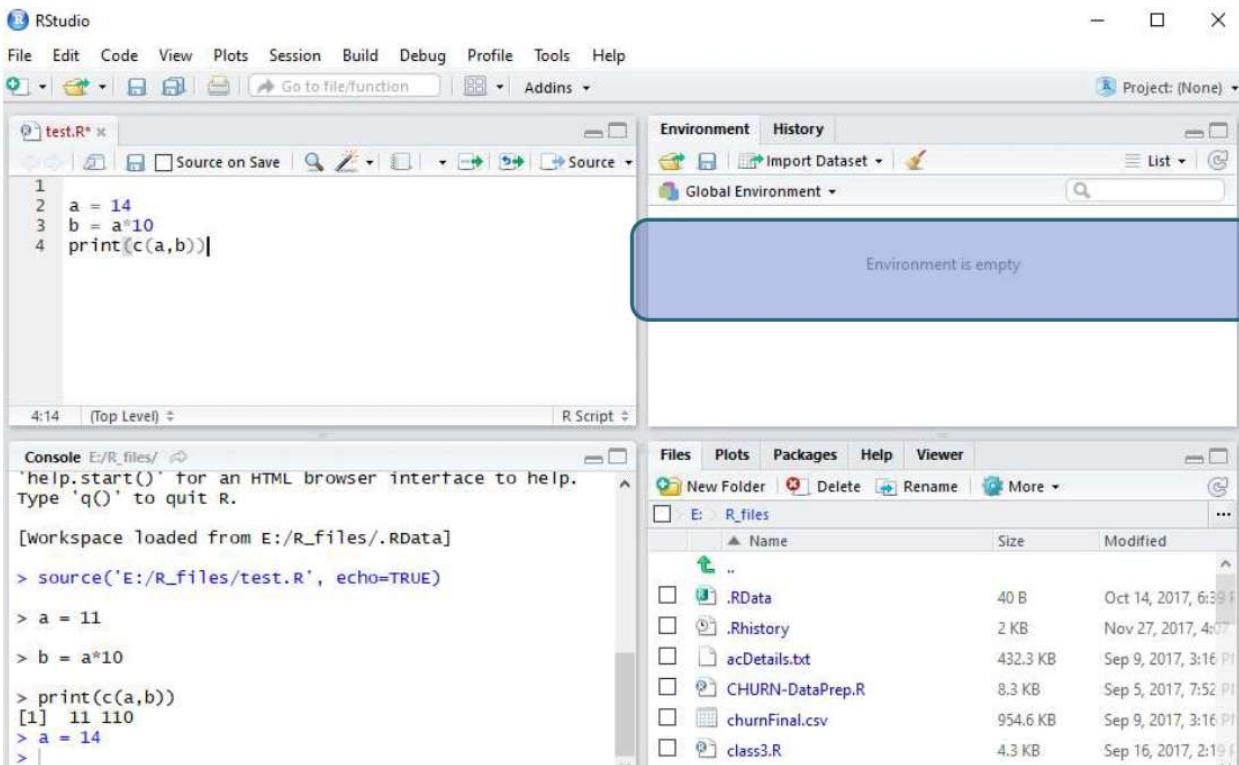
You see this brush button in the environment pane.



So when you press the brush button it will pop up a window saying “you want to remove all the objects from the environment?”



And if you say yes it will clear all the variables which are shown here and you can see the environment is empty now.



Hello World in R Programming

When we start to learn any programming languages we do follow a tradition to begin HelloWorld as our first basic program. Here we are going to learn that tradition. An interesting thing about [R programming](#) is that we can get our things done with very little code.

Before we start to learn to code, let's see the process of where to download the needed Softwares and programs.

- Go to the official site of [R programming](#) and download R for windows(or Mac).
- Install IDE like Rstudio, RTVS, StatET for running programs(you can download Rstudio [here](#)).We can write and run programs in IDE, but you have to download R before installing IDE.

HelloWorld Program

For the HelloWorld program, we just need a print function. No need of any packages or main functions, just a simple print function is enough.

```
print("HelloWorld")
```

Output:

"HelloWorld"

print() is a function which is used to print the values on to the output screen. It also has arguments, we can use it if needed. For example, in the above program, the output is printed with quotes by default we can remove it if needed.

```
print("HelloWorld", quote=FALSE)
```

Output:

HelloWorld

We passed the argument named quote in print function which value is set to be false to remove the quotes.

Different ways to run a R program

There are many ways to run an R program:

- **Method 1: Using command prompt or terminal**

Write your code in notepad or any text editor, save it as “helloworld.r”,



*helloworld.r - Notepad

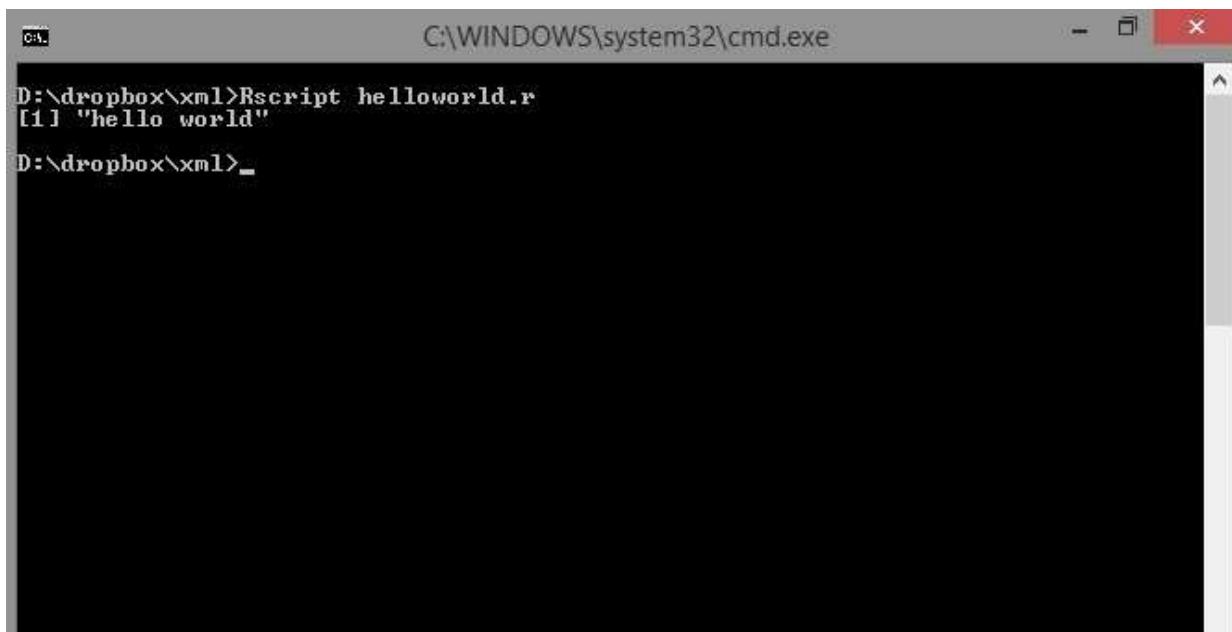
File Edit Format View Help

```
print("hello world")|
```

Ln 1, Col 21 100% Windows (CRLF) UTF-8

This screenshot shows a Microsoft Notepad window titled 'helloworld.r - Notepad'. The menu bar includes File, Edit, Format, View, and Help. The main text area contains the R command 'print("hello world")'. The status bar at the bottom shows 'Ln 1, Col 21', '100%', 'Windows (CRLF)', and 'UTF-8'.

Run it in command prompt or terminal using the command “Rscript helloworld.r”.



C:\WINDOWS\system32\cmd.exe

```
D:\dropbox\xml>Rscript helloworld.r
[1] "hello world"
D:\dropbox\xml>_
```

This screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'Rscript helloworld.r' was run, and the output '[1] "hello world"' was displayed. The prompt 'D:\dropbox\xml>' appears again at the end.

- **Method 2: Using an online IDE**

There are many online IDEs([click here](#)) available. We can use that without the need of installing or downloading anything.

- **Method 3: Using IDE like Rstudio, RTVS, StatET**

You can download and install these IDE in your system and can write and run the program there. Rstudio & statET(Eclipse software)is available for Windows, Mac, and Linux. RTVS presently available only on windows.

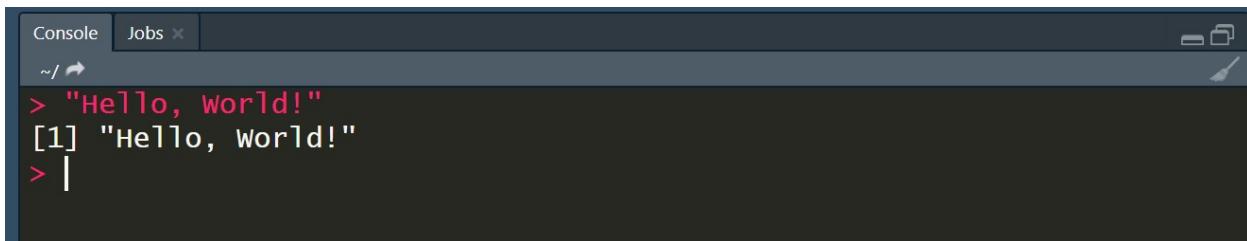
Some information about R commands

Basic Syntax in R Programming

R is the most popular language used for **Statistical Computing and Data Analysis** with the support of over 10, 000+ free packages in [CRAN](#) repository. Like any other programming language, R has a specific syntax which is important to understand if you want to make use of its powerful features. This article assumes R is already installed on your machine. We will be using RStudio but we can also use R command prompt by typing the following command in the command line.

```
$ R
```

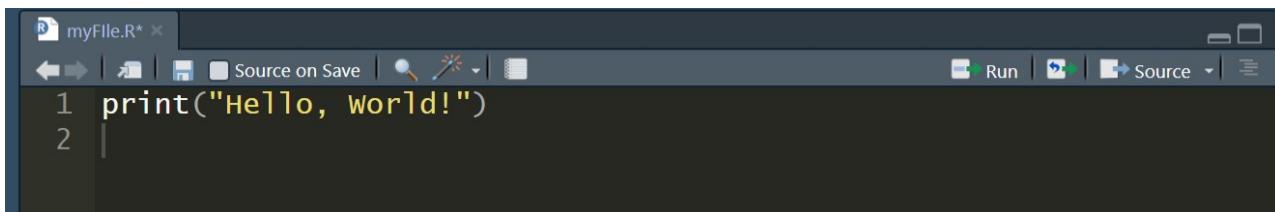
This will launch the interpreter and now let's write a basic Hello World program to get started.



```
Console Jobs ×
~/
> "Hello, world!"
[1] "Hello, world!"
> |
```

We can see that "Hello, World!" is being printed on the console. Now we can do the same thing using `print()` which prints to the console. Usually, we will write our code inside scripts which are called **RScripts** in R. To create one, write the below given code in a file and save it as **myFile.R** and then run it in console by writing:

```
Rscript myFile.R
```



```
myFile.R*
← → ⌂ Source on Save | ⚡ | ⌂
1 print("Hello, world!")
2 |
```

Output:

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

Syntax of R program

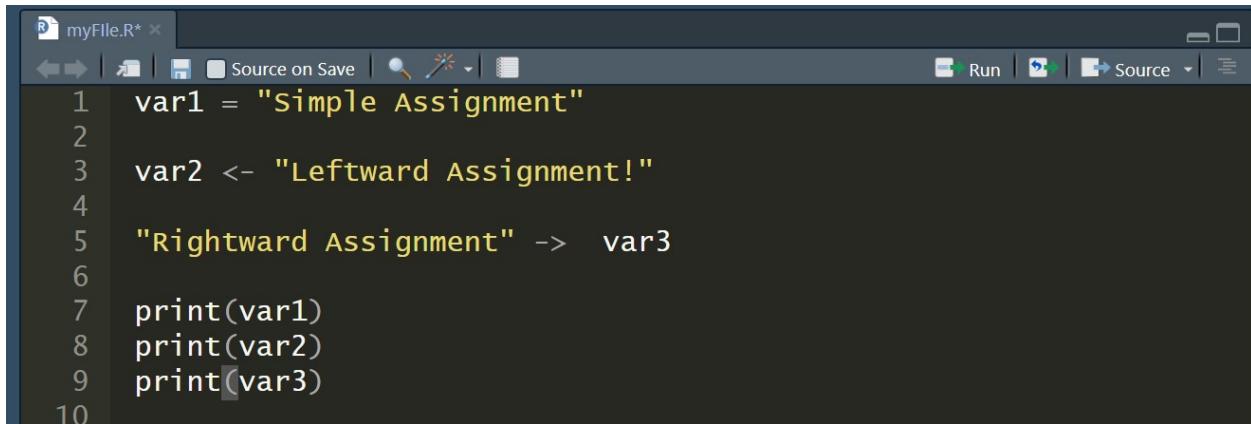
A program in R is made up of three things: Variables, Comments, and Keywords. Variables are used to store the data, Comments are used to improve code readability, and Keywords are reserved words that hold a specific meaning to the compiler.

Variables in R

Previously, we wrote all our code in a print() but we don't have a way to address them as to perform further operations. This problem can be solved by using **variables** which like any other programming language are the name given to reserved memory locations that can store any type of data. In R, the assignment can be denoted in three ways:

1. = (**Simple Assignment**)
2. <- (**Leftward Assignment**)
3. -> (**Rightward Assignment**)

Example:



```
myFile.R* 
Source on Save | Run | Source | 

1 var1 = "Simple Assignment"
2
3 var2 <- "Leftward Assignment!"
4
5 "Rightward Assignment" -> var3
6
7 print(var1)
8 print(var2)
9 print(var3)
10
```

Output:

"Simple Assignment"

"Leftward Assignment!"

"Rightward Assignment"

The rightward assignment is less common and can be confusing for some programmers, so it is generally recommended to use the <- or = operator for assigning values in R.

Comments in R

Comments are a way to improve your code's readability and are only meant for the user so the interpreter ignores it. Only single-line comments are available in R but we can also use multiline comments by using a simple trick which is shown below. Single line comments can be written by using # at the beginning of the statement. **Example:**

```
myFile.R* | Source on Save | Run | Source |
```

```
1 # This is a single line comment
2 print("This is fun!")
3
4 if(FALSE)
5 {
6   "This is multi-line comment which should be put inside either a
7     single or a double quote"
8 }
9
```

Output:

```
[1] "This is fun!"
```

From the above output, we can see that both comments were ignored by the interpreter.

Keywords in R

Keywords are the words reserved by a program because they have a special meaning thus a keyword can't be used as a variable name, function name, etc. We can view these keywords by using either `help(reserved)` or `?reserved`.

Reserved words in R

if	else	while	repeat	for
function	in	next	break	TRUE
FALSE	NULL	Inf	NaN	NA
NA_integer_	NA_real_	NA_complex_	NA_character_	...

- `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next` and `break` are used for control-flow statements and declaring user-defined functions.
- The ones left are used as constants like `TRUE`/`FALSE` are used as boolean constants.
- `NaN` defines Not a Number value and `NULL` are used to define an Undefined value.
- `Inf` is used for Infinity values.

Comments in R

Comments are generic English sentences, mostly written in a program to explain what it does or what a piece of code is supposed to do. More specifically, information that the programmer should be concerned with and it has nothing to do with the logic of the code.

All languages use a symbol to denote a comment and this symbol when encountered by the compiler helps it to differentiate between a comment and a statement.

Comments in R

In [R Programming Language](#) Comments are general English statements that are typically written in a program to describe what it does or what a piece of code is designed to perform. More precisely, information that should interest the coder and has nothing to do with the logic of the code. They are completely ignored by the compiler and are thus never reflected on the input.

The question arises here how will the compiler know whether the given statement is a comment or not? The answer is pretty simple.

Comments are generally used for the following purposes:

- Code Readability
- Explanation of the code or Metadata of the project
- Prevent execution of code
- To include resources

Types of Comments

There are generally three types of comments supported by languages, namely-

- **Single-line Comments**- Comment that only needs one line
- **Multi-line Comments**- Comment that requires more than one line.
- **Documentation Comments**- Comments that are drafted usually for a quick documentation lookup

Note: R doesn't support Multi-line and Documentation comments. It only supports single-line comments drafted by a '#' symbol.

Single-Line Comments in R

Single-line comments are comments that require only one line. They are usually drafted to explain what a single line of code does or what it is supposed to produce so that it can help someone refer to the source code. Just like Python single-line comments, any statement starting with “#” is a comment in R.

Syntax:

```
# comment statement
```

Example 1:

- R

```
# geeksforgeeks
```

The above code when executed will not produce any output, because R will consider the statement as a comment and hence the compiler will ignore the line.

Example 2:

- R

```
# R program to add two numbers
```

```
# Assigning values to variables
```

```
a <- 9
```

```
b <- 4
```

```
# Printing sum
```

```
print(a + b)
```

Output:

```
[1] 13
```

Multi-line Comments in R

As stated earlier that R doesn't support multi-lined comments, but to make the commenting process easier, R allows commenting on multiple single lines at once. There are two ways to add multiple single-line comments in R Studio:

- **First way:** Select the multiple lines on which you want to comment using the cursor and then use the key combination “**control + shift + C**” to comment or uncomment the selected lines.
- R

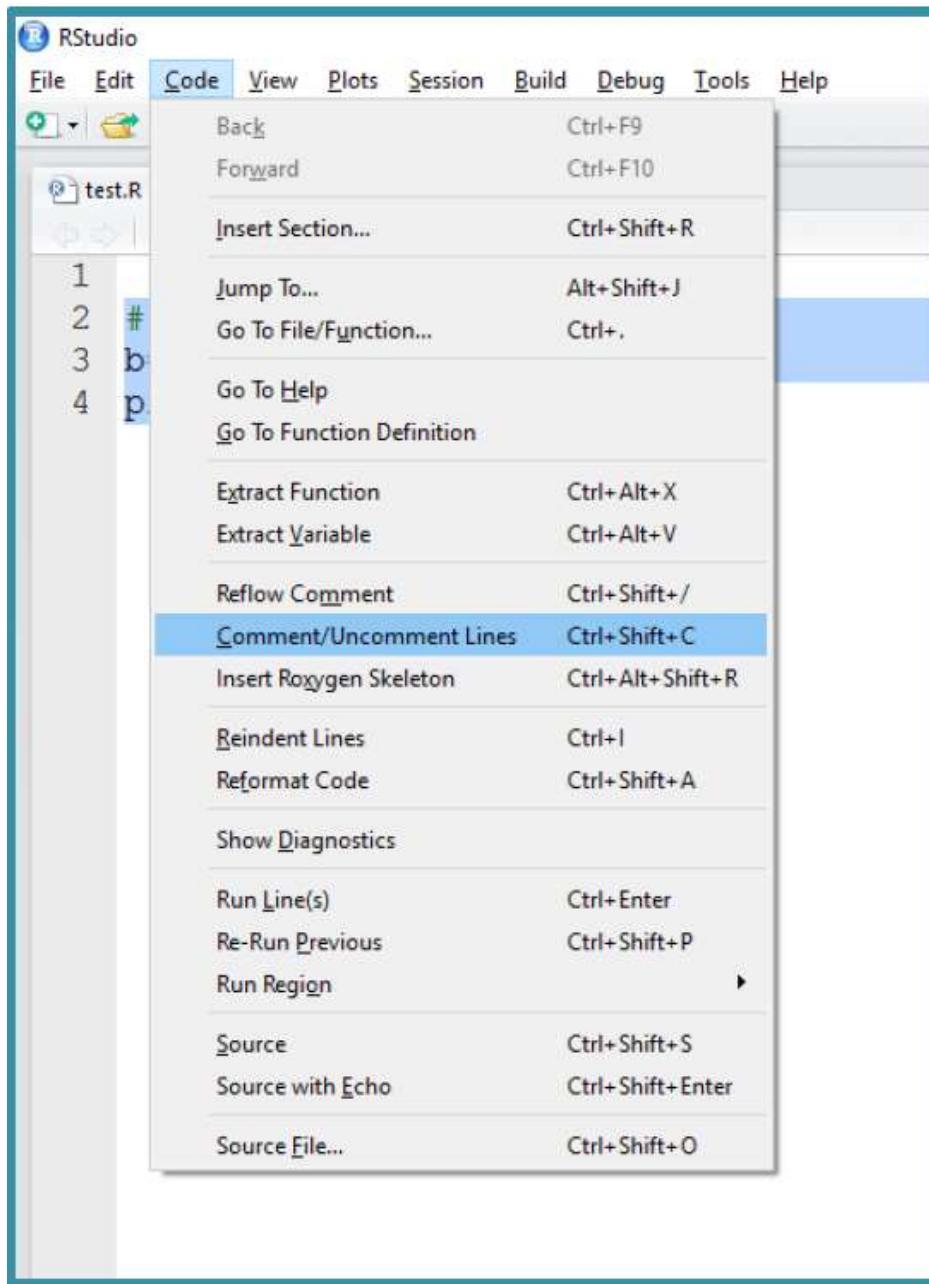
```
# This is a multiple-line comment  
  
# Each line starts with the '#' symbol  
  
# The following code will be executed  
  
x <- 10  
  
y <- 20  
  
sum <- x + y  
  
print(sum) # This line will print the value of 'sum'
```

Output:

[1] 30

- **Second way:** The other way is to **use the GUI**, select the lines on which you want to comment by using the cursor and click on “Code” in the menu, a pop-up window pops out in which we need to select “Comment/Uncomment Lines” which appropriately comments or uncomment the lines

which you have selected.



This makes the process of commenting a block of code easier and faster than adding # before each line one at a time.

R Operators

Operators are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numericals as input operands.

R Operators

R supports majorly four kinds of binary operators between a set of operands. In this article, we will see various types of **operators in R Programming language** and their usage.

Types of the operator in R language

- [Arithmetic Operators](#)
- [Logical Operators](#)
- [Relational Operators](#)
- [Assignment Operators](#)
- [Miscellaneous Operator](#)

Arithmetic Operators

Arithmetic operations in R simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors.

Addition operator (+)

The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:

- R

```
a <- c(1, 0.1)  
b <- c(2.33, 4)  
print(a+b)
```

Output : 3.33 4.10

Subtraction Operator (-)

The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:

- R

```
a <- 6  
b <- 8.4  
print (a-b)
```

Output : -2.4

Multiplication Operator (*)

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

- R

```
B= c(4,4)  
C= c(5,5)  
print (B*C)
```

Output : 20 20

Division Operator (/)

The first operand is divided by the second operand with the use of the '/' operator.

- R

```
a <- 10  
b <- 5  
print (a/b)
```

Output : 2

Power Operator (^)

The first operand is raised to the power of the second operand.

- R

```
a <- 4  
b <- 5  
print(a^b)
```

Output : 1024

Modulo Operator (%%)

The remainder of the first operand divided by the second operand is returned.

- R

```
list1<- c(2, 22)  
list2<-c(2,4)  
print(list1 %% list2)
```

Output : 0 2

The following R code illustrates the usage of all Arithmetic R operators.

- R

```
# R program to illustrate  
  
# the use of Arithmetic operators  
  
vec1 <- c(0, 2)  
vec2 <- c(2, 3)  
  
  
# Performing operations on Operands  
  
cat ("Addition of vectors :", vec1 + vec2, "\n")  
cat ("Subtraction of vectors :", vec1 - vec2, "\n")  
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
```

```
cat ("Division of vectors :", vec1 / vec2, "\n")
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Output

Addition of vectors : 2 5
Subtraction of vectors : -2 -1
Multiplication of vectors : 0 6
Division of vectors : 0 0.6666667
Modulo of vectors : 0 2
Power operator : 0 8

Logical Operators

Logical operations in R simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number.

Element-wise Logical AND operator (&)

Returns True if both the operands are True.

- R

```
list1 <- c(TRUE, 0.1)
list2 <- c(0,4+3i)
print(list1 & list2)
```

Output : FALSE TRUE

Any non zero integer value is considered as a TRUE value, be it complex or real number.

Element-wise Logical OR operator (|)

Returns True if either of the operands is True.

- R

```
list1 <- c(TRUE, 0.1)

list2 <- c(0,4+3i)

print(list1 | list2)
```

Output : TRUE TRUE

NOT operator (!)

A unary operator that negates the status of the elements of the operand.

- R

```
list1 <- c(0,FALSE)

print(!list1)
```

Output : TRUE TRUE

Logical AND operator (&&)

Returns True if both the first elements of the operands are True.

- R

```
list1 <- c(TRUE, 0.1)

list2 <- c(0,4+3i)

print(list1 && list2)
```

Output : FALSE

Compares just the first elements of both the lists.

Logical OR operator (||)

Returns True if either of the first elements of the operands is True.

- R

```
list1 <- c(TRUE, 0.1)
```

```
list2 <- c(0,4+3i)  
print(list1 | list2)
```

Output : TRUE

The following R code illustrates the usage of all Logical Operators in R:

- R

```
# R program to illustrate  
  
# the use of Logical operators  
  
vec1 <- c(0,2)  
  
vec2 <- c(TRUE,FALSE)  
  
  
# Performing operations on Operands  
  
cat ("Element wise AND :", vec1 & vec2, "\n")  
  
cat ("Element wise OR :", vec1 | vec2, "\n")  
  
cat ("Logical AND :", vec1 && vec2, "\n")  
  
cat ("Logical OR :", vec1 || vec2, "\n")  
  
cat ("Negation :", !vec1)
```

Output

Element wise AND : FALSE FALSE

Element wise OR : TRUE TRUE

Logical AND : FALSE

Logical OR : TRUE

Negation : TRUE FALSE

Relational Operators

The relational operators in R carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.

Less than (<)

Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(0, 0.1, "bat")
print(list1 < list2)
```

Output : FALSE FALSE TRUE

Less than equal to (<=)

Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(TRUE, 0.1, "bat")

# Convert lists to character strings
list1_char <- as.character(list1)
list2_char <- as.character(list2)

# Compare character strings
print(list1_char <= list2_char)
```

Output : TRUE TRUE TRUE

Greater than (>)

Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(TRUE, 0.1, "bat")
print(list1_char > list2_char)
```

Output : FALSE FALSE FALSE

Greater than equal to (>=)

Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(TRUE, 0.1, "bat")
print(list1_char >= list2_char)
```

Output : TRUE TRUE FALSE

Not equal to (!=)

Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1,'apple')
list2 <- c(0,0.1,"bat")
print(list1!=list2)
```

Output : TRUE FALSE TRUE

The following R code illustrates the usage of all Relational Operators in R:

- R

```
# R program to illustrate

# the use of Relational operators

vec1 <- c(0, 2)

vec2 <- c(2, 3)

# Performing operations on Operands

cat ("Vector1 less than Vector2 :", vec1 < vec2, "\n")

cat ("Vector1 less than equal to Vector2 :", vec1 <= vec2, "\n")

cat ("Vector1 greater than Vector2 :", vec1 > vec2, "\n")

cat ("Vector1 greater than equal to Vector2 :", vec1 >= vec2, "\n")

cat ("Vector1 not equal to Vector2 :", vec1 != vec2, "\n")
```

Output

Vector1 less than Vector2 : TRUE TRUE

Vector1 less than equal to Vector2 : TRUE TRUE

Vector1 greater than Vector2 : FALSE FALSE

Vector1 greater than equal to Vector2 : FALSE FALSE

Vector1 not equal to Vector2 : TRUE TRUE

Assignment Operators

Assignment operators in R are used to assigning values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right

Left Assignment (<- or <<- or =)

Assigns a value to a vector.

- R

```
vec1 = c("ab", TRUE)  
print (vec1)
```

Output : "ab" "TRUE"

Right Assignment (-> or ->>)

Assigns value to a vector.

- R

```
c("ab", TRUE) ->> vec1  
print (vec1)
```

Output : "ab" "TRUE"

The following R code illustrates the usage of all Relational Operators in R:

- R

```
# R program to illustrate  
  
# the use of Assignment operators  
  
vec1 <- c(2:5)  
  
c(2:5) ->> vec2  
  
vec3 <<- c(2:5)  
  
vec4 = c(2:5)  
  
c(2:5) -> vec5  
  
  
# Performing operations on Operands  
  
cat ("vector 1 :", vec1, "\n")
```

```
cat("vector 2 :", vec2, "\n")
cat ("vector 3 :", vec3, "\n")
cat("vector 4 :", vec4, "\n")
cat("vector 5 :", vec5)
```

Output

```
vector 1 : 2 3 4 5
vector 2 : 2 3 4 5
vector 3 : 2 3 4 5
vector 4 : 2 3 4 5
vector 5 : 2 3 4 5
```

Miscellaneous Operators

These are the mixed operators in R that simulate the printing of sequences and assignment of vectors, either left or right-handed.

%in% Operator

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

- R

```
val <- 0.1
list1 <- c(TRUE, 0.1, "apple")
print (val %in% list1)
```

Output : TRUE

Checks for the value 0.1 in the specified list. It exists, therefore, prints TRUE.

%*% Operator

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of the first matrix

must be equal to the number of rows of the second matrix. Multiplication of the matrix A with its transpose, B, produces a square matrix.

- R

```
mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)

print (mat)

print( t(mat))

pro = mat %*% t(mat)

print(pro)
```

Input :

Output :[,1] [,2] [,3] #original matrix of order 2x3

```
[1,] 1 3 5

[2,] 2 4 6

[,1] [,2]      #transposed matrix of order 3x2

[1,] 1 2

[2,] 3 4

[3,] 5 6

[,1] [,2]      #product matrix of order 2x2

[1,] 35 44

[2,] 44 56
```

The following R code illustrates the usage of all Miscellaneous Operators in R:

- R

```
# R program to illustrate

# the use of Miscellaneous operators
```

```
mat <- matrix (1:4, nrow = 1, ncol = 4)

print("Matrix elements using : ")

print(mat)

product = mat %*% t(mat)

print("Product of matrices")

print(product,)

cat ("does 1 exist in prod matrix :", "1" %in% product)
```

Output

```
[1] "Matrix elements using : "
```

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

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

```
[1] "Product of matrices"
```

```
[,1]
```

```
[1,] 30
```

```
does 1 exist in prod matrix : FALSE
```

R – Keywords

[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.

Keywords are specific reserved words in R, each of which has a specific feature associated with it. Almost all of the words which help one to use the functionality of the R language are included in the list of keywords. So one can imagine that the list of keywords is not going to be a small one! In R, one can view these keywords by using either **help(reserved)** or **?reserved**. Here is the list of keywords in R:

if

else

while

repeat

for

function

in

next

break

TRUE

FALSE

NULL

Inf

NaN
NA
NA_integer_
NA_real_
NA_complex_
NA_character_
...

Following are some most important keywords along with their examples:

- **if:** If statement is one of the Decision-making statements in the R programming language. It is one of the easiest decision-making statements. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Example:

```
# R program to illustrate if statement
```

```
# assigning value to variable a
```

```
a <- 5
```

```
# condition
```

```
if( a > 0 )
```

```
{  
  print("Positive Number") # Statement  
}
```

- **Output:**
- Positive Number
- **else:** It is similar to if statement but when the test expression in if condition fails, then statements in else condition are executed.

Example:

```
x <- 5  
  
# Check value is less than or greater than 10  
  
if(x > 10)  
{  
  print(paste(x, "is greater than 10"))  
}  
  
else  
{  
  print(paste(x, "is less than 10"))  
}
```

- **Output:**
- [1] "5 is less than 10"
- **while:** It is a type of control statement which will run a statement or a set of statements repeatedly unless the given condition becomes false. It is also an entry controlled loop, in this loop the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

Example:

```
# R program to demonstrate the use of while loop
```

```
val = 1
```

```
# using while loop
```

```
while (val <= 5 )
```

```
{
```

```
    # statements
```

```
    print(val)
```

```
    val = val + 1
```

```
}
```

- **Output:**

- [1] 1
- [1] 2
- [1] 3
- [1] 4
- [1] 5
- **repeat:** It is a simple loop that will run the same statement or a group of statements repeatedly until the stop condition has been encountered. Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop's body and use the declaration of a break statement to terminate this loop. If no condition is present in the body of the repeat loop then it will iterate infinitely.

Example:

```
# R program to demonstrate the use of repeat loop
```

```
val = 1

# using repeat loop

repeat

{

    # statements

    print(val)

    val = val + 1

    # checking stop condition

    if(val > 5)

    {

        # using break statement

        # to terminate the loop

        break

    }

}

}
```

- **Output:**
- [1] 1
- [1] 2
- [1] 3
- [1] 4
- [1] 5
- **for:** It is a type of control statement that enables one to easily construct a loop that has to run statements or a set of statements multiple times. For loop is commonly used to iterate over

items of a sequence. It is an entry controlled loop, in this loop the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

Example:

```
# R program to demonstrate the use of for loop

# using for loop

for (val in 1:5)

{

    # statement

    print(val)

}
```

- **Output:**
- [1] 1
- [1] 2
- [1] 3
- [1] 4
- [1] 5
- **function:** Functions are useful when you want to perform a certain task multiple number of times. In R functions are created using function keyword.

Example:

```
# A simple R function to check

# whether x is even or odd

evenOdd = function(x){
```

```
if(x %% 2 == 0)
  return("even")
else
  return("odd")
}
```

```
print(evenOdd(4))
print(evenOdd(3))
```

- **Output:**

- [1] "even"
- [1] "odd"
- **next:** Next statement in R is used to skip any remaining statements in the loop and continue the execution of the program. In other words, it is a statement that skips the current iteration without loop termination.

Example:

```
# R program to illustrate next in for loop
```

```
val <- 6:11
```

```
# Loop
```

```
for (i in val)
{
  if (i == 8)
  {
```

```
# test expression

next

}

print(i)

}
```

- **Output:**
- [1] 6
- [1] 7
- [1] 9
- [1] 10
- [1] 11
- **break:** The break keyword is a jump statement that is used to terminate the loop at a particular iteration.

Example:

```
# R Break Statement Example

a<-1

while (a < 10)

{

  print(a)

  if(a == 5)

    break

  a = a + 1

}
```

- **Output:**
- [1] 1

- [1] 2
- [1] 3
- [1] 4
- [1] 5
- **TRUE/FALSE:** The TRUE and FALSE keywords are used to represent a Boolean true and Boolean false. If the given statement is true, then the interpreter returns true else the interpreter returns false.

Example:

```
# A simple R program

# to illustrate TRUE / FALSE


# Sample values

x = 4

y = 3


# Comparing two values

z = x > y

p = x < y


# print the logical value

print(z)

print(p)
```

- **Output:**
- [1] TRUE
- [1] FALSE

- **NULL:** In R, NULL represents the null object. NULL is used to represent missing and undefined values. NULL is the logical representation of a statement which is neither TRUE nor FALSE.

Example:

```
# A simple R program

# to illustrate NULL


v = as.null(c(1, 2, 3, 4))

print(v)
```

- **Output:**

- NULL
- **Inf and NaN:** In R is.finite and is.infinite return a vector of the same length as x, where x is an R object to be tested. This indicating which elements are finite (not infinite and not missing) or infinite. Inf and -Inf keyword mean positive and negative infinity whereas NaN keyword means 'Not a Number'.

```
# A simple R program

# to illustrate Inf and NaN


# To check Inf

x = c(Inf, 2, 3)

print(is.finite(x))


# To check NaN

y = c(1, NaN, 3)

print(is.nan(y))
```

- **Output:**

- [1] FALSE TRUE TRUE

- [1] FALSE TRUE FALSE
- NA: NA stands for “Not Available” and is used to represent missing values. There are also constants NA_integer_, NA_real_, NA_complex_ and NA_character_ of the other atomic vector types which support missing values and all of these are reserved words in the R language.

```
# A simple R program
```

```
# to illustrate NA
```

```
# To check NA
```

```
x = c(1, NA, 2, 3)
```

```
print(is.na(x))
```

- **Output:**

- [1] FALSE TRUE FALSE FALSE

R Data Types

Different forms of data that can be saved and manipulated are defined and categorized using data types in computer languages, including R. Each R data type has unique properties and associated operations.

What are R Data types?

R Data types are used in computer programming to specify the kind of data that can be stored in a variable. For effective memory consumption and precise computation, the right data type must be selected. Each R data type has its own set of regulations and restrictions.

Data Types in R Programming Language

Each variable in R has an associated data type. Each R-Data Type requires different amounts of memory and has some specific operations which can be performed over it. [R Programming language](#) has the following basic R-data types and the following table shows the data type and the values that each data type can take.

Basic Data Types	Values	Examples
Numeric	Set of all real numbers	"numeric_value <- 3.14"

Basic Data Types	Values	Examples
Integer	Set of all integers, Z	"integer_value <- 42L"
Logical	TRUE and FALSE	"logical_value <- TRUE"
Complex	Set of complex numbers	"complex_value <- 1 + 2i"
Character	"a", "b", "c", ..., "@", "#", "\$",, "1", "2", ...etc	"character_value <- "Hello Geeks"
raw	as.raw()	"single_raw <- as.raw(255)"

Numeric Data type in R

Decimal values are called numerics in R. It is the default R data type for numbers in R. If you assign a decimal value to a variable x as follows, x will be of numeric type. Real numbers with a decimal point are represented using this data type in R. it uses a format for double-precision floating-point numbers to represent numerical values.

- R

```
# A simple R program

# to illustrate Numeric data type


# Assign a decimal value to x

x = 5.6


# print the class name of variable

print(class(x))
```

```
# print the type of variable  
print(typeof(x))
```

Output

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

```
[1] "double"
```

Even if an integer is assigned to a variable y, it is still saved as a numeric value.

- R

```
# A simple R program  
  
# to illustrate Numeric data type  
  
  
# Assign an integer value to y  
y = 5  
  
  
# print the class name of variable  
print(class(y))  
  
  
# print the type of variable  
print(typeof(y))
```

Output

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

```
[1] "double"
```

When R stores a number in a variable, it converts the number into a “double” value or a decimal type with at least two decimal places. This means that a value such as “5” here, is stored as 5.00 with a type

of double and a class of numeric. And also y is not an integer here can be confirmed with the `is.integer()` function.

- R

```
# A simple R program

# to illustrate Numeric data type


# Assign a integer value to y

y = 5


# is y an integer?

print(is.integer(y))
```

Output

[1] FALSE

Integer Data type in R

R supports integer data types which are the set of all integers. You can create as well as convert a value into an integer type using the `as.integer()` function. You can also use the capital 'L' notation as a suffix to denote that a particular value is of the integer R data type.

- R

```
# A simple R program

# to illustrate integer data type


# Create an integer value

x = as.integer(5)
```

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

# print the type of x
print(typeof(x))

# Declare an integer by appending an L suffix.
y = 5L

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

# print the type of y
print(typeof(y))
```

Output

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

Logical Data type in R

R has logical data types that take either a value of true or false. A logical value is often created via a comparison between variables. Boolean values, which have two possible values, are represented by this R data type: FALSE or TRUE

- R

```
# A simple R program  
# to illustrate logical data type
```

```
# Sample values
```

```
x = 4
```

```
y = 3
```

```
# Comparing two values
```

```
z = x > y
```

```
# print the logical value
```

```
print(z)
```

```
# print the class name of z
```

```
print(class(z))
```

```
# print the type of z
```

```
print(typeof(z))
```

Output

```
[1] TRUE
```

```
[1] "logical"
```

```
[1] "logical"
```

Complex Data type in R

R supports complex data types that are set of all the complex numbers. The complex data type is to store numbers with an imaginary component.

- R

```
# A simple R program

# to illustrate complex data type

# Assign a complex value to x

x = 4 + 3i

# print the class name of x

print(class(x))

# print the type of x

print(typeof(x))
```

Output

```
[1] "complex"

[1] "complex"
```

Character Data type in R

R supports character data types where you have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols. The easiest way to denote that a value is of character type in R data type is to wrap the value inside single or double inverted commas.

- R

```
# A simple R program

# to illustrate character data type
```

```
# Assign a character value to char  
char = "Geeksforgeeks"
```

```
# print the class name of char  
print(class(char))
```

```
# print the type of char  
print(typeof(char))
```

Output

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

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

There are several tasks that can be done using R data types. Let's understand each task with its action and the syntax for doing the task along with an R code to illustrate the task.

Raw data type in R

To save and work with data at the byte level in R, use the raw data type. By displaying a series of unprocessed bytes, it enables low-level operations on binary data. Here are some speculative data on R's raw data types:

- R

```
# Create a raw vector  
  
x <- as.raw(c(0x1, 0x2, 0x3, 0x4, 0x5))  
  
print(x)
```

Output:

```
[1] 01 02 03 04 05
```

Five elements make up this raw vector x, each of which represents a raw byte value.

Find data type of an object in R

To find the data type of an object you have to use **class()** function. The syntax for doing that is you need to pass the object as an argument to the function **class()** to find the data type of an object.

Syntax

```
class(object)
```

Example

- R

```
# A simple R program  
  
# to find data type of an object
```

```
# Logical  
  
print(class(TRUE))
```

```
# Integer  
  
print(class(3L))
```

```
# Numeric  
  
print(class(10.5))
```

```
# Complex  
  
print(class(1+2i))
```

```
# Character  
  
print(class("12-04-2020"))
```

Output

```
[1] "logical"
```

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

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

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

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

Type verification

To do that, you need to use the prefix “is.” before the data type as a command. The syntax for that is, `is.data_type()` of the object you have to verify.

Syntax:

```
is.data_type(object)
```

Example

- R

```
# A simple R program
```

```
# Verify if an object is of a certain datatype
```

```
# Logical
```

```
print(is.logical(TRUE))
```

```
# Integer
```

```
print(is.integer(3L))
```

```
# Numeric
```

```
print(is.numeric(10.5))
```

```
# Complex
```

```
print(is.complex(1+2i))
```

```
# Character  
  
print(is.character("12-04-2020"))  
  
print(is.integer("a"))  
  
print(is.numeric(2+3i))
```

Output

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

Coerce or convert the data type of an object to another

The process of altering the data type of an object to another type is referred to as coercion or data type conversion. This is a common operation in many programming languages that is used to alter data and perform various computations. When coercion is required, the language normally performs it automatically, whereas conversion is performed directly by the programmer.

Coercion can manifest itself in a variety of ways, depending on the [R programming language](#) and the context in which it is employed. In some circumstances, the coercion is implicit, which means that the language will change one type to another without the programmer having to expressly request it.

Syntax

```
as.data_type(object)
```

Note: All the coercions are not possible and if attempted will be returning an “NA” value.

Example

- R

```
# A simple R program

# convert data type of an object to another


# Logical

print(as.numeric(TRUE))

# Integer

print(as.complex(3L))

# Numeric

print(as.logical(10.5))

# Complex

print(as.character(1+2i))

# Can't possible

print(as.numeric("12-04-2020"))
```

Output

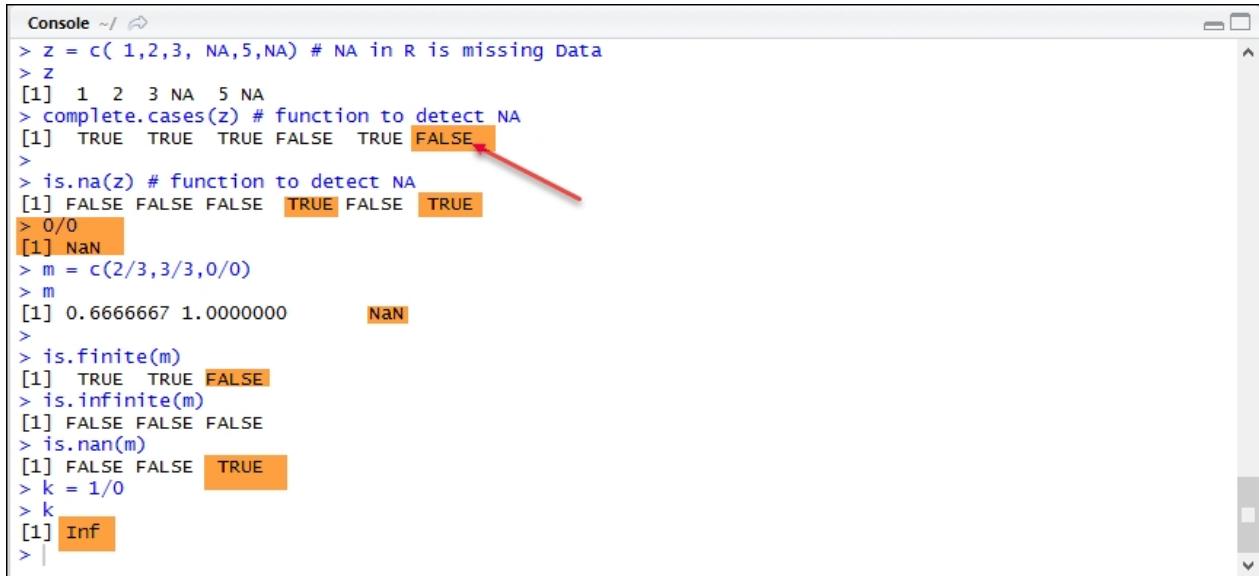
```
[1] 1
[1] 3+0i
[1] TRUE
[1] "1+2i"
[1] NA
```

Warning message:

```
In print(as.numeric("12-04-2020")) : NAs introduced by coercion
```

Special values in R

R comes with some special values. Some of the special values in R are NA, Inf, -Inf, and NaN.



The screenshot shows an R console window with the following session history:

```
Console ~/ 
> z = c( 1,2,3, NA,5,NA) # NA in R is missing Data
> z
[1] 1 2 3 NA 5 NA
> complete.cases(z) # function to detect NA
[1] TRUE TRUE TRUE FALSE TRUE FALSE
>
> is.na(z) # function to detect NA
[1] FALSE FALSE FALSE TRUE FALSE TRUE
> 0/0
[1] NaN
> m = c(2/3,3/3,0/0)
> m
[1] 0.6666667 1.0000000      NaN
>
> is.finite(m)
[1] TRUE TRUE FALSE
> is.infinite(m)
[1] FALSE FALSE FALSE
> is.nan(m)
[1] FALSE FALSE TRUE
> k = 1/0
> k
[1] Inf
> |
```

A red arrow points from the text "R comes with some special values. Some of the special values in R are NA, Inf, -Inf, and NaN." to the word "FALSE" in the console output, which is highlighted in orange.

How to do it...

The missing values are represented in R by NA. When we download data, it may have missing data and this is represented in R by NA:

```
z = c( 1,2,3, NA,5,NA) # NA in R is missing Data
```

To detect missing values, we can use the `install.packages()` function or `is.na()`, as shown:

```
complete.cases(z) # function to detect NA
```

```
is.na(z) # function to detect NA
```

To remove the NA values from our data, we can type the following in our active R session console window:

```
clean <- complete.cases(z)
```

```
z[clean] # used to remove NA from data
```

Please note the use of square brackets ([]) instead of parentheses.

In R, not a number is abbreviated as NaN. The following lines will generate NaN values:

```
##NaN  
0/0  
m <- c(2/3,3/3,0/0)  
m
```

The is.finite, is.infinite, or is.nan functions will generate logical values (TRUE or FALSE).

```
is.finite(m)  
is.infinite(m)  
is.nan(m)
```

The following line will generate inf as a special value in R:

```
## infinite  
k = 1/0
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

How it works...

`complete.cases(z)` is a logical vector indicating complete cases that have no missing value (NA). On the other hand, `is.na(z)` indicates which elements are missing. In both cases, the argument is our data, a vector, or a matrix.

```

Console ~/ 
> z = c( 1,2,3, NA,5,NA)
> complete.cases(z)
[1] TRUE TRUE TRUE FALSE TRUE FALSE
> is.na(z)
[1] FALSE FALSE FALSE TRUE FALSE TRUE
> dk = c(1,45,67,20)
> anyNA(dk)
[1] FALSE
> dk[3] = NA
> dk
[1] 1 45 NA 20
> anyNA(dk)
[1] TRUE
> |

```

R also allows its users to check if any element in a matrix or a vector is NA by using the anyNA() function. We can coerce or assign NA to any element of a vector using the square brackets ([]). The [3] input instructs R to assign NA to the third element of the dk vector.

Handling Missing Values in R Programming

As the name indicates, Missing values are those elements that are not known. NA or NaN are reserved words that indicate a missing value in [R Programming language](#) for arithmetic operations that are undefined.

R – Handling Missing Values

Missing values are practical in life. For example, some cells in spreadsheets are empty. If an insensible or impossible arithmetic operation is tried then NAs occur.

Dealing Missing Values in R

Missing Values in R, are handled with the use of some pre-defined functions:

is.na() Function for Finding Missing values:

A logical vector is returned by this function that indicates all the NA values present. It returns a Boolean value. If NA is present in a vector it returns TRUE else FALSE.

```

x<- c(NA, 3, 4, NA, NA, NA)

is.na(x)

```

Output:

[1] TRUE FALSE FALSE TRUE TRUE TRUE

Properties of Missing Values:

- For testing objects that are NA use `is.na()`
- For testing objects that are NaN use `is.nan()`
- There are classes under which NA comes. Hence integer class has integer type NA, the character class has character type NA, etc.
- A NaN value is counted in NA but the reverse is not valid.

The creation of a vector with one or multiple NAs is also possible.

```
x<- c(NA, 3, 4, NA, NA, NA)
```

```
x
```

Output:

```
[1]NA 3 4 NA NA NA
```

Removing NA or NaN values

There are two ways to remove missing values:

Extracting values except for NA or NaN values:

Example 1:

- R

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

```
d <- is.na(x)
```

```
x[! d]
```

Output:

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

Example 2:

- R

```
x <- c(1, 2, 0 / 0, 3, NA, 4, 0 / 0)

x

x[! is.na(x)]
```

Output:

```
[1] 1 2 NaN 3 NA 4 NaN
```

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

A function called **complete.cases()** can also be used. This function also works on data frames.

Missing Value Filter Functions

The modeling functions in R language acknowledge a **na.action** argument which provides instructions to the function regarding its response if NA comes in its way.

And hence this way the function calls one of the missing value filter functions. Missing Value Filter Functions alter the data set and in the new data set the value of NAs has been changed. The default Missing Value Filter Function is **na.omit**. It omits every row containing even one NA. Some other Missing Value Filter Functions are:

- **na.omit**— omits every row containing even one NA
- **na.fail**— halts and does not proceed if NA is encountered
- **na.exclude**— excludes every row containing even one NA but keeps a record of their original position
- **na.pass**— it just ignores NA and passes through it
- R

```
# Creating a data frame

df <- data.frame (c1 = 1:8,

c2 = factor (c("B", "A", "B", "C",

"A", "C", "B", "A")))
```

```

# Filling some NA in data frame

df[4, 1] <- df[6, 2] <- NA


# Printing all the levels(NA is not considered one)

levels(df$c2)


# fails if NA is encountered

na.fail (df)


# excludes every row containing even one NA

na.exclude (a)

```

Output:

```

[1]"A""B""C"
Error in na.fail.default(df) : missing values in object
Calls: na.fail -> na.fail.default
Execution halted

```

Find and Remove NA or NaN values from a dataset

In R we can remove and find missing values from the entire dataset. there are some main functions we can use and perform the tasks.

First, we will create one data frame and then we will find and remove all the missing values which are present in the data.

- R

```

# Create a data frame with 5 rows and 3 columns

data <- data.frame(
  A = c(1, 2, NA, 4, 5),

```

```
B = c(NA, 2, 3, NA, 5),  
C = c(1, 2, 3, NA, NA)  
)  
  
# View the resulting data frame  
data
```

Output:

```
A B C  
1 1 NA 1  
2 2 2 2  
3 NA 3 3  
4 4 NA NA  
5 5 5 NA
```

Find all the missing values in the data

- R

```
# Finding missing values in data.  
sum(is.na(data))
```

Output:

```
[1] 5
```

Find all the missing values in the columns

- R

```
# Finding missing values column wise  
colSums(is.na(data))
```

Output:

A	B	C
1	2	2

Visualization of missing values of a dataset

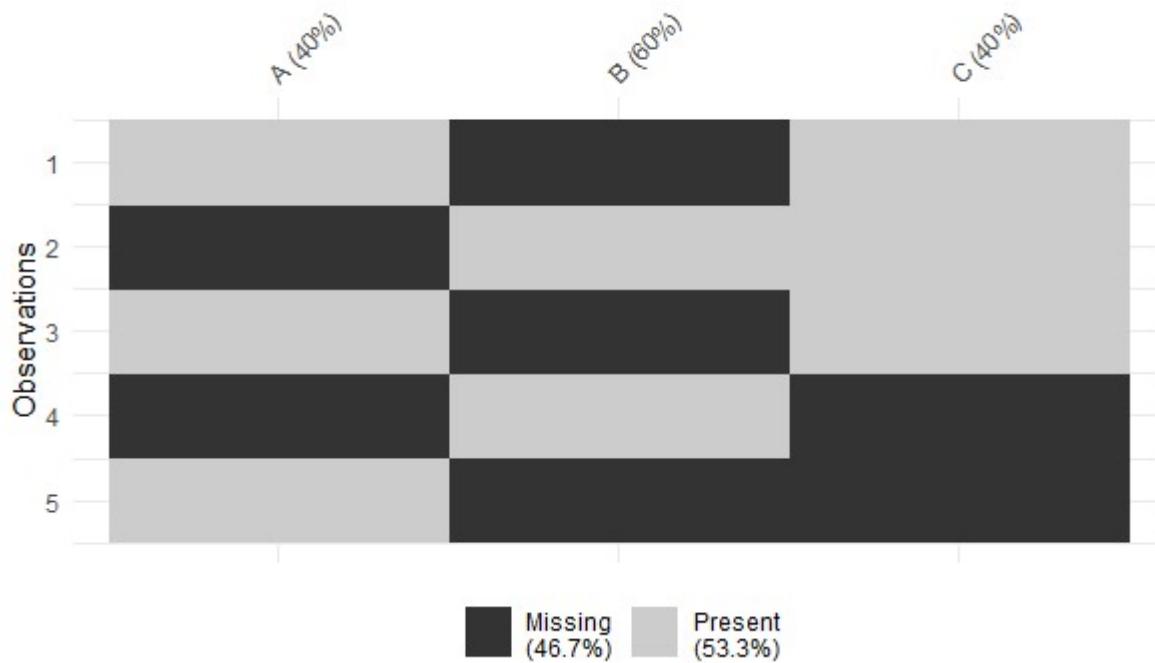
- R

```
# Install and load the 'visdat' package
install.packages("visdat")
library(visdat)

# Create a data frame with missing values
data <- data.frame(
  A = c(1, NA, 3, NA, 5),
  B = c(NA, 2, NA, 4, NA),
  C = c(1, 2, 3, NA, NA)
)

# Plot the missing value diagram
vis_miss(data)
```

Output:



Handling missing values in R

Remove missing values from dataframe

- R

```
# Remove missing values using na.omit function.
```

```
data<- na.omit(data)
```

```
data
```

Output:

```
A B C
2 2 2
```

Special Cases

There are two special cases where NA is denoted or presented differently:

- **Factor Vectors**– is the symbol displayed in factor vectors for missing values.

- **NaN** – This is a special case of NA only. It is displayed when an arithmetic operation yields a result that is not a number. For example, dividing zero by zero produces NaN.

R – Objects

Every programming language has its own data types to store values or any information so that the user can assign these data types to the variables and perform operations respectively. Operations are performed accordingly to the data types.

These data types can be character, integer, float, long, etc. Based on the data type, memory/storage is allocated to the variable. For example, in C language character variables are assigned with 1 byte of memory, integer variable with 2 or 4 bytes of memory and other data types have different memory allocation for them.

Unlike other programming languages, variables are assigned to objects rather than data types in [R programming](#).

Type of Objects

There are 5 basic types of objects in the R language:

Vectors

Atomic [vectors](#) are one of the basic types of objects in R programming. Atomic vectors can store homogeneous data types such as character, doubles, integers, raw, logical, and complex. A single element variable is also said to be vector.

Example:

```
# Create vectors

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

y <- c("a", "b", "c", "d")

z <- 5

# Print vector and class of vector

print(x)

print(class(x))
```

```
print(y)  
print(class(y))
```

```
print(z)  
print(class(z))
```

Output:

```
[1] 1 2 3 4  
[1] "numeric"  
[1] "a" "b" "c" "d"  
[1] "character"  
[1] 5  
[1] "numeric"
```

Lists

[List](#) is another type of object in R programming. List can contain heterogeneous data types such as vectors or another lists.

Example:

```
# Create list  
  
ls <- list(c(1, 2, 3, 4), list("a", "b", "c"))  
  
  
# Print  
  
print(ls)  
print(class(ls))
```

Output:

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

```
[[2]]  
[[2]][[1]]  
[1] "a"  
  
[[2]][[2]]  
[1] "b"
```

```
[[2]][[3]]  
[1] "c"  
  
[1] "list"
```

Matrices

To store values as 2-Dimensional array, matrices are used in R. Data, number of rows and columns are defined in the `matrix()` function.

Syntax:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Example:

```
x <- c(1, 2, 3, 4, 5, 6)  
  
# Create matrix  
mat <- matrix(x, nrow = 2)
```

```
print(mat)  
print(class(mat))
```

Output:

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

```
[1] "matrix"
```

Factors

[Factor](#) object encodes a vector of unique elements (levels) from the given data vector.

Example:

```
# Create vector  
  
s <- c("spring", "autumn", "winter", "summer",  
"spring", "autumn")  
  
  
print(factor(s))  
print(nlevels(factor(s)))
```

Output:

```
[1] spring autumn winter summer spring autumn
```

```
Levels: autumn spring summer winter
```

```
[1] 4
```

Arrays

[array\(\)](#) function is used to create n-dimensional array. This function takes dim attribute as an argument and creates required length of each dimension as specified in the attribute.

Syntax:

```
array(data, dim = length(data), dimnames = NULL)
```

Example:

```
# Create 3-dimensional array  
# and filling values by column  
  
arr <- array(c(1, 2, 3), dim = c(3, 3, 3))  
  
print(arr)
```

Output:

```
,, 1
```

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

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

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

Data Frames

[Data frames](#) are 2-dimensional tabular data object in R programming. Data frames consists of multiple columns and each column represents a vector. Columns in data frame can have different modes of data unlike matrices.

Example:

```
# Create vectors  
  
x <- 1:5  
  
y <- LETTERS[1:5]  
  
z <- c("Albert", "Bob", "Charlie", "Denver", "Elie")  
  
  
# Create data frame of vectors  
  
df <- data.frame(x, y, z)  
  
  
# Print data frame  
  
print(df)
```

Output:

```
x y     z  
1 1 A Albert  
2 2 B Bob  
3 3 C Charlie  
4 4 D Denver  
5 5 E Elie
```

Functions in R Programming

Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In [R Programming Language](#) when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more functions in R.

Table of Content

- [Function in R Programming](#)
- [Types of Function in R Language](#)
- [Built-in Function in R Programming Language](#)
- [User-defined Functions in R Programming Language](#)
- [R Function Example](#)

Function in R Programming

Functions are created in R by using the command **function()**. The general structure of the function file is as follows:

```
f = function(arguments){  
    statements  
}
```

Here **f** = function name

Functions in R Programming

Note: In the above syntax f is the function name, this means that you are creating a function with name f which takes certain arguments and executes the following statements.

Types of Function in R Language

1. **Built-in Function:** Built-in functions in R are pre-defined functions that are available in [R programming languages](#) to perform common tasks or operations.
2. **User-defined Function:** R language allow us to write our own function.

Built-in Function in R Programming Language

Here we will use built-in functions like sum(), max() and min().

- R

```
# Find sum of numbers 4 to 6.
```

```

print(sum(4:6))

# Find max of numbers 4 and 6.

print(max(4:6))

# Find min of numbers 4 and 6.

print(min(4:6))

```

Output

```

[1] 15

[1] 6

[1] 4

```

Other Built-in Functions in R

Functions	Syntax
Mathematical Functions	
a. <u>abs()</u>	calculates a number's absolute value.
b. <u>sqrt()</u>	calculates a number's square root.
c. <u>round()</u>	rounds a number to the nearest integer.
d. <u>exp()</u>	calculates a number's exponential value
e. <u>log()</u>	which calculates a number's natural logarithm.

Functions	Syntax
f. cos() , sin() , and tan()	calculates a number's cosine, sine, and tangent.
Statistical Functions	
a. mean()	A vector's arithmetic mean is determined by the mean() function.
b. median()	A vector's median value is determined by the median() function.
c. cor()	calculates the correlation between two vectors.
d. var()	calculates the variance of a vector and calculates the standard deviation of a vector.
Data Manipulation Functions	
a. unique()	returns the unique values in a vector.
b. subset()	subsets a data frame based on conditions.
c. aggregate()	groups data according to a grouping variable.
d. order()	uses ascending or descending order to sort a vector.

Functions	Syntax
File Input/Output Functions	
a. <u>read.csv()</u>	reads information from a CSV file.
b. <u>Write.csv()</u>	publishes information to write a CSV file.
c. <u>Read.table()</u>	reads information from a tabular.
d. <u>Write.table()</u>	creates a tabular file with data.

User-defined Functions in R Programming Language

R provides built-in functions like **print()**, **cat()**, etc. but we can also create our own functions. These functions are called user-defined functions.

Example

- R

```
# A simple R function to check
# whether x is even or odd

evenOdd = function(x){
  if(x %% 2 == 0)
    return("even")
  else
    return("odd")
}
```

```
print(evenOdd(4))  
print(evenOdd(3))
```

Output

[1] "even"

[1] "odd"

R Function Example

Single Input Single Output

Now create a function in R that will take a single input and gives us a single output.

Following is an example to create a function that calculates the area of a circle which takes in the arguments the radius. So, to create a function, name the function as “areaOfCircle” and the arguments that are needed to be passed are the “radius” of the circle.

- R

```
# A simple R function to calculate
```

```
# area of a circle
```

```
areaOfCircle = function(radius){
```

```
    area = pi*radius^2
```

```
    return(area)
```

```
}
```

```
print(areaOfCircle(2))
```

Output

12.56637

Multiple Input Multiple Output

Now create a function in R Language that will take multiple inputs and gives us multiple outputs using a list.

The functions in R Language take multiple input objects but returned only one object as output, this is, however, not a limitation because you can create lists of all the outputs which you want to create and once the list is created you can access them into the elements of the list and get the answers which you want.

Let us consider this example to create a function “Rectangle” which takes “length” and “width” of the rectangle and returns area and perimeter of that rectangle. Since R Language can return only one object. Hence, create one object which is a list that contains “area” and “perimeter” and return the list.

- R

```
# A simple R function to calculate  
  
# area and perimeter of a rectangle  
  
Rectangle = function(length, width){  
  
  area = length * width  
  
  perimeter = 2 * (length + width)  
  
  # create an object called result which is  
  
  # a list of area and perimeter  
  
  result = list("Area" = area, "Perimeter" = perimeter)  
  
  return(result)  
  
}  
  
  
resultList = Rectangle(2, 3)  
  
print(resultList["Area"])  
  
print(resultList["Perimeter"])
```

Output

```
$Area
```

```
[1] 6
```

```
$Perimeter
```

```
[1] 10
```

Inline Functions in R Programming Language

Sometimes creating an R script file, loading it, executing it is a lot of work when you want to just create a very small function. So, what we can do in this kind of situation is an inline function.

To create an inline function you have to use the function command with the argument x and then the expression of the function.

Example

- R

```
# A simple R program to  
  
# demonstrate the inline function  
  
f = function(x) x^2*4+x/3  
  
  
print(f(4))  
print(f(-2))  
print(0)
```

Output

```
65.33333
```

```
15.33333
```

```
0
```

Passing Arguments to Functions in R Programming Language

There are several ways you can pass the arguments to the function:

- **Case 1:** Generally in R, the arguments are passed to the function in the same order as in the function definition.
- **Case 2:** If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
- **Case 3:** If the arguments are not passed the default values are used to execute the function.

Now, let us see the examples for each of these cases in the following R code:

- R

```
# A simple R program to demonstrate

# passing arguments to a function

Rectangle = function(length=5, width=4){

  area = length * width

  return(area)

}

# Case 1:

print(Rectangle(2, 3))

# Case 2:

print(Rectangle(width = 8, length = 4))

# Case 3:

print(Rectangle())
```

Output

32

20

Lazy Evaluations of Functions in R Programming Language

In R the functions are executed in a lazy fashion. When we say lazy what it means is if some arguments are missing the function is still executed as long as the execution does not involve those arguments.

Example

In the function “Cylinder” given below. There are defined three-argument “diameter”, “length” and “radius” in the function and the volume calculation does not involve this argument “radius” in this calculation. Now, when you pass this argument “diameter” and “length” even though you are not passing this “radius” the function will still execute because this radius is not used in the calculations inside the function.

Let's illustrate this in an R code given below:

- R

```
# A simple R program to demonstrate

# Lazy evaluations of functions

Cylinder = function(diameter, length, radius ){
  volume = pi*diameter^2*length/4
  return(volume)
}

# This'll execute because this
# radius is not used in the
# calculations inside the function.

print(Cylinder(5, 10))
```

Output

196.3495

If you do not pass the argument and then use it in the definition of the function it will throw an error that this “radius” is not passed and it is being used in the function definition.

Example

- R

```
# A simple R program to demonstrate

# Lazy evaluations of functions

Cylinder = function(diameter, length, radius ){

  volume = pi*diameter^2*length/4

  print(radius)

  return(volume)

}

# This'll throw an error

print(Cylinder(5, 10))
```

Output

Error in print(radius) : argument "radius" is missing, with no default

Simple manipulations; numbers and vectors

2.1 Vectors and assignment

R operates on named *data structures*. The simplest such structure is the numeric *vector*, which is a single entity consisting of an ordered collection of numbers. To set up a vector named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

This is an *assignment* statement using the *function* `c()` which in this context can take an arbitrary number of vector *arguments* and whose value is a vector got by concatenating its arguments end to end.¹

 ¹ With other than vector types of argument, such as list mode arguments, the action of `c()` is rather different. See [Concatenating lists](#).

A number occurring by itself in an expression is taken as a vector of length one.

Notice that the assignment operator (`<-`), which consists of the two characters `<` (“less than”) and `-` (“minus”) occurring strictly side-by-side and it ‘points’ to the object receiving the value of the expression. In most contexts the `=` operator can be used as an alternative.

Assignment can also be made using the function `assign()`. An equivalent way of making the same assignment as above is with:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

The usual operator, `<-`, can be thought of as a syntactic short-cut to this.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

If an expression is used as a complete command, the value is printed *and lost*². So now if we were to use the command

 ² Actually, it is still available as `.Last.value` before any other statements are executed.

```
> 1/x
```

the reciprocals of the five values would be printed at the terminal (and the value of `x`, of course, unchanged).

The further assignment

```
> y <- c(x, 0, x)
```

would create a vector `y` with 11 entries consisting of two copies of `x` with a zero in the middle place.

2.2 Vector arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are *recycled* as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
> v <- 2*x + y + 1
```

generates a new vector v of length 11 constructed by adding together, element by element, $2*x$ repeated 2.2 times, y repeated just once, and 1 repeated 11 times.

The elementary arithmetic operators are the usual $+$, $-$, $*$, $/$ and $^$ for raising to a power. In addition all of the common arithmetic functions are available. \log , \exp , \sin , \cos , \tan , $\sqrt{}$, and so on, all have their usual meaning. \max and \min select the largest and smallest elements of a vector respectively. range is a function whose value is a vector of length two, namely $c(\min(x), \max(x))$. $\text{length}(x)$ is the number of elements in x , $\text{sum}(x)$ gives the total of the elements in x , and $\text{prod}(x)$ their product.

Two statistical functions are $\text{mean}(x)$ which calculates the sample mean, which is the same as $\text{sum}(x)/\text{length}(x)$, and $\text{var}(x)$ which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance. If the argument to $\text{var}()$ is an n -by- p matrix the value is a p -by- p sample covariance matrix got by regarding the rows as independent p -variate sample vectors.

$\text{sort}(x)$ returns a vector of the same size as x with the elements arranged in increasing order; however there are other more flexible sorting facilities available (see $\text{order}()$ or $\text{sort.list}()$ which produce a permutation to do the sorting).

Note that \max and \min select the largest and smallest values in their arguments, even if they are given several vectors. The *parallel* maximum and minimum functions pmax and pmin return a vector (of length equal to their longest argument) that contains in each element the largest (smallest) element in that position in any of the input vectors.

For most purposes the user will not be concerned if the “numbers” in a numeric vector are integers, reals or even complex. Internally calculations are done as double precision real numbers, or double precision complex numbers if the input data are complex.

To work with complex numbers, supply an explicit complex part. Thus

```
sqrt(-17)
```

will give NaN and a warning, but

```
sqrt(-17+0i)
```

will do the computations as complex numbers.

2.3 Generating regular sequences

R has a number of facilities for generating commonly used sequences of numbers. For example $1:30$ is the vector $c(1, 2, \dots, 29, 30)$. The colon operator has high priority within an expression, so, for example $2*1:15$ is the vector $c(2, 4, \dots, 28, 30)$. Put $n <- 10$ and compare the sequences $1:n-1$ and $1:(n-1)$.

The construction $30:1$ may be used to generate a sequence backwards.

The function `seq()` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is `seq(2,10)` is the same vector as `2:10`.

Arguments to `seq()`, and to many other R functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two arguments may be named `from=value` and `to=value`; thus `seq(1,30)`, `seq(from=1, to=30)` and `seq(to=30, from=1)` are all the same as `1:30`. The next two arguments to `seq()` may be named `by=value` and `length=value`, which specify a step size and a length for the sequence respectively. If neither of these is given, the default `by=1` is assumed.

For example

```
> seq(-5, 5, by=.2) -> s3
```

generates in `s3` the vector `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)`. Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

generates the same vector in `s4`.

The fifth argument may be named `along=vector`, which is normally used as the only argument to create the sequence `1, 2, ..., length(vector)`, or the empty sequence if the vector is empty (as it can be).

A related function is `rep()` which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of `x` end-to-end in `s5`. Another useful version is

```
> s6 <- rep(x, each=5)
```

which repeats each element of `x` five times before moving on to the next.

2.4 Logical vectors

As well as numerical vectors, R allows manipulation of logical quantities. The elements of a logical vector can have the values `TRUE`, `FALSE`, and `NA` (for “not available”, see below). The first two are often abbreviated as `T` and `F`, respectively. Note however that `T` and `F` are just variables which are set to `TRUE` and `FALSE` by default, but are not reserved words and hence can be overwritten by the user. Hence, you should always use `TRUE` and `FALSE`.

Logical vectors are generated by *conditions*. For example

```
> temp <- x > 13
```

sets temp as a vector of the same length as x with values FALSE corresponding to elements of x where the condition is *not* met and TRUE where it is.

The logical operators are <, <=, >, >=, == for exact equality and != for inequality. In addition if c1 and c2 are logical expressions, then c1 & c2 is their intersection (“*and*”), c1 | c2 is their union (“*or*”), and !c1 is the negation of c1.

Logical vectors may be used in ordinary arithmetic, in which case they are *coerced* into numeric vectors, FALSE becoming 0 and TRUE becoming 1. However there are situations where logical vectors and their coerced numeric counterparts are not equivalent, for example see the next subsection.

2.5 Missing values

In some cases the components of a vector may not be completely known. When an element or value is “not available” or a “missing value” in the statistical sense, a place within a vector may be reserved for it by assigning it the special value NA. In general any operation on an NA becomes an NA. The motivation for this rule is simply that if the specification of an operation is incomplete, the result cannot be known and hence is not available.

The function is.na(x) gives a logical vector of the same size as x with value TRUE if and only if the corresponding element in x is NA.

```
> z <- c(1:3,NA); ind <- is.na(z)
```

Notice that the logical expression x == NA is quite different from is.na(x) since NA is not really a value but a marker for a quantity that is not available. Thus x == NA is a vector of the same length as x *all* of whose values are NA as the logical expression itself is incomplete and hence undecidable.

Note that there is a second kind of “missing” values which are produced by numerical computation, the so-called *Not a Number*, NaN, values. Examples are

```
> 0/0
```

or

```
> Inf - Inf
```

which both give NaN since the result cannot be defined sensibly.

In summary, is.na(xx) is TRUE *both* for NA and NaN values. To differentiate these, is.nan(xx) is only TRUE for NaNs.

Missing values are sometimes printed as <NA> when character vectors are printed without quotes.

2.6 Character vectors

Character quantities and character vectors are used frequently in R, for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character, e.g., "x-values", "New iteration results".

Character strings are entered using either matching double ("") or single () quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using \ as the escape character, so \ is entered and printed as \\, and inside double quotes " is entered as \". Other useful escape sequences are \n, newline, \t, tab and \b, backspace—see ?Quotes for a full list.

Character vectors may be concatenated into a vector by the c() function; examples of their use will emerge frequently.

The paste() function takes an arbitrary number of arguments and concatenates them one by one into character strings. Any numbers given among the arguments are coerced into character strings in the evident way, that is, in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named argument, sep=string, which changes it to string, possibly empty.

For example

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

makes labs into the character vector

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Note particularly that recycling of short lists takes place here too; thus c("X", "Y") is repeated 5 times to match the sequence 1:10. ³:

¶ ³ paste(..., collapse=ss) joins the arguments into a single character string putting ss in between, e.g., ss <- "|". There are more tools for character manipulation, see the help for sub and substring.

2.7 Index vectors; selecting and modifying subsets of a data set

Subsets of the elements of a vector may be selected by appending to the name of the vector an *index vector* in square brackets. More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.

Such index vectors can be any of four distinct types.

1. **A logical vector.** In this case the index vector is recycled to the same length as the vector from which elements are to be selected. Values corresponding to TRUE in the index vector are selected and those corresponding to FALSE are omitted. For example

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

creates (or re-creates) an object *y* which will contain the non-missing values of *x*, in the same order. Note that if *x* has missing values, *y* will be shorter than *x*. Also

```
> (x+1)[(!is.na(x)) & x>0] -> z
```

creates an object *z* and places in it the values of the vector *x*+1 for which the corresponding value in *x* was both non-missing and positive.

2. **A vector of positive integral quantities.** In this case the values in the index vector must lie in the set {1, 2, ..., length(*x*)}. The corresponding elements of the vector are selected and concatenated, *in that order*, in the result. The index vector can be of any length and the result is of the same length as the index vector. For example *x*[6] is the sixth component of *x* and

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

selects the first 10 elements of *x* (assuming length(*x*) is not less than 10). Also

```
> c("x","y")[rep(c(1,2,2,1), times=4)]
```

(an admittedly unlikely thing to do) produces a character vector of length 16 consisting of "x", "y", "y", "x" repeated four times.

3. **A vector of negative integral quantities.** Such an index vector specifies the values to be *excluded* rather than included. Thus

```
> y <- x[-(1:5)]
```

gives *y* all but the first five elements of *x*.

4. **A vector of character strings.** This possibility only applies where an object has a names attribute to identify its components. In this case a sub-vector of the names vector may be used in the same way as the positive integral labels in item 2 further above.

5. > fruit <- c(5, 10, 1, 20)

6. > names(fruit) <- c("orange", "banana", "apple", "peach")

```
> lunch <- fruit[c("apple","orange")]
```

The advantage is that alphanumeric *names* are often easier to remember than *numeric indices*. This option is particularly useful in connection with data frames, as we shall see later.

An indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed *only on those elements of the vector*. The expression must be of the form vector[index_vector] as having an arbitrary expression in place of the vector name does not make much sense here.

For example

```
> x[is.na(x)] <- 0
```

replaces any missing values in x by zeros and

```
> y[y < 0] <- -y[y < 0]
```

has the same effect as

```
> y <- abs(y)
```

2.8 Other types of objects

Vectors are the most important type of object in R, but there are several others which we will meet more formally in later sections.

- *matrices* or more generally *arrays* are multi-dimensional generalizations of vectors. In fact, they *are* vectors that can be indexed by two or more indices and will be printed in special ways. See [Arrays and matrices](#).
- *factors* provide compact ways to handle categorical data. See [Ordered and unordered factors](#).
- *lists* are a general form of vector in which the various elements need not be of the same type, and are often themselves vectors or lists. Lists provide a convenient way to return the results of a statistical computation. See [Lists](#).
- *data frames* are matrix-like structures, in which the columns can be of different types. Think of data frames as ‘data matrices’ with one row per observational unit but with (possibly) both numerical and categorical variables. Many experiments are best described by data frames: the treatments are categorical but the response is numeric. See [Data frames](#).
- *functions* are themselves objects in R which can be stored in the project’s workspace. This provides a simple and convenient way to extend R. See [Writing your own functions](#).

Array vs Matrix in R Programming

The [data structure](#) is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in [R programming](#) are tools for holding multiple values. The two most important data structures in R are Arrays and Matrices.

Arrays in R

[Arrays](#) are data storage objects in R containing more than or equal to 1 dimension. Arrays can contain only a single data type. The `array()` function is an in-built function which takes input as a vector and arranges them according to `dim` argument. Array is an iterable object, where the array elements are indexed, accessed and modified individually. [Operations on array](#) can be performed with similar

structures and dimensions. Uni-dimensional arrays are called vectors in R. Two-dimensional arrays are called matrices.

Syntax:

`array(array1, dim = c (r, c, m), dimnames = list(c.names, r.names, m.names))`

Parameters:

array1: a vector of values

dim: contains the number of matrices, m of the specified number of rows and columns

dimnames: contain the names for the dimensions

Example:

- Python3

```
# R program to illustrate an array

# creating a vector
vector1 <- c("A", "B", "C")

# declaring a character array
uni_array <- array(vector1)

print("Uni-Dimensional Array")
print(uni_array)

# creating another vector
vector <- c(1:12)

# declaring 2 numeric multi-dimensional
# array with size 2x3
multi_array <- array(vector, dim = c(2, 3, 2))

print("Multi-Dimensional Array")
```

```
print(multi_array)
```

Output:

```
[1] "Uni-Dimensional Array"
```

```
[1] "A" "B" "C"
```

```
[1] "Multi-Dimensional Array"
```

```
, , 1
```

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

```
[1,] 1 3 5
```

```
[2,] 2 4 6
```

```
, , 2
```

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

```
[1,] 7 9 11
```

```
[2,] 8 10 12
```

Matrices in R

[Matrix](#) in R is a table-like structure consisting of elements arranged in a fixed number of rows and columns. All the elements belong to a single data type. R contains an in-built function **matrix()** to create a matrix. Elements of a matrix can be accessed by providing indexes of rows and columns. The arithmetic operation, addition, subtraction, and multiplication can be performed on matrices with the same dimensions. Matrices can be easily converted to data frames CSVs.

Syntax:

```
matrix(data, nrow, ncol, byrow)
```

Parameters:

data: contain a vector of similar data type elements.

nrow: number of rows.

ncol: number of columns.

byrow: By default matrices are in column-wise order. So this parameter decides how to arrange the matrix

Example:

- Python3

```
# R program to illustrate a matrix
```

```
A = matrix(
```

```
  # Taking sequence of elements
```

```
  c(1, 2, 3, 4, 5, 6, 7, 8, 9),
```

```
  # No of rows and columns
```

```
  nrow = 3, ncol = 3,
```

```
  # By default matrices are
```

```
  # in column-wise order
```

```
  # So this parameter decides
```

```
  # how to arrange the matrix
```

```
  byrow = TRUE
```

```
)
```

```
print(A)
```

Output:

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

[1,] 1 2 3

[2,] 4 5 6

[3,] 7 8 9

Arrays vs Matrices

Arrays	Matrices
Arrays can contain greater than or equal to 1 dimensions.	Matrices contains 2 dimensions in a table like structure.
Array is a homogeneous data structure.	Matrix is also a homogeneous data structure.
It is a singular vector arranged into the specified dimensions.	It comprises of multiple equal length vectors stacked together in a table.
array() function can be used to create matrix by specifying the third dimension to be 1.	matrix() function however can be used to create at most 2-dimensional array.
Arrays are superset of matrices.	Matrices are a subset, special case of array where dimensions is two.
Limited set of collection-based operations.	Wide range of collection operations possible.
Mostly, intended for storage of data.	Mostly, matrices are intended for data transformation.

R Factors

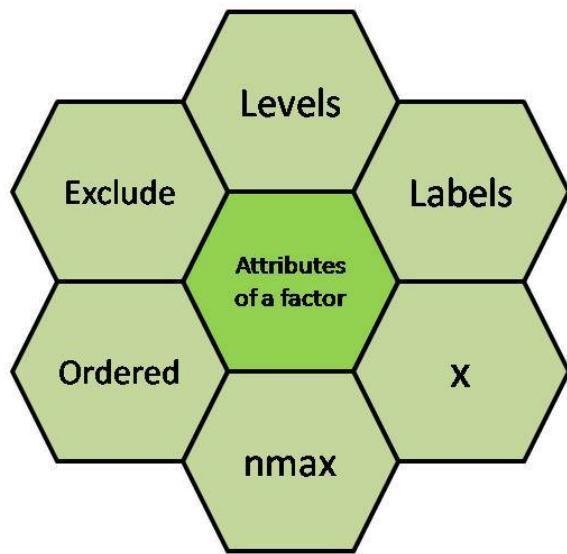
Factors in R Programming Language are data structures that are implemented to categorize the data or represent categorical data and store it on multiple levels.

They can be stored as integers with a corresponding label to every unique integer. The R factors may look similar to character [vectors](#), they are integers and care must be taken while using them as strings. The R factor accepts only a restricted number of distinct values. For example, a data field such as gender may contain values only from female, male, or transgender.

In the above example, all the possible cases are known beforehand and are predefined. These distinct values are known as levels. After a factor is created it only consists of levels that are by default sorted alphabetically.

Attributes of Factors in R Language

- **x:** It is the vector that needs to be converted into a factor.
- **Levels:** It is a set of distinct values which are given to the input vector x.
- **Labels:** It is a character vector corresponding to the number of labels.
- **Exclude:** This will mention all the values you want to exclude.
- **Ordered:** This logical attribute decides whether the levels are ordered.
- **nmax:** It will decide the upper limit for the maximum number of levels.



Creating a Factor in R Programming Language

The command used to create or modify a factor in R language is – **factor()** with a vector as input.

The two steps to creating an R factor :

- Creating a vector
- Converting the vector created into a factor using function **factor()**

Examples: Let us create a factor gender with levels female, male and transgender.

- R

```
# Creating a vector

x<-c("female", "male", "male", "female")

print(x)
```

```
# Converting the vector x into a factor
```

```
# named gender  
  
gender <- factor(x)  
  
print(gender)
```

Output

```
[1] "female" "male" "male" "female"  
  
[1] female male male female  
  
Levels: female male  
  
Levels can also be predefined by the programmer.
```

- R

```
# Creating a factor with levels defined by programmer  
  
gender <- factor(c("female", "male", "male", "female"),  
  
levels = c("female", "transgender", "male"));  
  
gender
```

Output

```
[1] female male male female  
  
Levels: female transgender male
```

Further one can check the levels of a factor by using function **levels()**.

Checking for a Factor in R

The function **is.factor()** is used to check whether the variable is a factor and returns “TRUE” if it is a factor.

- R

```
gender <- factor(c("female", "male", "male", "female"));  
  
print(is.factor(gender))
```

Output

[1] TRUE

Function **class()** is also used to check whether the variable is a factor and if true returns “factor”.

- R

```
gender <- factor(c("female", "male", "male", "female"));  
class(gender)
```

Output

[1] "factor"

Accessing elements of a Factor in R

Like we access elements of a vector, the same way we access the elements of a factor. If gender is a factor then gender[i] would mean accessing an i^{th} element in the factor.

Example

- R

```
gender <- factor(c("female", "male", "male", "female"));  
gender[3]
```

Output

[1] male

Levels: female male

More than one element can be accessed at a time.

Example

- R

```
gender <- factor(c("female", "male", "male", "female"));  
gender[c(2, 4)]
```

Output

```
[1] male female
```

Levels: female male

Subtract one element at a time.

Example

- R

```
gender <- factor(c("female", "male", "male", "female" ));  
gender[-3]
```

Output

```
[1] female male female
```

Levels: female male

- First, we create a factor vector **gender** with four elements: “female”, “male”, “male”, and “female”.
- Then, we use the square brackets **[-3]** to subset the vector and remove the third element, which is “male”.
- The output is the remaining elements of the **gender** vector, which are “female”, “male”, and “female”. The output also shows the levels of the factor, which are “female” and “male”.

Modification of a Factor in R

After a factor is formed, its components can be modified but the new values which need to be assigned must be at the predefined level.

Example

- R

```
gender <- factor(c("female", "male", "male", "female" ));  
gender[2]<-"female"  
gender
```

Output

```
[1] female female male  female
```

Levels: female male

For selecting all the elements of the factor gender except ith element, gender[-i] should be used. So if you want to modify a factor and add value out of predefined levels, then first modify levels.

Example

- R

```
gender <- factor(c("female", "male", "male", "female" ));
```

```
# add new level
```

```
levels(gender) <- c(levels(gender), "other")
```

```
gender[3] <- "other"
```

```
gender
```

Output

```
[1] female male  other female
```

Levels: female male other

Factors in Data Frame

The [Data frame](#) is similar to a 2D array with the columns containing all the values of one variable and the rows having one set of values from every column. There are four things to remember about data frames:

- column names are compulsory and cannot be empty.
- Unique names should be assigned to each row.
- The data frame's data can be only of three types- factor, numeric, and character type.
- The same number of data items must be present in each column.

In R language when we create a data frame, its column is categorical data, and hence a R factor is automatically created on it.

We can create a data frame and check if its column is a factor.

Example

- R

```
age <- c(40, 49, 48, 40, 67, 52, 53)
salary <- c(103200, 106200, 150200,
          10606, 10390, 14070, 10220)
gender <- c("male", "male", "transgender",
           "female", "male", "female", "transgender")
employee<- data.frame(age, salary, gender)
print(employee)
print(is.factor(employee$gender))
```

Output

```
age salary   gender
1 40 103200   male
2 49 106200   male
3 48 150200 transgender
4 40 10606   female
5 67 10390   male
6 52 14070   female
7 53 10220 transgender
[1] TRUE
```

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 an R list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0 like in 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

```
# R program to create a List

# The first attribute 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(empld, 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 an R 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 R list using the dollar command.

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 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 R list using indices. To access the top-level components of a R 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 R 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 R list can also be modified by accessing the components and replacing them with the ones which you want.

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 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  
[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 R 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)
```

```
# Concatenation of list using concatenation operator

empList = c(empName, empAge)

cat("After concatenation of the new list\n")

print(empList)
```

Output:

Before concatenation of the new list

```
$ID  
[1] 1 2 3 4
```

\$Names

```
[1] "Debi"  "Sandeep" "Subham" "Shiba"
```

\$`Total Staff`

```
[1] 4
```

After concatenation of the new list

```
[1] "Debi"  "Sandeep" "Subham" "Shiba"  "34"    "23"    "18"    "45"
```

Deleting components of a list

To delete components of a R 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 R 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 R 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)  
  
  
# 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
```

Class: list

After conversion to matrix:

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

Class: matrix

R – Data Frames

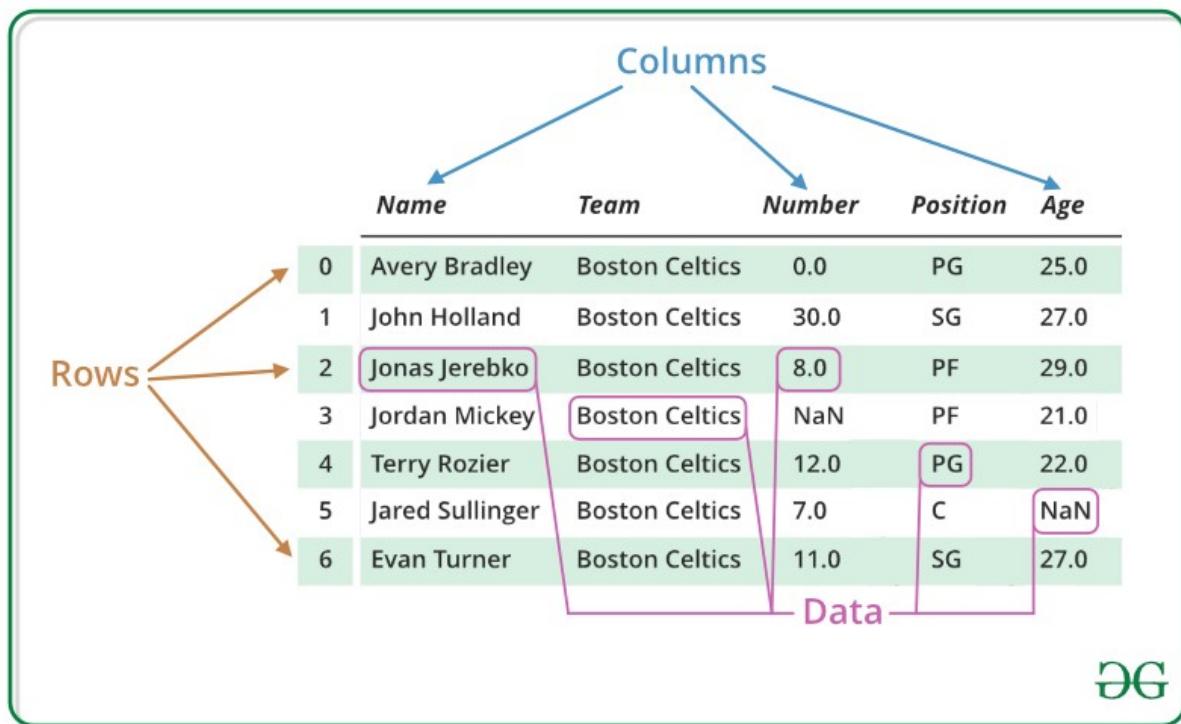
[R Programming Language](#) is an open-source programming language that is widely used as a statistical software and data analysis tool. **Data Frames in R Language** are generic data objects of R that are used to store tabular data.

Data frames can also be interpreted as matrices where each column of a [matrix](#) can be of different data types. R DataFrame is made up of three principal components, the data, rows, and columns.

R Data Frames Structure

As you can see in the image below, this is how a data frame is structured.

The data is presented in tabular form, which makes it easier to operate and understand.



R – Data Frames

Create Dataframe in R Programming Language

To create an R data frame use **data.frame()** function and then pass each of the vectors you have created as arguments to the function.

- R

```
# R program to create dataframe
```

```
# creating a data frame
```

```
friend.data <- data.frame(
```

```
  friend_id = c(1:5),
```

```
friend_name = c("Sachin", "Sourav",
               "Dravid", "Sehwag",
               "Dhoni"),
stringsAsFactors = FALSE
)
# print the data frame
print(friend.data)
```

Output:

```
friend_id friend_name
1         1   Sachin
2         2   Sourav
3         3   Dravid
4         4   Sehwag
5         5   Dhoni
```

Get the Structure of the R Data Frame

One can get the structure of the R data frame using [str\(\)](#) function in R.

It can display even the internal structure of large lists which are nested. It provides one-liner output for the basic R objects letting the user know about the object and its constituents.

- R

```
# R program to get the
# structure of the data frame

# creating a data frame
friend.data <- data.frame(
  friend_id = c(1:5),
  friend_name = c("Sachin", "Sourav",
```

```
"Dravid", "Sehwag",
"Dhoni"),
stringsAsFactors = FALSE
)

# using str()

print(str(friend.data))
```

Output:

```
'data.frame': 5 obs. of 2 variables:
 $ friend_id : int 1 2 3 4 5
 $ friend_name: chr "Sachin" "Sourav" "Dravid" "Sehwag" ...
NULL
```

Summary of Data in the R data frame

In the R data frame, the statistical summary and nature of the data can be obtained by applying [summary\(\)](#) function.

It is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular methods which depend on the class of the first argument.

- R

```
# R program to get the

# summary of the data frame


# creating a data frame

friend.data <- data.frame(
  friend_id = c(1:5),
  friend_name = c("Sachin", "Sourav",
  "Dravid", "Sehwag",
```

```
"Dhoni"),
stringsAsFactors = FALSE
)
# using summary()
print(summary(friend.data))
```

Output:

```
friend_id friend_name
Min. :1 Length:5
1st Qu.:2 Class :character
Median :3 Mode :character
Mean :3
3rd Qu.:4
Max. :5
```

Extract Data from Data Frame in R

Extracting data from an R data frame means that to access its rows or columns. One can extract a specific column from an R data frame using its column name.

- R

```
# R program to extract
# data from the data frame

# creating a data frame
friend.data <- data.frame(
  friend_id = c(1:5),
  friend_name = c("Sachin", "Sourav",
                 "Dravid", "Sehwag",
                 "Dhoni"),
```

```
stringsAsFactors = FALSE  
)  
  
# Extracting friend_name column  
  
result <- data.frame(friend.data$friend_name)  
  
print(result)
```

Output:

```
friend.data.friend_name  
1      Sachin  
2      Sourav  
3      Dravid  
4      Sehwag  
5      Dhoni
```

Expand Data Frame in R Language

A data frame in R can be expanded by adding new columns and rows to the already existing R data frame.

- R

```
# R program to expand  
  
# the data frame  
  
# creating a data frame  
  
friend.data <- data.frame(  
  friend_id = c(1:5),  
  friend_name = c("Sachin", "Sourav",  
    "Dravid", "Sehwag",  
    "Dhoni"),
```

```

stringsAsFactors = FALSE
)

# Expanding data frame
friend.data$location <- c("Kolkata", "Delhi",
                           "Bangalore", "Hyderabad",
                           "Chennai")
resultant <- friend.data

# print the modified data frame
print(resultant)

```

Output:

	friend_id	friend_name	location
1	1	Sachin	Kolkata
2	2	Sourav	Delhi
3	3	Dravid	Bangalore
4	4	Sehwag	Hyderabad
5	5	Dhoni	Chennai

In R, one can perform various types of operations on a data frame like **accessing rows and columns**, **selecting the subset of the data frame**, **editing data frames**, **delete rows and columns in a data frame**, etc.

Please refer to [DataFrame Operations in R](#) to know about all types of operations that can be performed on a data frame.

Access Items in R Data Frame

We can select and access any element from data frame by using single \$,brackets [] or double brackets [[]] to access columns from a data frame.

- R

```
# creating a data frame
```

```
friend.data <- data.frame(  
  friend_id = c(1:5),  
  friend_name = c("Sachin", "Sourav",  
    "Dravid", "Sehwag",  
    "Dhoni"),  
  stringsAsFactors = FALSE  
)
```

```
# Access Items using []
```

```
friend.data[1]
```

```
# Access Items using [[]]
```

```
friend.data[['friend_name']]
```

```
# Access Items using $
```

```
friend.data$friend_id
```

Output:

```
friend_id  
1     1  
2     2  
3     3  
4     4  
5     5  
Access Items using [[]]  
[1] "Sachin" "Sourav" "Dravid" "Sehwag" "Dhoni"  
Access Items using $  
[1] 1 2 3 4 5
```

Amount of Rows and Columns

We can find out how many rows and columns present in our dataframe by using dim function.

- R

```
# creating a data frame  
  
friend.data <- data.frame(  
  
  friend_id = c(1:5),  
  
  friend_name = c("Sachin", "Sourav",  
  
    "Dravid", "Sehwag",  
  
    "Dhoni"),  
  
  stringsAsFactors = FALSE  
  
)  
  
  
# find out the number of rows and columns  
  
dim(friend.data)
```

Output:

```
[1] 5 2
```

Add Rows and Columns in R Data Frame

You can easily add rows and columns in a R DataFrame. Insertion helps in expanding the already existing DataFrame, without needing a new one.

Let's look at how to add rows and columns in a DataFrame ? with an example:

Add Rows in R Data Frame

To add rows in a Data Frame, you can use a built-in function **rbind()**.

Following example demonstrate the working of rbind() in R Data Frame.

- R

```
# Creating a dataframe representing products in a store
```

```

Products <- data.frame(
  Product_ID = c(101, 102, 103),
  Product_Name = c("T-Shirt", "Jeans", "Shoes"),
  Price = c(15.99, 29.99, 49.99),
  Stock = c(50, 30, 25)
)

# Print the existing dataframe
cat("Existing dataframe (Products):\n")
print(Products)

# Adding a new row for a new product
New_Product <- c(104, "Sunglasses", 39.99, 40)
Products <- rbind(Products, New_Product)

# Print the updated dataframe after adding the new product
cat("\nUpdated dataframe after adding a new product:\n")
print(Products)

```

Output:

Existing dataframe (Products):

	Product_ID	Product_Name	Price	Stock
1	101	T-Shirt	15.99	50
2	102	Jeans	29.99	30
3	103	Shoes	49.99	25

Updated dataframe after adding a new product:

	Product_ID	Product_Name	Price	Stock
1	101	T-Shirt	15.99	50
2	102	Jeans	29.99	30
3	103	Shoes	49.99	25
4	104	Sunglasses	39.99	40

Add Columns in R Data Frame

To add columns in a Data Frame, you can use a built-in function **cbind()**.

Following example demonstrate the working of cbind() in R Data Frame.

- R

```
# Existing dataframe representing products in a store
```

```
Products <- data.frame(
```

```
  Product_ID = c(101, 102, 103),
```

```
  Product_Name = c("T-Shirt", "Jeans", "Shoes"),
```

```
  Price = c(15.99, 29.99, 49.99),
```

```
  Stock = c(50, 30, 25)
```

```
)
```

```
# Print the existing dataframe
```

```
cat("Existing dataframe (Products):\n")
```

```
print(Products)
```

```
# Adding a new column for 'Discount' to the dataframe
```

```
Discount <- c(5, 10, 8) # New column values for discount
```

```
Products <- cbind(Products, Discount)
```

```
# Rename the added column
```

```

colnames(Products)[ncol(Products)] <- "Discount" # Renaming the last column

# Print the updated dataframe after adding the new column
cat("\nUpdated dataframe after adding a new column 'Discount':\n")
print(Products)

```

Output:

Existing dataframe (Products):

	Product_ID	Product_Name	Price	Stock
1	101	T-Shirt	15.99	50
2	102	Jeans	29.99	30
3	103	Shoes	49.99	25

Updated dataframe after adding a new column 'Discount':

	Product_ID	Product_Name	Price	Stock	Discount
1	101	T-Shirt	15.99	50	5
2	102	Jeans	29.99	30	10
3	103	Shoes	49.99	25	8

Remove Rows and Columns

A data frame in R removes columns and rows from the already existing R data frame.

Remove Row in R DataFrame

- R

```

library(dplyr)

# Create a data frame

data <- data.frame(
  friend_id = c(1, 2, 3, 4, 5),
  friend_name = c("Sachin", "Sourav", "Dravid", "Sehwag", "Dhoni"),

```

```
location = c("Kolkata", "Delhi", "Bangalore", "Hyderabad", "Chennai")  
)  
  
data  
  
# Remove a row with friend_id = 3  
data <- subset(data, friend_id != 3)  
  
data
```

Output:

```
friend_id friend_name location  
1      1   Sachin  Kolkata  
2      2   Sourav   Delhi  
3      3   Dravid Bangalore  
4      4   Sehwag Hyderabad  
5      5   Dhoni   Chennai
```

```
# Remove a row with friend_id = 3
```

```
friend_id friend_name location  
1      1   Sachin  Kolkata  
2      2   Sourav   Delhi  
4      4   Sehwag Hyderabad  
5      5   Dhoni   Chennai
```

In the above code, we first created a data frame called **data** with three columns: **friend_id**, **friend_name**, and **location**. To remove a row with **friend_id** equal to 3, we used the **subset()** function and specified the condition **friend_id != 3**. This removed the row with **friend_id** equal to 3.

Remove Column in R DataFrame

- R

```
library(dplyr)

# Create a data frame

data <- data.frame(
  friend_id = c(1, 2, 3, 4, 5),
  friend_name = c("Sachin", "Sourav", "Dravid", "Sehwag", "Dhoni"),
  location = c("Kolkata", "Delhi", "Bangalore", "Hyderabad", "Chennai")
)

data
```

```
# Remove the 'location' column

data <- select(data, -location)
```

```
data
```

Output:

```
friend_id friend_name location
1      1    Sachin   Kolkata
2      2    Sourav     Delhi
3      3    Dravid Bangalore
4      4    Sehwag Hyderabad
5      5    Dhoni    Chennai
>
```

```
Remove the 'location' column
```

```
friend_id friend_name
1      1    Sachin
2      2    Sourav
3      3    Dravid
4      4    Sehwag
5      5    Dhoni
```

To remove the **location** column, we used the **select()** function and specified **-location**. The – sign indicates that we want to remove the **location** column. The resulting data frame **data** will have only two columns: **friend_id** and **friend_name**.

Combining Data Frames in R

There are 2 way to combine data frames in R. You can either combine them vertically or horizontally.

Let's look at both cases with example:

Combine R Data Frame Vertically

If you want to combine 2 data frames vertically, you can use **rbind()** function. This function works for combination of two or more data frames.

- R

```
# Creating two sample dataframes

df1 <- data.frame(
  Name = c("Alice", "Bob"),
  Age = c(25, 30),
  Score = c(80, 75)
)

df2 <- data.frame(
  Name = c("Charlie", "David"),
  Age = c(28, 35),
  Score = c(90, 85)
)

# Print the existing dataframes
cat("Dataframe 1:\n")
```

```
print(df1)

cat("\nDataframe 2:\n")

print(df2)

# Combining the dataframes using rbind()
combined_df <- rbind(df1, df2)

# Print the combined dataframe
cat("\nCombined Dataframe:\n")

print(combined_df)
```

Output:

Dataframe 1:

	Name	Age	Score
1	Alice	25	80
2	Bob	30	75

Dataframe 2:

	Name	Age	Score
1	Charlie	28	90
2	David	35	85

Combined Dataframe:

	Name	Age	Score
1	Alice	25	80
2	Bob	30	75
3	Charlie	28	90
4	David	35	85

Combine R Data Frame Horizontally:

If you want to combine 2 data frames horizontally, you can use **cbind() function**. This function works for combination of two or more data frames.

- R

```
# Creating two sample dataframes

df1 <- data.frame(
  Name = c("Alice", "Bob"),
  Age = c(25, 30),
  Score = c(80, 75)
)

df2 <- data.frame(
  Height = c(160, 175),
  Weight = c(55, 70)
)

# Print the existing dataframes
cat("Dataframe 1:\n")
print(df1)

cat("\nDataframe 2:\n")
print(df2)

# Combining the dataframes using cbind()
combined_df <- cbind(df1, df2)
```

```
# Print the combined dataframe  
  
cat("\nCombined Dataframe:\n")  
  
print(combined_df)
```

Output:

Dataframe 1:

	Name	Age	Score
1	Alice	25	80
2	Bob	30	75

Dataframe 2:

	Height	Weight
1	160	55
2	175	70

Combined Dataframe:

	Name	Age	Score	Height	Weight
1	Alice	25	80	160	55
2	Bob	30	75	175	70

Unit 2

Functions in R Programming

Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In [R Programming Language](#) when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more functions in R.

Table of Content

- [Function in R Programming](#)
- [Types of Function in R Language](#)
- [Built-in Function in R Programming Language](#)
- [User-defined Functions in R Programming Language](#)
- [R Function Example](#)

Function in R Programming

Functions are created in R by using the command **function()**. The general structure of the function file is as follows:

```
f = function(arguments){  
    statements  
}
```

Here f = function name

Functions in R Programming

Note: In the above syntax f is the function name, this means that you are creating a function with name f which takes certain arguments and executes the following statements.

Types of Function in R Language

1. **Built-in Function:** Built-in functions in R are pre-defined functions that are available in [R programming languages](#) to perform common tasks or operations.
2. **User-defined Function:** R language allow us to write our own function.

Built-in Function in R Programming Language

Here we will use built-in functions like sum(), max() and min().

- R

```
# Find sum of numbers 4 to 6.
```

```
print(sum(4:6))
```

```
# Find max of numbers 4 and 6.
```

```
print(max(4:6))
```

```
# Find min of numbers 4 and 6.
```

```
print(min(4:6))
```

Output

```
[1] 15
```

```
[1] 6
```

```
[1] 4
```

Other Built-in Functions in R

Functions	Syntax
Mathematical Functions	
a. abs()	calculates a number's absolute value.
b. sqrt()	calculates a number's square root.

Functions	Syntax
c. round()	rounds a number to the nearest integer.
d. exp()	calculates a number's exponential value
e. log()	which calculates a number's natural logarithm.
f. cos() , sin() , and tan()	calculates a number's cosine, sine, and tang.
Statistical Functions	
a. mean()	A vector's arithmetic mean is determined by the mean() function.
b. median()	A vector's median value is determined by the median() function.
c. cor()	calculates the correlation between two vectors.
d. var()	calculates the variance of a vector and calculates the standard deviation of a vector.
Data Manipulation Functions	
a. unique()	returns the unique values in a vector.
b. subset()	subsets a data frame based on conditions.

Functions	Syntax
c. aggregate()	groups data according to a grouping variable.
d. order()	uses ascending or descending order to sort a vector.
File Input/Output Functions	
a. read.csv()	reads information from a CSV file.
b. Write.csv()	publishes information to write a CSV file.
c. Read.table()	reads information from a tabular.
d. Write.table()	creates a tabular file with data.

User-defined Functions in R Programming Language

R provides built-in functions like **print()**, **cat()**, etc. but we can also create our own functions. These functions are called user-defined functions.

Example

- R

```
# A simple R function to check
# whether x is even or odd

evenOdd = function(x){
```

```
if(x %% 2 == 0)
  return("even")
else
  return("odd")
}

print(evenOdd(4))
print(evenOdd(3))
```

Output

```
[1] "even"
```

```
[1] "odd"
```

R Function Example

Single Input Single Output

Now create a function in R that will take a single input and gives us a single output.

Following is an example to create a function that calculates the area of a circle which takes in the arguments the radius. So, to create a function, name the function as “areaOfCircle” and the arguments that are needed to be passed are the “radius” of the circle.

- R

```
# A simple R function to calculate
# area of a circle

areaOfCircle = function(radius){
  area = pi*radius^2
  return(area)
```

```
}
```

```
print(areaOfCircle(2))
```

Output

12.56637

Multiple Input Multiple Output

Now create a function in R Language that will take multiple inputs and gives us multiple outputs using a list.

The functions in R Language take multiple input objects but returned only one object as output, this is, however, not a limitation because you can create lists of all the outputs which you want to create and once the list is created you can access them into the elements of the list and get the answers which you want.

Let us consider this example to create a function “Rectangle” which takes “length” and “width” of the rectangle and returns area and perimeter of that rectangle. Since R Language can return only one object. Hence, create one object which is a list that contains “area” and “perimeter” and return the list.

- R

```
# A simple R function to calculate  
  
# area and perimeter of a rectangle  
  
Rectangle = function(length, width){  
  
  area = length * width  
  
  perimeter = 2 * (length + width)  
  
  # create an object called result which is  
  
  # a list of area and perimeter  
  
  result = list("Area" = area, "Perimeter" = perimeter)
```

```
    return(result)
}

resultList = Rectangle(2, 3)
print(resultList["Area"])

print(resultList["Perimeter"])
```

Output

\$Area

[1] 6

\$Perimeter

[1] 10

Inline Functions in R Programming Language

Sometimes creating an R script file, loading it, executing it is a lot of work when you want to just create a very small function. So, what we can do in this kind of situation is an inline function.

To create an inline function you have to use the function command with the argument x and then the expression of the function.

Example

- R

```
# A simple R program to
# demonstrate the inline function

f = function(x) x^2*4+x/3

print(f(4))
```

```
print(f(-2))
```

```
print(0)
```

Output

65.33333

15.33333

0

Passing Arguments to Functions in R Programming Language

There are several ways you can pass the arguments to the function:

- **Case 1:** Generally in R, the arguments are passed to the function in the same order as in the function definition.
- **Case 2:** If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
- **Case 3:** If the arguments are not passed the default values are used to execute the function.

Now, let us see the examples for each of these cases in the following R code:

- R

```
# A simple R program to demonstrate
```

```
# passing arguments to a function
```

```
Rectangle = function(length=5, width=4){
```

```
    area = length * width
```

```
    return(area)
```

```
}
```

```
# Case 1:
```

```
print(Rectangle(2, 3))

# Case 2:

print(Rectangle(width = 8, length = 4))

# Case 3:

print(Rectangle())
```

Output

6

32

20

Lazy Evaluations of Functions in R Programming Language

In R the functions are executed in a lazy fashion. When we say lazy what it means is if some arguments are missing the function is still executed as long as the execution does not involve those arguments.

Example

In the function “Cylinder” given below. There are defined three-argument “diameter”, “length” and “radius” in the function and the volume calculation does not involve this argument “radius” in this calculation. Now, when you pass this argument “diameter” and “length” even though you are not passing this “radius” the function will still execute because this radius is not used in the calculations inside the function.

Let's illustrate this in an R code given below:

- R

```
# A simple R program to demonstrate

# Lazy evaluations of functions

Cylinder = function(diameter, length, radius ){
```

```
volume = pi*diameter^2*length/4  
return(volume)  
}
```

```
# This'll execute because this  
# radius is not used in the  
# calculations inside the function.  
print(Cylinder(5, 10))
```

Output

196.3495

If you do not pass the argument and then use it in the definition of the function it will throw an error that this “radius” is not passed and it is being used in the function definition.

Example

- R

```
# A simple R program to demonstrate  
# Lazy evaluations of functions
```

```
Cylinder = function(diameter, length, radius ){  
  volume = pi*diameter^2*length/4  
  print(radius)  
  return(volume)  
}
```

```
# This'll throw an error
```

```
print(Cylinder(5, 10))
```

Output

Error in print(radius) : argument "radius" is missing, with no default

Function Arguments in R Programming

Arguments are the parameters provided to a function to perform operations in a programming language. In [R programming](#), we can use as many arguments as we want and are separated by a comma. There is no limit on the number of arguments in a function in R. In this article, we'll discuss different ways of adding arguments in a function in R programming.

Adding Arguments in R

We can pass an argument to a function while calling the function by simply giving the value as an argument inside the parenthesis. Below is an implementation of a function with a single argument.

Syntax:

```
function_name <- function(arg1, arg2, ... )  
{  
  code  
}
```

Example:1

- R

```
calculate_square <- function(x) {  
  
  result <- x^2  
  
  return(result)  
  
}
```

```
value1 <- 5  
  
square1 <- calculate_square(value1)  
  
print(square1)
```

```
value2 <- -2.5  
square2 <- calculate_square(value2)  
print(square2)
```

Output:

[1] 25

[1] 6.25

Example:2

- R

```
# Function definition  
  
# To check n is divisible by 5 or not  
  
divisibleBy5 <- function(n){  
  
  if(n %% 5 == 0)  
  {  
    return("number is divisible by 5")  
  }  
  
  else  
  {  
    return("number is not divisible by 5")  
  }  
}
```

```
# Function call  
  
divisibleBy5(100)  
  
divisibleBy5(4)  
  
divisibleBy5(20.0)
```

Output:

```
[1] "number is divisible by 5"  
  
[1] "number is not divisible by 5"  
  
[1] "number is divisible by 5"
```

Adding Multiple Arguments in R

A function in R programming can have multiple arguments too. Below is an implementation of a function with multiple arguments.

Example:

- R

```
# Function definition  
  
# To check a is divisible by b or not  
  
divisible <- function(a, b){  
  
  if(a %% b == 0)  
  
  {  
  
    return(paste(a, "is divisible by", b))  
  
  }  
  
  else  
  
  {  
  
    return(paste(a, "is not divisible by", b))  
  
  }  
  
}
```

```
}
```

```
# Function call
```

```
divisible(7, 3)
```

```
divisible(36, 6)
```

```
divisible(9, 2)
```

Output:

```
[1] "7 is not divisible by 3"
```

```
[1] "36 is divisible by 6"
```

```
[1] "9 is not divisible by 2"
```

Adding Default Value in R

The default value in a function is a value that is not required to specify each time the function is called. If the value is passed by the user, then the user-defined value is used by the function otherwise, the default value is used. Below is an implementation of a function with a default value.

Example:

- R

```
# Function definition to check
```

```
# a is divisible by b or not.
```

```
# If b is not provided in function call,
```

```
# Then divisibility of a is checked with 3 as default
```

```
divisible <- function(a, b = 3){
```

```
if(a %% b == 0)
```

```
{
```

```

    return(paste(a, "is divisible by", b))

}

else

{

  return(paste(a, "is not divisible by", b))

}

}

# Function call

divisible(10, 5)

divisible(12)

```

Output:

```
[1] "10 is divisible by 5"

[1] "12 is divisible by 3"
```

Dots Argument

Dots argument (...) is also known as ellipsis which allows the function to take an undefined number of arguments. It allows the function to take an arbitrary number of arguments. Below is an example of a function with an arbitrary number of arguments.

Example:

- R

```
# Function definition of dots operator

fun <- function(n, ...){

l <- list(n, ...)

paste(l, collapse = " ")
```

```
}
```



```
# Function call
```



```
fun(5, 1L, 6i, TRUE, "GeeksForGeeks", "Dots operator")
```

Output:

```
[1] "5 1 0+6i TRUE GeeksForGeeks Dots operator"
```

Function as Argument

In R programming, functions can be passed to another functions as arguments. Below is an implementation of function as an argument.

Example:

- R

```
# Function definition
```



```
# Function is passed as argument
```



```
fun <- function(x, fun2){
```



```
  return(fun2(x))
```



```
}
```



```
# sum is built-in function
```



```
fun(c(1:10), sum)
```



```
# mean is built-in function
```



```
fun(rnorm(50), mean)
```

Output:

```
[1] 55
```

```
[1] 0.2153183
```

Types of Functions in R Programming

A [function](#) is a set of statements orchestrated together to perform a specific operation. A function is an object so the interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions. The function in turn performs the task and returns control to the interpreter as well as any return values that may be stored in other objects.

How to Define a Function?

In [R programming](#), a function can be defined using the keyword `function`. The syntax to define a function in R is as follows:

Syntax:

```
function_name = function(arg_1, arg_2, ...)  
{  
  Function body  
}
```

The various components/parts of a function are:

- **Function name:** It is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments:** An argument is a placeholder. Whenever a function is invoked, a value if passed to the argument. They are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body:** It contains all the set of statements that defines what actually the function does.
- **Return Values:** It is the values that function returns after the successful execution of the tasks. In more general, it is the last expression in the function body to be evaluated.

Calling a Function

It is nothing but calling the original function with a valid number of arguments. A function can be called with an argument, without an argument and with a default value as well.

Example: Calling a function without an argument

- r

```
# create a function cube  
  
# without an argument  
  
cube <- function()  
  
{  
  
  for(i in 1:10)  
  
  {  
  
    print(i^3)  
  
  }  
  
}  
  
  
# calling function cube without an argument  
  
cube()
```

Output:

```
[1] 1  
  
[1] 8  
  
[1] 27  
  
[1] 64  
  
[1] 125
```

```
[1] 216
```

```
[1] 343
```

```
[1] 512
```

```
[1] 729
```

```
[1] 1000
```

Example: Calling a function with an argument.

- r

```
# create a function factorial
```

```
# with a numeric argument n
```

```
factorial <- function(n)
```

```
{
```

```
  if(n==0)
```

```
{
```

```
  return(1)
```

```
}
```

```
else
```

```
{
```

```
  return(n * factorial(n - 2))
```

```
}
```

```
}
```

```
# calling function cube with an argument
```

```
factorial(7)
```

Output:

```
[1] 5040
```

Example: Calling a function with default argument.

- r

```
# create a function def_arg

# without an argument

def_arg <- function(a = 23, b = 35)

{

  output <- (a + b) * a + (a - b) * b

  print(output)

}

# calling function def_arg without an argument

def_arg()

# call the function with giving new values of the argument.

def_arg(16, 22)
```

Output:

[1] 914

[1] 476

Types of Function

There are mainly three types of function in R programming:

1. Primitive Functions
2. Infix Functions
3. Replacement Functions

Primitive Functions

Generally, a function comprises of three parts:

- The **formals()**, the list of arguments that control how you call the function.
- The **body()**, the code inside the function.
- The **environment()**, the data structure that determines how the function finds the values associated with the names.

The formals and body are defined explicitly whenever one creates a function, but the environment is specified implicitly, based on where you define the function. But there is an exception to the rule that a function has three components, some functions call C code directly. These functions are known as primitive functions. Primitive functions exist primarily in C, not R, so their **formals()**, **body()**, and **environment()** are NULL. These functions are only found in the base package. Primitive functions are harder to write but are highly efficient. They are of two types, either type builtin or type special.

- r

```
typeof(sum)
```

```
typeof('[')
```

```
[1] "builtin" #> typeof(sum)
```

```
[1] "character" #> typeof([''])
```

Example: To print the names of available primitive functions in your R console run the following code.

- r

```
names(methods:::BasicFunList)
```

Output:

```
[1] "$"          "$<-"        "["           "[<-"        "[["          "[[="          "cosh"  
"cummax"      "dimnames<-"  
  
[22] "as.raw"     "log2"        "tan"         "dim"         "as.logical"   "^"  
"is.finite"
```

```
[29] "sinh"      "log10"      "as.numeric"   "dim<-"       "is.array"     "tanpi"
"gamma"

[36] "atan"      "as.integer"  "Arg"         "signif"      "cumprod"     "cos"
"length"

[43] "!="        "digamma"    "exp"         "floor"       "acos"        "seq.int"
"abs"

[50] "length<-"  "sqrt"       "!"          "acosh"       "is.nan"      "Re"
"tanh"

[57] "names"     "cospi"      "&"          "anyNA"       "trunc"       "cummin"
"levels<"

[64] "*"         "Mod"        "|"          "names<-"    "+"          "log"
"Igamma"

[71] "as.complex" "asinh"      "-"          "sin"         "/"          "as.environment"
"<="

[78] "as.double"  "is.infinite" "is.numeric"   "rep"         "round"      "sinpi"
"dimnames"

[85] "asin"       "as.character" "%/%"       "is.na"       ""           "Im"

[92] "%%"        "trigamma"   "=="         "cumsum"     "atanh"      "sign"
"ceiling"

[99] "Conj"       "as.call"    "log1p"       "expm1"      "("          ":"          "="

[106] "@"         "{"          "~"          "&&"        ".C"         "baseenv"
"quote"

[113] "<-"        "is.name"    "if"          "||"         "attr<-"     "untraceMem"
".cache_class"

[120] "substitute" "interactive" "is.call"     "switch"     "function"   "is.single"
"is.null"

[127] "is.language" "is.pairlist" ".External.graphics" "globalenv"   "class<="
".Primitive"   "is.logical"

[134] "enc2utf8"   "UseMethod"  ".subset"     "proc.time"  "enc2native"
"repeat"        "<<-"
```

```

[141] "@<-"      "missing"     "nargs"       "isS4"        ".isMethodsDispatchOn"
"forceAndCall"   ".primTrace"

[148] "storage.mode<-"    ".Call"      "unclass"     "gc.time"     ".subset2"
"environment<-"  "emptyenv"

[155] "seq_len"     ".External2"  "is.symbol"   "class"       "on.exit"     "is.raw"
"for"

[162] "is.complex"   "list"       "invisible"  "is.character" "oldClass<="
"is.environment" "attributes"

[169] "break"        "return"     "attr"        "tracemem"   "next"       ".Call.graphics"
"standardGeneric"

[176] "is.atomic"     "retracemem" "expression"  "is.expression" "call"
"is.object"       "pos.to.env"

[183] "attributes<-"  ".primUntrace" "...length"   ".External"   "oldClass"
".Internal"       ".Fortran"

[190] "browser"      "is.double"   ".class2"    "while"      "nzchar"     "is.list"
"lazyLoadDBfetch"

[197] "...elt"       "is.integer" "is.function" "is.recursive" "seq_along"  "unlist"
"as.vector"

[204] "lengths"

```

Infix Functions

Infix functions are those functions in which the function name comes in between its arguments, and hence have two arguments. R comes with a number of built-in infix operators such as `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, and `<<-`. One can create his own infix functions that start and end with `%`. The name of an infix function is more flexible as it can contain any sequence of characters except `%`. There are some predefined infix operators in R programming.

Operators	Description
<code>%%</code>	Remainder operator

Operators	Description
%/%	Integer Division
%*%	Matrix multiplication
%o%	Outer Product
%x%	Kronecker product
%in%	Matching Operator

Example: Create a two argument function that gives greater of two numbers and bind it to a name that starts and ends with %.

- r

```
# R program to illustrate

# Infix function

'%Greater%' <- function(a, b)
{
  if(a > b) print(a)
  else if(b > a) print(b)
  else print("equal")
}
```

```
5 %Greater% 7  
2300 %Greater% 67
```

Output:

```
[1] 7  
[1] 2300
```

Replacement Functions

Replacement functions modify their arguments in place(modifying an R object usually creates a copy). The name of replacement functions are always succeeded by <->. They must have arguments named x and value, and return the modified object. In case of a replacement, a function needs additional arguments, the additional arguments should be placed between x and value, and must be called with additional arguments on the left. The name of the function has to be quoted as it is a syntactically valid but non-standard name and the parser would interpret <-> as the operator not as part of the function name if it weren't quoted.

Syntax:

```
"function_name<-" <- function(x, additional arguments, value)  
{  
  function body  
}
```

Example:

- r

```
# R program to illustrate  
  
# Replacement function  
  
"replace<-" <- function(x, value)
```

```
{  
  x[1] = value  
  x  
}  
  
x = rep.int(5, 7)  
replace(x) = 0L  
print(x)
```

Output:

```
[1] 0 5 5 5 5 5 5
```

Recursive Functions in R Programming

Recursion, in the simplest terms, is a type of [looping](#) technique. It exploits the basic working of functions in [R](#).

Recursive Function in R:

Recursion is when the function calls itself. This forms a loop, where every time the function is called, it calls itself again and again and this technique is known as recursion. Since the loops increase the memory we use the recursion. The recursive function uses the concept of recursion to perform iterative tasks they call themselves, again and again, which acts as a loop. These kinds of functions need a stopping condition so that they can stop looping continuously. Recursive functions call themselves. They break down the problem into smaller components. The function() calls itself within the original function() on each of the smaller components. After this, the results will be put together to solve the original problem.

Example: Factorial using Recursion in R

- R

```
rec_fac <- function(x){  
  if(x==0 || x==1)  
  {
```

```

        return(1)

    }

else

{

    return(x*rec_fac(x-1))

}

}

rec_fac(5)

```

Output:

[1] 120

Here, `rec_fac(5)` calls `rec_fac(4)`, which then calls `rec_fac(3)`, and so on until the input argument `x`, has reached 1. The function returns 1 and is destroyed. The return value is multiplied by the argument value and returned. This process continues until the first function call returns its output, giving us the final result.

0 seconds of 10 seconds Volume 0%

This ad will end in 6

Example: Sum of Series Using Recursion

Recursion in R is most useful for finding the sum of self-repeating series. In this example, we will find the sum of squares of a given series of numbers. $\text{Sum} = 1^2 + 2^2 + \dots + N^2$

Example:

- R

```

sum_series <- function(vec){

    if(length(vec)<=1)

```

```
{  
  return(vec^2)  
}  
  
else  
{  
  return(vec[1]^2+sum_series(vec[-1]))  
}  
}  
  
series <- c(1:10)  
sum_series(series)
```

Output:

[1] 385

• R

```
sum_n <- function(n) {  
  if (n == 1) {  
    return(1)  
  } else {  
    return(n + sum_n(n-1))  
  }  
}
```

```
# Test the sum_n function  
sum_n(5)
```

Output:

```
[1] 15
```

In this example, the sum_n function recursively increases n until it reaches 1, which is the base case of the recursion, by adding the current value of n to the sum of the first n-1 values.

- R

```
exp_n <- function(base, n) {  
  if (n == 0) {  
    return(1)  
  } else {  
    return(base * exp_n(base, n-1))  
  }  
}  
  
# Test the exp_n function  
exp_n(4, 5)
```

Output:

```
[1] 1024
```

In this example, the base case of the recursion is represented by the exp_n function, which recursively multiplies the base by itself n times until n equals 0.

Key Features of R Recursion

- The use of recursion, often, makes the code shorter and it also looks clean.
- It is a simple solution for a few cases.
- It expresses in a function that calls itself.

Applications of Recursion in R

- Recursive functions are used in many efficient programming techniques like dynamic programming language(DSL) or divide-and-conquer algorithms.

- In dynamic programming, for both top-down as well as bottom-up approaches, recursion is vital for performance.
- In divide-and-conquer algorithms, we divide a problem into smaller sub-problems that are easier to solve. The output is then built back up to the top. Recursion has a similar process, which is why it is used to implement such algorithms.
- In its essence, recursion is the process of breaking down a problem into many smaller problems, these smaller problems are further broken down until the problem left is trivial. The solution is then built back up piece by piece.

Types of Recursion in R

1. **Direct Recursion:** The recursion that is direct involves a function calling itself directly. This kind of recursion is the easiest to understand.
2. **Indirect Recursion:** An indirect recursion is a series of function calls in which one function calls another, which in turn calls the original function.
3. **Mutual Recursion:** Multiple functions that call each other repeatedly make up mutual recursion. To complete a task, each function depends on the others.
4. **Nested Recursion:** Nested recursion happens when one recursive function calls another recursively while passing the output of the first call as an argument. The arguments of one recursion are nested inside of this one.
5. **Structural Recursion:** Recursion that is based on the structure of the data is known as structural recursion. It entails segmenting a complicated data structure into smaller pieces and processing each piece separately.

Conversion Functions in R Programming

Sometimes to analyze data using [R](#), we need to convert data into another data type. As we know R has the following data types Numeric, Integer, Logical, Character, etc. similarly R has various conversion functions that are used to convert the data type.

In R, Conversion Function are of two types:

- **Conversion Functions for Data Types**

- **Conversion Functions for Data Structures**

Conversion Functions For Data Types

There are various conversion functions available for Data Types. These are:

- **as.numeric()**

Decimal value known numeric values in R. It is the default data type for real numbers in R. In R **as.numeric()** converts any values into numeric values.

Syntax:

```
// Conversion into numeric data type  
as.numeric(x)
```

Example:

```
# A simple R program to convert  
  
# character data type into numeric data type  
x<-c('1', '2', '3')  
  
  
# Print x  
print(x)  
  
  
# Print the type of x  
print(typeof(x))  
  
  
# Conversion into numeric data type  
y<-as.numeric(x)  
  
  
# print the type of y  
print(typeof(y))
```

Output:

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

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

```
[1] "double"
```

- **as.integer()**

In R, Integer data type is a collection of all integers. In order to create an integer variable in R and convert any data type in to Integer we use **as.integer()** function.

Syntax:

- // Conversion of any data type into Integer data type
- as.integer(x)

Example:

```
# A simple R program to convert  
  
# numeric data type into integer data type  
  
x<-c(1.3, 5.6, 55.6)  
  
  
# Print x  
  
print(x)  
  
  
# Print type of x  
  
print(typeof(x))  
  
  
# Conversion into integer data type  
  
y<-as.integer(x)  
  
  
# Print y
```

```
print(y)

# Print type of y
print(typeof(y))
```

Output:

```
[1] 1.3 5.6 55.6
[1] "double"
[1] 1 5 55
[1] "integer"
```

- **as.character()**

In R, character data is used to store character value and string. To create an character variable in R, we invoke the **as.character()** function and also if we want to convert any data type in to character we use **as.character()** function.

Syntax:

- // Conversion of any data type into character data type
- as.character(x)

Example:

```
x<-c(1.3, 5.6, 55.6)

# Print x
print(x)

# Print type of x
print(typeof(x))
```

```
# Conversion into character data type  
y<-as.character(x)
```

```
# Print y  
print(y)
```

```
# Print type of y  
print(typeof(y))
```

Output:

```
[1] 1.3 5.6 55.6  
[1] "double"  
[1] "1.3" "5.6" "55.6"  
[1] "character"
```

- **as.logical()**

Logical value is created to compare variables which return either true or false. To compare variables and to convert any value into true or false, R uses **as.logical()** function.

Syntax:

- // Conversion of any data type into logical data type
- as.logical(x)

Example:

```
x = 3  
y = 8
```

```
# Conversion into logical value
```

```
result<-as.logical(x>y)
```

```
# Print result
```

```
print(result)
```

Output:

```
[1] FALSE
```

- **as.date()**

In R as.date() function is used to convert string into date format.

Syntax:

```
// Print string into date format
```

```
as.date(variable, "%m/%d/%y")
```

Example:

```
dates <- c("02/27/92", "02/27/92",
         "01/14/92", "02/28/92",
         "02/01/92")
```

```
# Conversion into date format
```

```
result<-as.Date(dates, "%m/%d/%y")
```

```
# Print result
```

```
print(result)
```

Output:

```
[1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

Conversion Functions For Data Structure

There are various conversion functions available for Data Structure. These are:

- **as.data.frame()**

Data Frame is used to store data tables. Which is list of vectors of equal length. In R, sometimes to analyse data we need to convert list of vector into data.frame. So for this R uses **as.data.frame()** function to convert list of vector into data frame.

Syntax:

- // Conversion of any data structure into data frame
- as.data.frame(x)

Example:

```
x<- list( c('a', 'b', 'c'),
           c('e', 'f', 'g'), c('h', 'i', 'j'))

# Print x
print(x)

# Conversion in to data frame
y<-as.data.frame(x)

# Print y
print(y)
```

Output:

[[1]]

[1] "a" "b" "c"

[[2]]

[1] "e" "f" "g"

```
[[3]]
```

```
[1] "h" "i" "j"
```

```
c.a....b....c.. c..e....f....g.. c..h....i....j..
```

1	a	e	h
2	b	f	i
3	c	g	j

- **as.vector()**

R has a function **as.vector()** which is used to convert a distributed matrix into a non-distributed vector. Vector generates a vector of the given length and mode.

Syntax:

- // Conversion of any data structure into vector
- as.vector(x)

Example:

```
x<-c(a=1, b=2)

# Print x
print(x)

# Conversion into vector
y<-as.vector(x)

# Print y
print(y)
```

Output:

```
a b  
1 2  
[1] 1 2
```

- **as.matrix()**

In R, there is a function **as.matrix()** which is used to convert a data.table into a matrix, optionally using one of the columns in the data.table as the matrix row names.

Syntax:

- // Conversion into matrix
- as.matrix(x)

Example:

```
# Importing library  
  
library(data.table)  
  
x <- data.table(A = letters[1:5], X = 1:5, Y = 6:10)
```

```
# Print x
```

```
print(x)
```

```
# Conversion into matrix
```

```
z<-as.matrix(x)
```

```
# Print z
```

```
print(z)
```

Output:

```
A X Y
```

```
1: a 1 6
```

```
2: b 2 7
```

3: c 3 8

4: d 4 9

5: e 5 10

A X Y

[1,] "a" "1" "6"

[2,] "b" "2" "7"

[3,] "c" "3" "8"

[4,] "d" "4" "9"

[5,] "e" "5" "10"

R Operators

Operators are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numerical as input operands.

R Operators

R supports majorly four kinds of binary operators between a set of operands. In this article, we will see various types of **operators in R Programming language** and their usage.

Types of the operator in R language

- Arithmetic Operators

• Logical Operators

- Relational Operators
- Assignment Operators
- Miscellaneous Operator

Arithmetic Operators

Arithmetic operations in R simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors.

Addition operator (+)

The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:

- R

```
a <- c(1, 0.1)  
b <- c(2.33, 4)  
print(a+b)
```

Output : 3.33 4.10

Subtraction Operator (-)

The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:

- R

```
a <- 6  
b <- 8.4  
print(a-b)
```

Output : -2.4

Multiplication Operator (*)

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

- R

```
B= c(4,4)
```

```
C= c(5,5)  
print (B*C)
```

Output : 20 20

Division Operator (/)

The first operand is divided by the second operand with the use of the '/' operator.

- R

```
a <- 10  
b <- 5  
print (a/b)
```

Output : 2

Power Operator (^)

The first operand is raised to the power of the second operand.

- R

```
a <- 4  
b <- 5  
print(a^b)
```

Output : 1024

Modulo Operator (%%)

The remainder of the first operand divided by the second operand is returned.

- R

```
list1<- c(2, 22)  
list2<-c(2,4)
```

```
print(list1 %% list2)
```

Output : 0 2

The following R code illustrates the usage of all Arithmetic R operators.

- R

```
# R program to illustrate  
  
# the use of Arithmetic operators  
  
vec1 <- c(0, 2)  
  
vec2 <- c(2, 3)  
  
  
# Performing operations on Operands  
  
cat ("Addition of vectors :", vec1 + vec2, "\n")  
  
cat ("Subtraction of vectors :", vec1 - vec2, "\n")  
  
cat ("Multiplication of vectors :", vec1 * vec2, "\n")  
  
cat ("Division of vectors :", vec1 / vec2, "\n")  
  
cat ("Modulo of vectors :", vec1 %% vec2, "\n")  
  
cat ("Power operator :", vec1 ^ vec2)
```

Output

```
Addition of vectors : 2 5  
  
Subtraction of vectors : -2 -1  
  
Multiplication of vectors : 0 6  
  
Division of vectors : 0 0.6666667  
  
Modulo of vectors : 0 2  
  
Power operator : 0 8
```

Logical Operators

Logical operations in R simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number.

Element-wise Logical AND operator (&)

Returns True if both the operands are True.

- R

```
list1 <- c(TRUE, 0.1)

list2 <- c(0,4+3i)

print(list1 & list2)
```

Output : FALSE TRUE

Any non zero integer value is considered as a TRUE value, be it complex or real number.

Element-wise Logical OR operator (|)

Returns True if either of the operands is True.

- R

```
list1 <- c(TRUE, 0.1)

list2 <- c(0,4+3i)

print(list1 | list2)
```

Output : TRUE TRUE

NOT operator (!)

A unary operator that negates the status of the elements of the operand.

- R

```
list1 <- c(0,FALSE)
```

```
print(!list1)
```

Output : TRUE TRUE

Logical AND operator (&&)

Returns True if both the first elements of the operands are True.

- R

```
list1 <- c(TRUE, 0.1)  
list2 <- c(0,4+3i)  
print(list1 && list2)
```

Output : FALSE

Compares just the first elements of both the lists.

Logical OR operator (||)

Returns True if either of the first elements of the operands is True.

- R

```
list1 <- c(TRUE, 0.1)  
list2 <- c(0,4+3i)  
print(list1 | list2)
```

Output : TRUE

The following R code illustrates the usage of all Logical Operators in R:

- R

```
# R program to illustrate  
# the use of Logical operators  
vec1 <- c(0,2)
```

```
vec2 <- c(TRUE,FALSE)

# Performing operations on Operands

cat ("Element wise AND :", vec1 & vec2, "\n")

cat ("Element wise OR :", vec1 | vec2, "\n")

cat ("Logical AND :", vec1 && vec2, "\n")

cat ("Logical OR :", vec1 || vec2, "\n")

cat ("Negation :", !vec1)
```

Output

Element wise AND : FALSE FALSE

Element wise OR : TRUE TRUE

Logical AND : FALSE

Logical OR : TRUE

Negation : TRUE FALSE

Relational Operators

The relational operators in R carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.

Less than (<)

Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1,"apple")

list2 <- c(0,0.1,"bat")
```

```
print(list1<list2)
```

Output : FALSE FALSE TRUE

Less than equal to (<=)

Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
```

```
list2 <- c(TRUE, 0.1, "bat")
```

```
# Convert lists to character strings
```

```
list1_char <- as.character(list1)
```

```
list2_char <- as.character(list2)
```

```
# Compare character strings
```

```
print(list1_char <= list2_char)
```

Output : TRUE TRUE TRUE

Greater than (>)

Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
```

```
list2 <- c(TRUE, 0.1, "bat")
```

```
print(list1_char > list2_char)
```

Output : FALSE FALSE FALSE

Greater than equal to (\geq)

Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(TRUE, 0.1, "bat")
print(list1_char >= list2_char)
```

Output : TRUE TRUE FALSE

Not equal to (\neq)

Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.

- R

```
list1 <- c(TRUE, 0.1,'apple')
list2 <- c(0.0.1,"bat")
print(list1!=list2)
```

Output : TRUE FALSE TRUE

The following R code illustrates the usage of all Relational Operators in R:

- R

```
# R program to illustrate
# the use of Relational operators
vec1 <- c(0, 2)
vec2 <- c(2, 3)
```

```
# Performing operations on Operands

cat ("Vector1 less than Vector2 :", vec1 < vec2, "\n")

cat ("Vector1 less than equal to Vector2 :", vec1 <= vec2, "\n")

cat ("Vector1 greater than Vector2 :", vec1 > vec2, "\n")

cat ("Vector1 greater than equal to Vector2 :", vec1 >= vec2, "\n")

cat ("Vector1 not equal to Vector2 :", vec1 != vec2, "\n")
```

Output

Vector1 less than Vector2 : TRUE TRUE
Vector1 less than equal to Vector2 : TRUE TRUE
Vector1 greater than Vector2 : FALSE FALSE
Vector1 greater than equal to Vector2 : FALSE FALSE
Vector1 not equal to Vector2 : TRUE TRUE

Assignment Operators

Assignment operators in R are used to assigning values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right

Left Assignment (<- or <-< or =)

Assigns a value to a vector.

- R

```
vec1 = c("ab", TRUE)

print (vec1)
```

Output : "ab" "TRUE"

Right Assignment (-> or ->>)

Assigns value to a vector.

- R

```
c("ab", TRUE) ->> vec1  
print (vec1)
```

Output : "ab" "TRUE"

The following R code illustrates the usage of all Relational Operators in R:

- R

```
# R program to illustrate  
  
# the use of Assignment operators  
  
vec1 <- c(2:5)  
  
c(2:5) ->> vec2  
  
vec3 <<- c(2:5)  
  
vec4 = c(2:5)  
  
c(2:5) -> vec5
```

```
# Performing operations on Operands  
  
cat ("vector 1 :", vec1, "\n")  
  
cat("vector 2 :", vec2, "\n")  
  
cat ("vector 3 :", vec3, "\n")  
  
cat("vector 4 :", vec4, "\n")  
  
cat("vector 5 :", vec5)
```

Output

vector 1 : 2 3 4 5

vector 2 : 2 3 4 5

vector 3 : 2 3 4 5

```
vector 4 : 2 3 4 5
```

```
vector 5 : 2 3 4 5
```

Miscellaneous Operators

These are the mixed operators in R that simulate the printing of sequences and assignment of vectors, either left or right-handed.

%in% Operator

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

- R

```
val <- 0.1  
  
list1 <- c(TRUE, 0.1, "apple")  
  
print (val %in% list1)
```

Output : TRUE

Checks for the value 0.1 in the specified list. It exists, therefore, prints TRUE.

%*% Operator

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of the first matrix must be equal to the number of rows of the second matrix. Multiplication of the matrix A with its transpose, B, produces a square matrix.

- R

```
mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)  
  
print (mat)  
  
print( t(mat))  
  
pro = mat %*% t(mat)
```

```
print(pro)
```

Input :

Output :[,1] [,2] [,3] #original matrix of order 2x3

```
[1,] 1 3 5
```

```
[2,] 2 4 6
```

,[1] [,2] #transposed matrix of order 3x2

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

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

```
[3,] 5 6
```

,[1] [,2] #product matrix of order 2x2

```
[1,] 35 44
```

```
[2,] 44 56
```

The following R code illustrates the usage of all Miscellaneous Operators in R:

- R

```
# R program to illustrate

# the use of Miscellaneous operators

mat <- matrix (1:4, nrow = 1, ncol = 4)

print("Matrix elements using : ")

print(mat)

product = mat %*% t(mat)

print("Product of matrices")

print(product,)
```

```
cat ("does 1 exist in prod matrix :", "1" %in% product)
```

Output

```
[1] "Matrix elements using : "
```

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

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

```
[1] "Product of matrices"
```

```
[,1]
```

```
[1,] 30
```

does 1 exist in prod matrix : FALSE

Control Statements in R Programming

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable. In this article, we'll discuss all the control statements with the examples.

In [R programming](#), there are 8 types of control statements as follows:

- [if condition](#)
- [if-else condition](#)
- [for loop](#)
- [nested loops](#)
- [while loop](#)
- [repeat and break statement](#)
- [return statement](#)
- [next statement](#)

if condition

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

Syntax:

```
if(expression){
```

```
    statements
```

```
    ....
```

```
    ....
```

```
}
```

Example:

```
x <- 100

if(x > 10){
  print(paste(x, "is greater than 10"))
}
```

Output:

```
[1] "100 is greater than 10"
```

if-else condition

It is similar to **if** condition but when the test expression in if condition fails, then statements in **else** condition are executed.

Syntax:

```
if(expression){
```

```
    statements
```

```
    ....
```

```
    ....
```

```
}
```

```
else{
```

```
    statements
```

```
    ....
```

```
    ....
```

```
}
```

Example:

```
x <- 5
```

```
# Check value is less than or greater than 10
```

```
if(x > 10){
```

```
    print(paste(x, "is greater than 10"))
```

```
}else{
```

```
    print(paste(x, "is less than 10"))
```

```
}
```

Output:

```
[1] "5 is less than 10"
```

for loop

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

Syntax:

```
for(value in vector){
```

```
    statements
```

```
    ....
```

```
    ....
```

```
}
```

Example:

```
x <- letters[4:10]
```

```
for(i in x){  
  print(i)  
}
```

Output:

```
[1] "d"  
[1] "e"  
[1] "f"  
[1] "g"  
[1] "h"  
[1] "i"  
[1] "j"
```

Nested loops

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

Example:

```
# Defining matrix  
  
m <- matrix(2:15, 2)  
  
  
for (r in seq(nrow(m))) {  
  for (c in seq(ncol(m))) {  
    print(m[r, c])
```

```
}
```

Output:

```
[1] 2
```

```
[1] 4
```

```
[1] 6
```

```
[1] 8
```

```
[1] 10
```

```
[1] 12
```

```
[1] 14
```

```
[1] 3
```

```
[1] 5
```

```
[1] 7
```

```
[1] 9
```

```
[1] 11
```

```
[1] 13
```

```
[1] 15
```

while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

Syntax:

```
while(expression){
```

```
    statement
```

```
    ....
```

....

}

Example:

```
x = 1

# Print 1 to 5

while(x <= 5){

  print(x)

  x = x + 1

}
```

Output:

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

repeat loop and break statement

repeat is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, **break** statement is used to exit from the loop. **break** statement can be used in any type of loop to exit from the loop.

Syntax:

```
repeat {
  statements
  ....
  ....
```

```
if(expression) {  
    break  
}  
}
```

Example:

```
x = 1  
  
# Print 1 to 5  
  
repeat{  
    print(x)  
    x = x + 1  
    if(x > 5){  
        break  
    }  
}
```

Output:

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

return statement

return statement is used to return the result of an executed function and returns control to the calling function.

Syntax:

```
return(expression)
```

Example:

```
# Checks value is either positive, negative or zero

func <- function(x){

  if(x > 0){

    return("Positive")

  }else if(x < 0){

    return("Negative")

  }else{

    return("Zero")

  }

}

func(1)

func(0)

func(-1)
```

Output:

```
[1] "Positive"
```

```
[1] "Zero"
```

```
[1] "Negative"
```

next statement

next statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

Example:

```
# Defining vector  
  
x <- 1:10  
  
  
# Print even numbers  
  
for(i in x){  
  
  if(i%%2 != 0){  
  
    next #Jumps to next loop  
  
  }  
  
  print(i)  
  
}
```

Output:

```
[1] 2  
  
[1] 4  
  
[1] 6  
  
[1] 8  
  
[1] 10
```

Decision Making in R Programming – if, if-else, if-else-if ladder, nested if-else, and switch

Decision making is about deciding the order of execution of statements based on certain conditions. In decision making programmer needs to provide some condition which is evaluated by the program, along with it there also provided some statements which are executed if the condition is true and optionally other statements if the condition is evaluated to be false.

The decision making statement in [R](#) are as followed:

- [if statement](#)
- [if-else statement](#)
- [if-else-if ladder](#)
- [nested if-else statement](#)

- [switch statement](#)

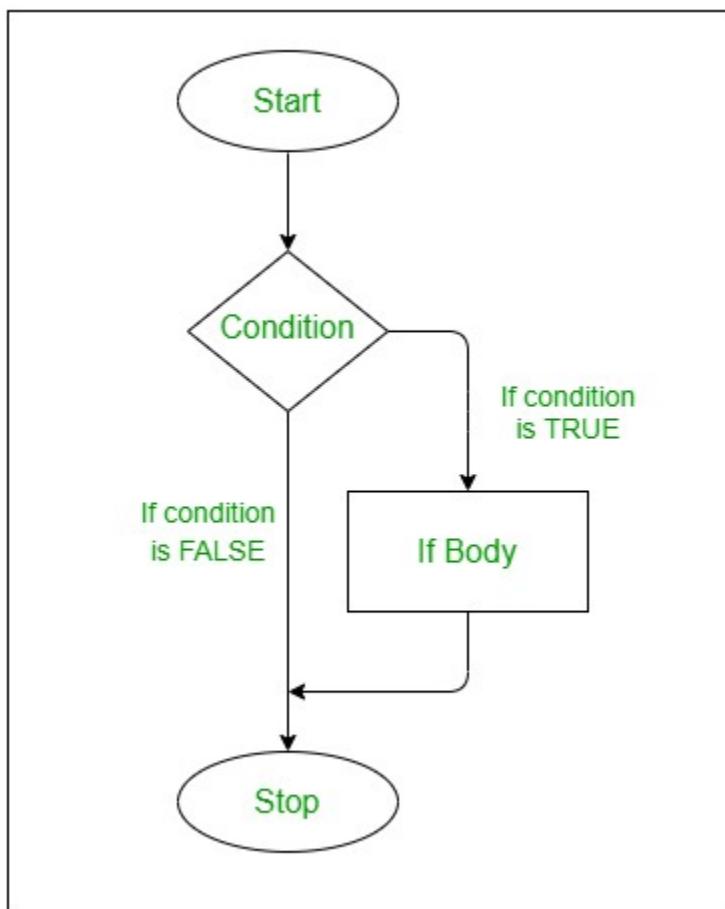
if statement

Keyword [if](#) tells compiler that this is a decision control instruction and the condition following the keyword if is always enclosed within a pair of parentheses. If the condition is TRUE the statement gets executed and if condition is FALSE then statement does not get executed.

Syntax:

```
if(condition is true){
    execute this statement
}
```

Flow Chart:



Example:

- r

```

# R program to illustrate

# if statement

a <- 76

b <- 67


# TRUE condition

if(a > b)

{

  c <- a - b

  print("condition a > b is TRUE")

  print(paste("Difference between a, b is : ", c))

}

# FALSE condition

if(a < b)

{

  c <- a - b

  print("condition a < b is TRUE")

  print(paste("Difference between a, b is : ", c))

}

```

Output:

```

[1] "condition a > b is TRUE"

[1] "Difference between a, b is : 9"

```

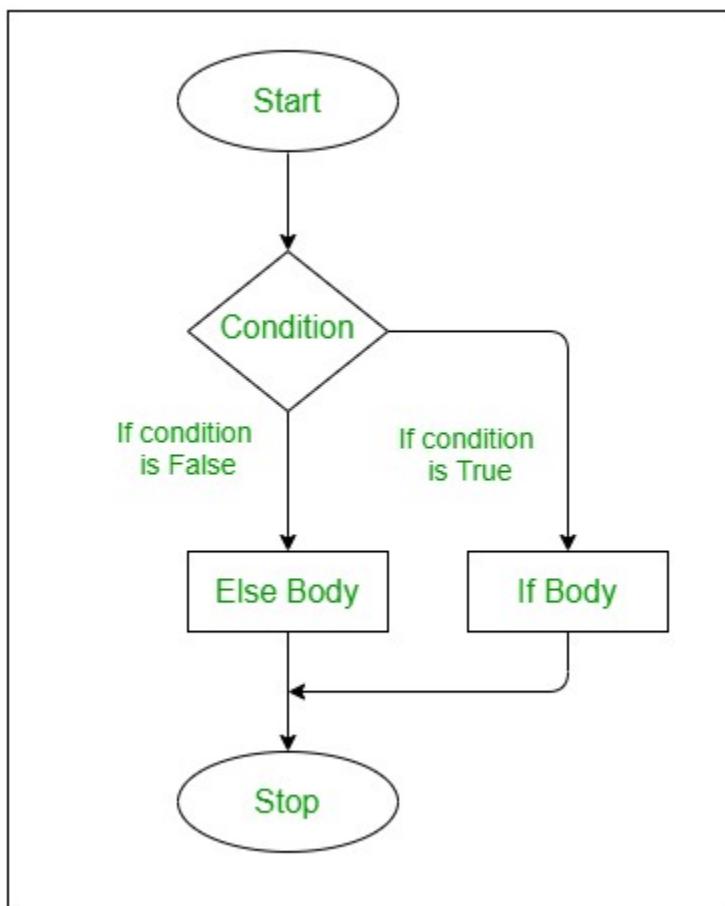
if-else statement

If-else, provides us with an optional else block which gets executed if the condition for if block is false. If the condition provided to if block is true then the statement within the if block gets executed, else the statement within the else block gets executed.

Syntax:

```
if(condition is true) {  
    execute this statement  
} else {  
    execute this statement  
}
```

Flow Chart:



Example :

- r

```

# R if-else statement Example

a <- 67

b <- 76

# This example will execute else block

if(a > b)

{

  c <- a - b

  print("condition a > b is TRUE")

  print(paste("Difference between a, b is : ", c))

} else

{

  c <- a - b

  print("condition a > b is FALSE")

  print(paste("Difference between a, b is : ", c))

}

```

Output:

```

[1] "condition a > b is FALSE"

[1] "Difference between a, b is : -9"

```

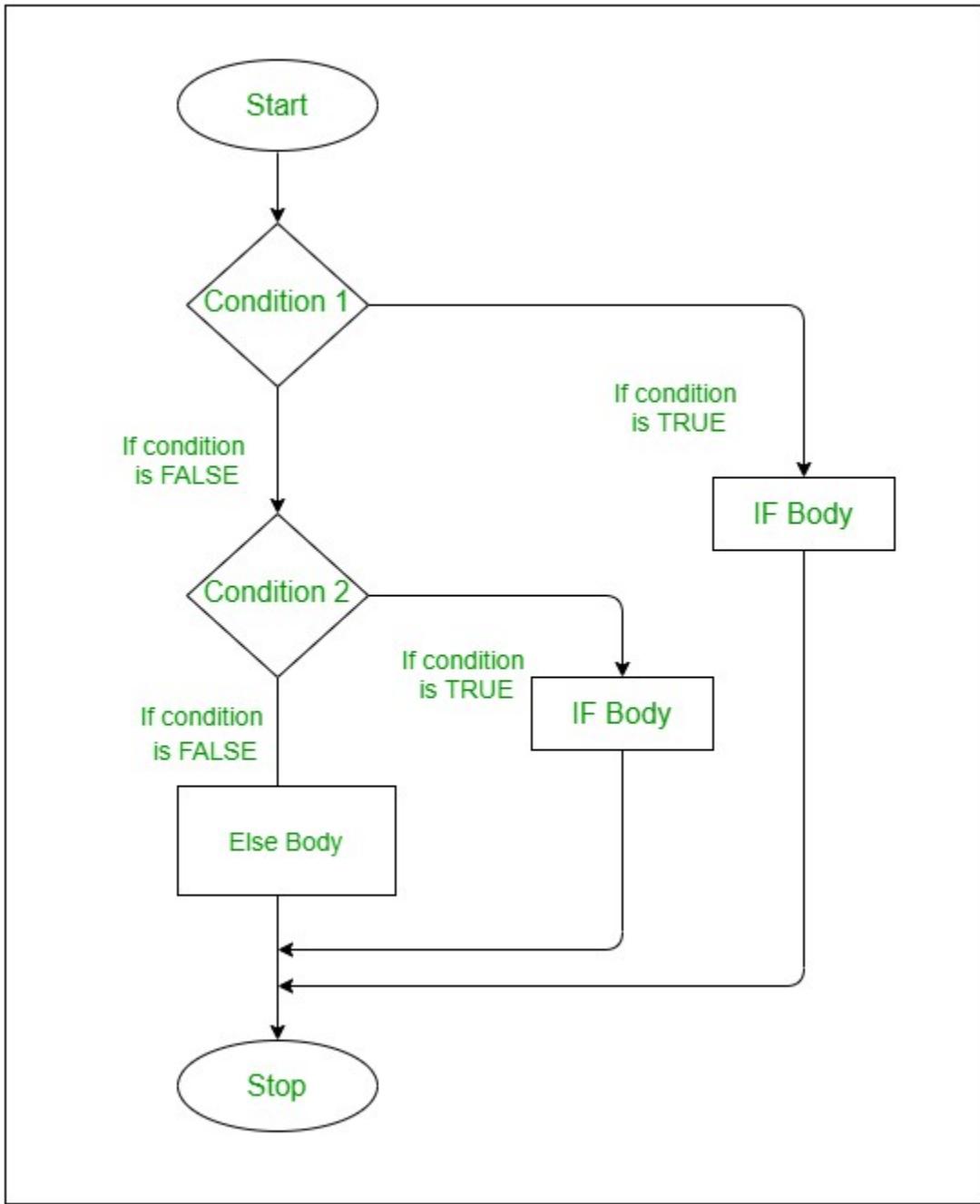
if-else-if ladder

It is similar to if-else statement, here the only difference is that an if statement is attached to else. If the condition provided to if block is true then the statement within the if block gets executed, else-if the another condition provided is checked and if true then the statement within the block gets executed.

Syntax:

```
if(condition 1 is true) {  
    execute this statement  
} else if(condition 2 is true) {  
    execute this statement  
} else {  
    execute this statement  
}
```

Flow Chart:



Example :

- r

```
# R if-else-if ladder Example
```

```
a <- 67
```

```

b <- 76

c <- 99

if(a > b && b > c)

{

  print("condition a > b > c is TRUE")

} else if(a < b && b > c)

{

  print("condition a < b > c is TRUE")

} else if(a < b && b < c)

{

  print("condition a < b < c is TRUE")

}

```

Output:

```
[1] "condition a < b < c is TRUE"
```

Nested if-else statement

When we have an if-else block as an statement within an if block or optionally within an else block, then it is called as nested if else statement. When an if condition is true then following child if condition is validated and if the condition is wrong else statement is executed, this happens within parent if condition. If parent if condition is false then else block is executed with also may contain child if else statement.

Syntax:

```

if(parent condition is true) {

  if( child condition 1 is true) {

    execute this statement
  }
}

```

```

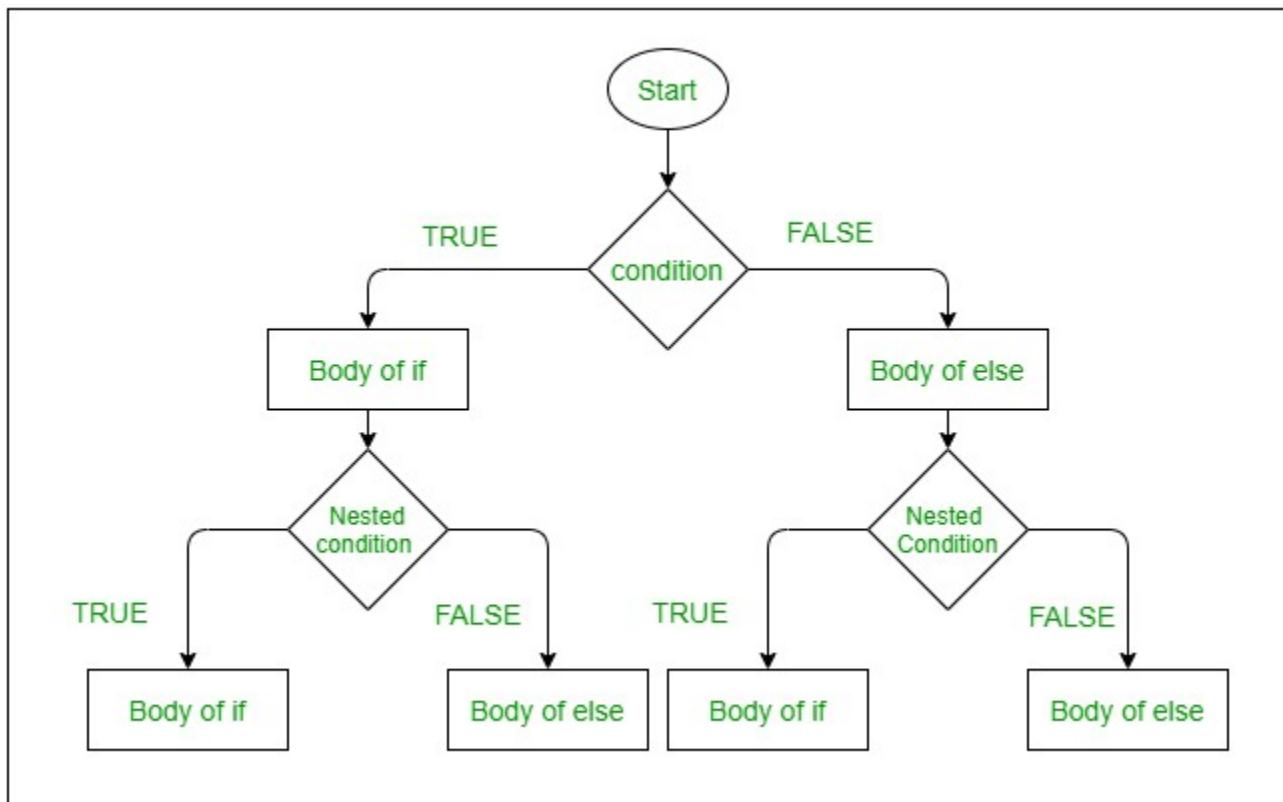
} else {
    execute this statement
}

} else {
    if(child condition 2 is true) {

        execute this statement
    } else {
        execute this statement
    }
}

```

Flow Chart:



Example:

- r

```
# R Nested if else statement Example
```

```
a <- 10
```

```
b <- 11
```

```
if(a == 10)
```

```
{
```

```
    if(b == 10)
```

```
{
```

```
        print("a:10 b:10")
```

```
    } else
```

```
{
```

```
        print("a:10 b:11")
```

```
}
```

```
} else
```

```
{
```

```
    if(a == 11)
```

```
{
```

```
        print("a:11 b:10")
```

```
    } else
```

```
{
```

```
        print("a:11 b:11")
```

```
}
```

```
}
```

Output:

```
[1] "a:10 b:11"
```

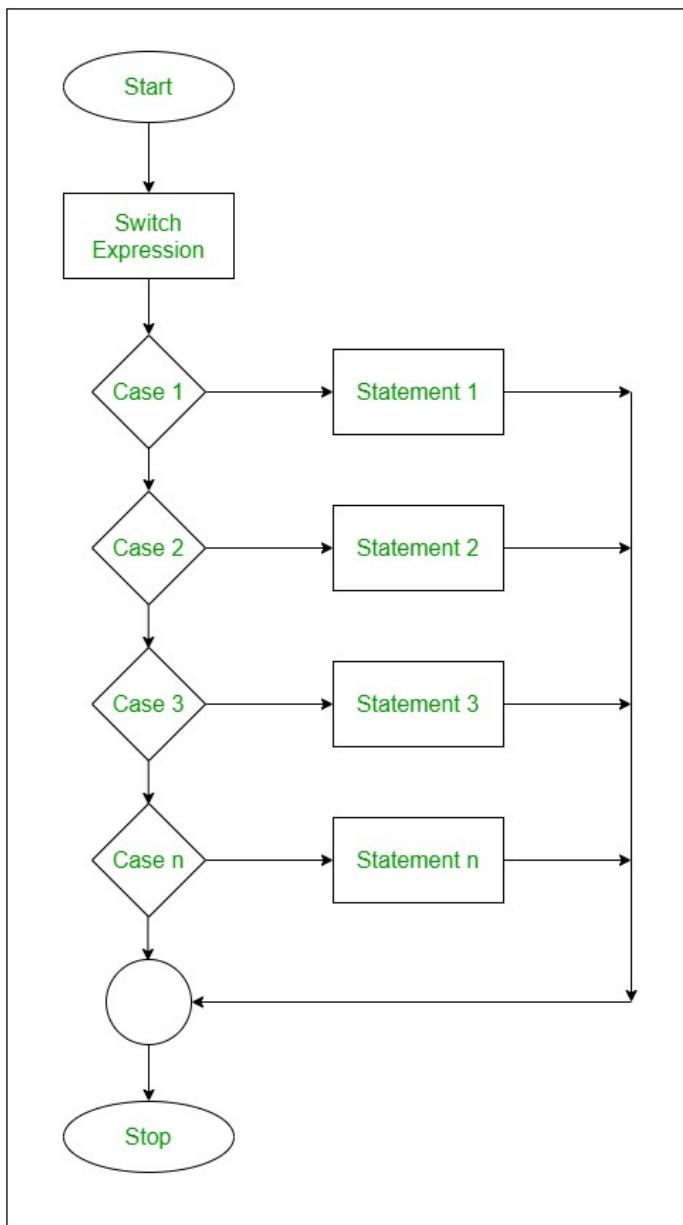
switch statement

In this [switch](#) function expression is matched to list of cases. If a match is found then it prints that case's value. No default case is available here. If no case is matched it outputs NULL as shown in example.

Syntax:

```
switch (expression, case1, case2, case3,...,case n )
```

Flow Chart :



Example:

- r

```
# R switch statement example
```

```
# Expression in terms of the index value
```

```
x <- switch(  
  2,      # Expression  
  "Geeks1", # case 1  
  "for",   # case 2  
  "Geeks2" # case 3  
)  
print(x)
```

```
# Expression in terms of the string value
```

```
y <- switch(  
  "GfG3",      # Expression  
  "GfG0"="Geeks1", # case 1  
  "GfG1"="for",   # case 2  
  "GfG3"="Geeks2" # case 3  
)  
print(y)
```

```
z <- switch(  
  "GfG",      # Expression
```

```
"GfG0"="Geeks1", # case 1  
"GfG1"="for",    # case 2  
"GfG3"="Geeks2"  # case 3  
)  
  
print(z)  
  
print(z)
```

Output:

```
[1] "for"  
[1] "Geeks2"  
NULL
```

Switch case in R

 **Read**

 **Discuss**

 **Courses**

 **Practice**

-
-
-

Switch case statements are a substitute for long if statements that compare a variable to several integral values. Switch case in R is a multiway branch statement. It allows a variable to be tested for equality against a list of values.

Switch statement follows the approach of mapping and searching over a list of values. If there is more than one match for a specific value, then the switch statement will return the first match found of the value matched with the expression.

Syntax:

```
switch(expression, case1, case2, case3....)
```

Here, the expression is matched with the list of values and the corresponding value is returned.

Important Points about Switch Case Statements:

- An expression type with character string always matched to the listed cases.
- An expression which is not a character string then this exp is coerced to integer.
- For multiple matches, the first match element will be used.
- No default argument case is available there in R switch case.
- An unnamed case can be used, if there is no matched case.

Flowchart:

Example 1:

```
# Following is a simple R program  
# to demonstrate syntax of switch.  
  
val <- switch(  
  4,  
  "Geeks1",  
  "Geeks2",  
  "Geeks3",  
  "Geeks4",  
  "Geeks5",  
  "Geeks6"  
)  
  
print(val)
```

Output:

```
[1] "Geeks4"
```

Example 2:

```
# Following is val1 simple R program  
# to demonstrate syntax of switch.  
  
# Mathematical calculation  
  
val1 = 6  
val2 = 7  
val3 = "s"
```

```

result = switch(
  val3,
  "a"= cat("Addition =", val1 + val2),
  "d"= cat("Subtraction =", val1 - val2),
  "r"= cat("Division = ", val1 / val2),
  "s"= cat("Multiplication =", val1 * val2),
  "m"= cat("Modulus =", val1 %% val2),
  "p"= cat("Power =", val1 ^ val2)
)

print(result)

```

Output:

multiplication = 42NULL

For loop in R

For loop in R Programming Language is useful to iterate over the elements of a list, data frame, [vector](#), [matrix](#), or any other object. It means the for loop can be used to execute a group of statements repeatedly depending upon the number of elements in the object. It is an entry-controlled loop, in this loop, the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

For loop in R Syntax:

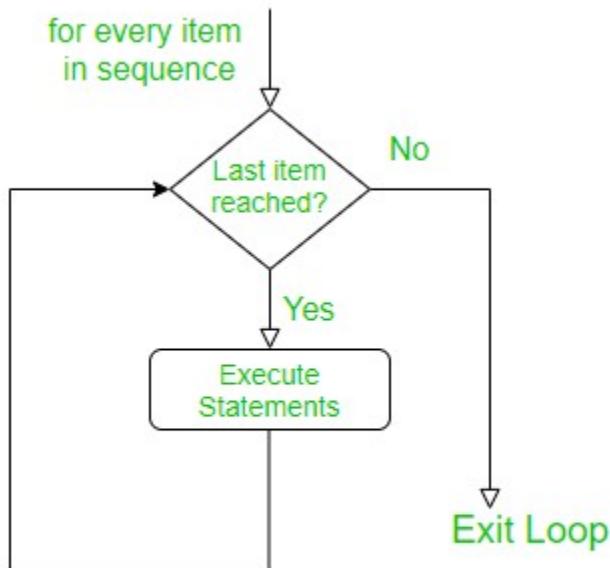
```

for (var in vector)
{
  statement(s)
}

```

Here, var takes on each value of the vector during the loop. In each iteration, the statements are evaluated.

Flowchart of For loop in R:



For Loop in R

For loop in R

Iterating over a range in R – For loop

- R

```
# R Program to demonstrate
# the use of for loop
for (i in 1: 4)
{
  print(i ^ 2)
}
```

Output:

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

[1] 16

In the above example, we iterated over the range 1 to 4 which was our vector. Now there can be several variations of this general for loop. Instead of using a sequence 1:5, we can use the concatenate function as well.

Using concatenate function in R – For loop

- R

```
# R Program to demonstrate the use of  
  
# for loop along with concatenate  
  
for (i in c(-8, 9, 11, 45))  
  
{  
  print(i)  
}
```

Output:

[1] -8

[1] 9

[1] 11

[1] 45

Instead of writing our vector inside the loop, we can also define it beforehand.

Using concatenate outside the loop R – For loop

- R

```
# R Program to demonstrate the use of  
  
# for loop with vector  
  
x <- c(-8, 9, 11, 45)  
  
for (i in x)
```

```
{  
  print(i)  
}
```

Output:

```
[1] -8  
[1] 9  
[1] 11  
[1] 45
```

Nested For-loop in R

R programming language allows using one loop inside another loop. In loop nesting, we can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa. The following section shows an example to illustrate the concept:

Example:

- R

```
# R Program to demonstrate the use of  
  
# nested for loop  
  
for (i in 1:3)  
{  
  for (j in 1:i)  
  {  
    print(i * j)  
  }  
}
```

Output:

```
[1] 1
```

```
[1] 2
```

```
[1] 4
```

```
[1] 3
```

```
[1] 6
```

```
[1] 9
```

Jump Statements in R

We use a jump statement in loops to terminate the loop at a particular iteration or to skip a particular iteration in the loop. The two most commonly used jump statements in loops are:

Break Statement:

A break statement is a jump statement that is used to terminate the loop at a particular iteration. The program then continues with the next statement outside the loop(if any).

Example:

- R

```
# R Program to demonstrate the use of  
  
# break in for loop  
  
for (i in c(3, 6, 23, 19, 0, 21))  
  
{  
  if (i == 0)  
  {  
    break  
  }  
  print(i)  
}  
  
print("Outside Loop")
```

Output:

```
[1] 3  
[1] 6  
[1] 23  
[1] 19  
[1] Outside loop
```

Here the loop exits as soon as zero is encountered.

Next Statement

It discontinues a particular iteration and jumps to the next iteration. So when next is encountered, that iteration is discarded and the condition is checked again. If true, the next iteration is executed. Hence, the next statement is used to skip a particular iteration in the loop.

Example:

- R

```
# R Program to demonstrate the use of  
  
# next in for loop  
  
for (i in c(3, 6, 23, 19, 0, 21))  
  
{  
  if (i == 0)  
  {  
    next  
  }  
  print(i)  
}  
  
print('Outside Loop')
```

Output:

```
[1] 3
```

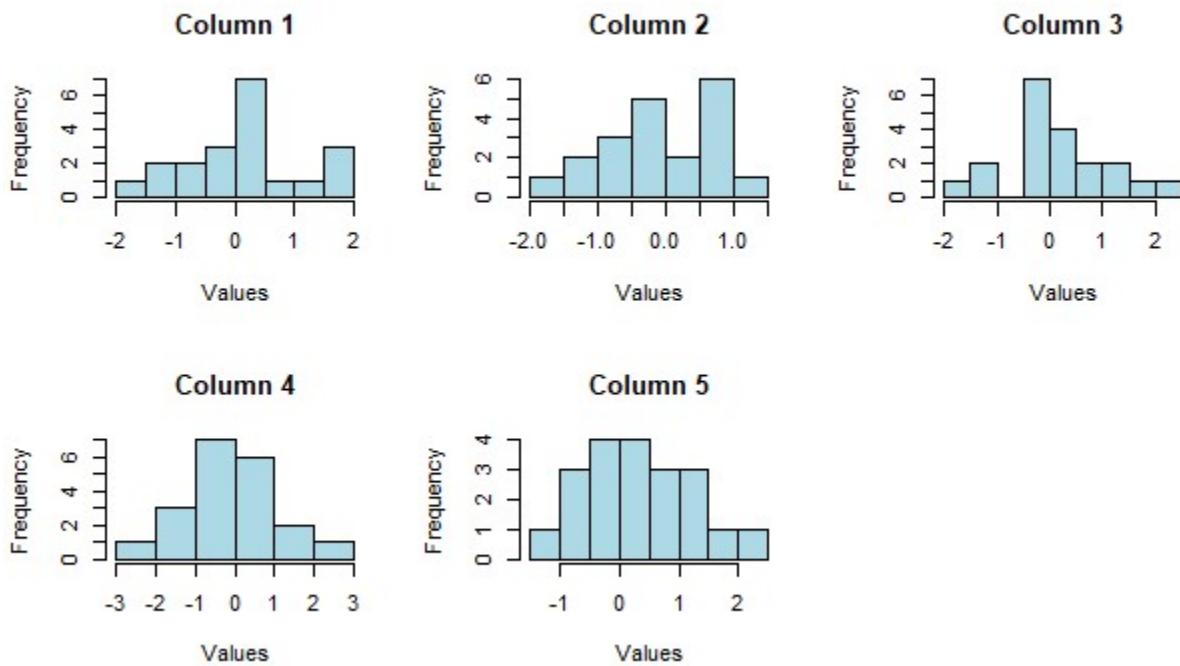
```
[1] 6  
[1] 23  
[1] 19  
[1] 21  
[1] Outside loop
```

Creating Multiple Plots within for-Loop in R

- R

```
# create a matrix of data  
  
mat <- matrix(rnorm(100), ncol = 5)  
  
  
# set up the plot layout  
  
par(mfrow = c(2, 3))  
  
  
# loop over columns of the matrix  
  
for (i in 1:5) {  
  
  # create a histogram for each column  
  
  hist(mat[, i], main = paste("Column", i), xlab = "Values", col = "lightblue")  
}
```

Output:



For loop in R

In this example, the **for** loop iterates over the columns of the matrix **mat**, and for each column, a histogram of the values is created using the **hist()** function. The **main** argument of the **hist()** function is used to set the title of each plot, and the **xlab** argument is used to label the x-axis. The **col** argument is used to set the color of the bars in the histogram to light blue.

The **par()** function is used to set up the plot layout with **mflow = c(2, 3)**, which specifies that the plots should be arranged in 2 rows and 3 columns. This means that the **for** loop will create 5 plots, each of which is a histogram of one of the columns of the matrix **mat**, arranged in a 2×3 grid.

Here as soon as zero is encountered, that iteration is discontinued and the condition is checked again. Since 21 is not equal to 0, it is printed. As we can conclude from the above two programs the basic difference between the two jump statements is that the break statement terminates the loop and the next statement skips a particular iteration of the loop.

R – while loop

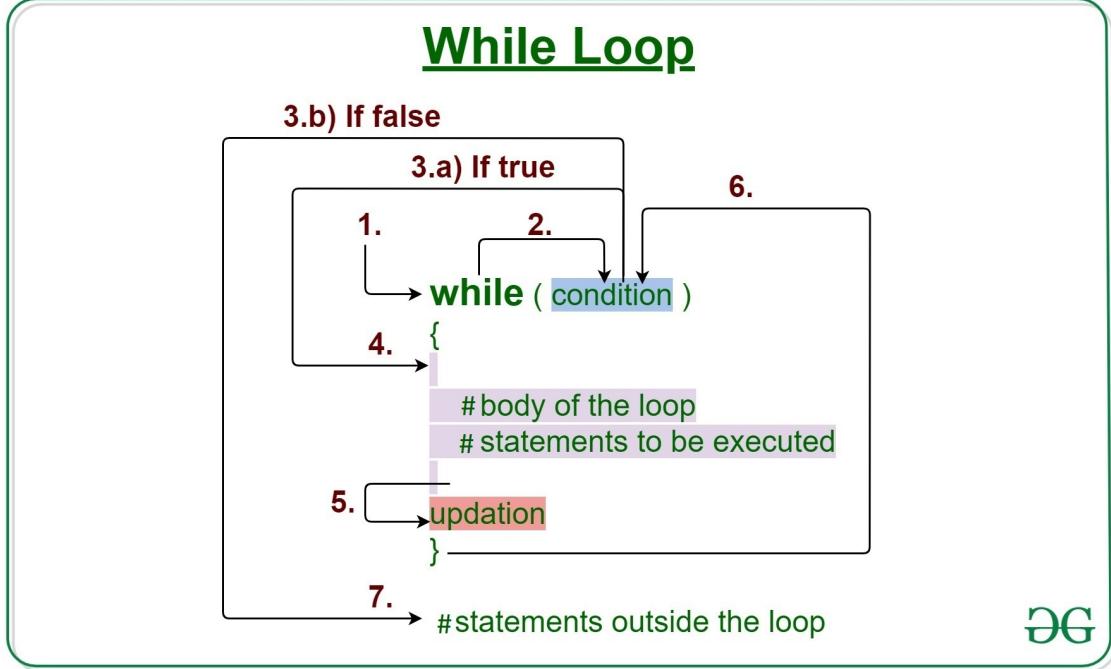
While loop in R programming language is used when the exact number of iterations of a **loop** is not known beforehand. It executes the same code again and again until a stop condition is met. While loop checks for the condition to be true or false **n+1** times rather than **n** times. This is because the while loop checks for the condition before entering the body of the loop.

R- While loop Syntax:

```

while (test_expression) {
  statement
  update_expression
}

```



DG

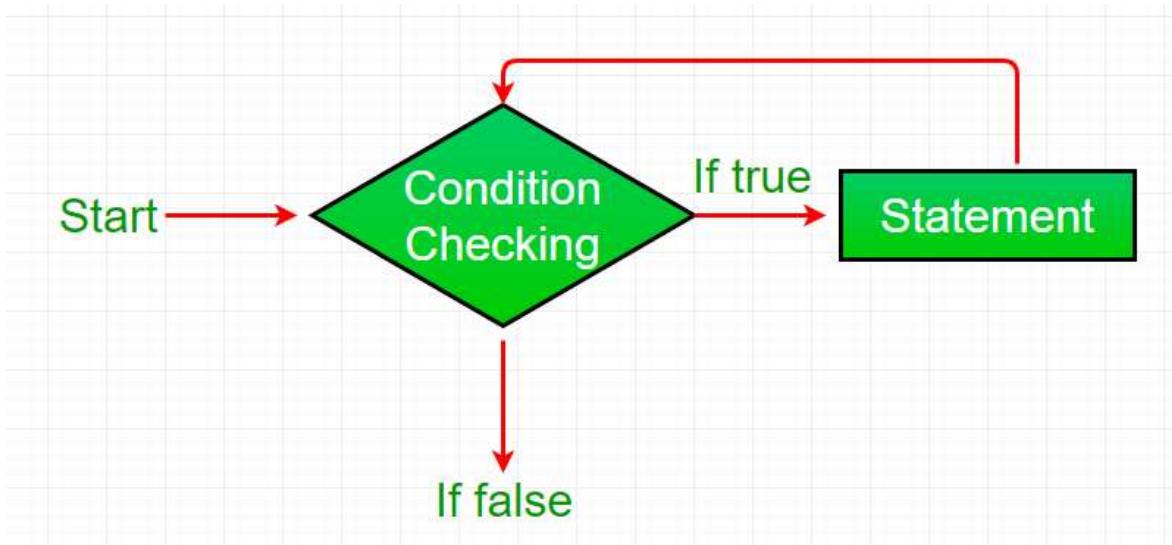
How does a While loop execute?

- Control falls into the while loop.
- The flow jumps to Condition
- Condition is tested.
 - If the Condition yields true, the flow goes into the Body.
 - If the Condition yields false, the flow goes outside the loop
- The statements inside the body of the loop get executed.
- Updation takes place.
- Control flows back to Step 2.
- The while loop has ended and the flow has gone outside.

Important Points about while loop in R language:

- It seems to be that while the loop will run forever but it is not true, condition is provided to stop it.
- When the condition is tested and the result is false then the loop is terminated.
- And when the tested result is True, then the loop will continue its execution.

R – while loop Flowchart:



R – while loop

While Loop in R Programming Examples

Example 1:

- R

```
# R program to illustrate while loop
```

```
result <- c("Hello World")
```

```
i <- 1
```

```
# test expression
```

```
while (i < 6) {
```

```
print(result)

# update expression
i = i + 1
}
```

Output:

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

Example 2:

- R

```
# R program to illustrate while loop

result <- 1
i <- 1

# test expression
while (i < 6) {

  print(result)
```

```
# update expression  
i = i + 1  
  
result = result + 1  
  
}
```

Output:

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

R – while loop break

Here we will use the break statement in the R programming language. The break statement in R is used to bring the control out of the loop when some external condition is triggered.

- R

```
# R program to illustrate while loop
```

```
result <- c("Hello World")
```

```
i <- 1
```

```
# test expression
```

```
while (i < 6) {
```

```
    print(result)
```

```
if( i == 3){  
  break}  
  
# update expression  
  
i = i + 1  
  
}
```

Output:

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

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

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

R – while loop next

- R

```
# Set an initial value for a variable
```

```
x <- 1
```

```
# Loop while x is less than 10
```

```
while (x < 10) {
```

```
  if (x == 3) {
```

```
    # Skip iteration when x is 3
```

```
    x <- x + 1
```

```
    next
```

```
}
```

```
print(paste("The current value of x is:", x))
```

```
x <- x + 1  
}  
  
# Print a message after the loop has finished  
print("The loop has ended.")
```

Output

```
[1] "The current value of x is: 1"  
[1] "The current value of x is: 2"  
[1] "The current value of x is: 4"  
[1] "The current value of x is: 5"  
[1] "The current value of x is: 6"  
[1] "The current value of x is: 7"  
[1] "The current value of x is: 8"  
[1] "The current value of x is: 9"
```

In this instance, x's initial value is set to 1 at the beginning. Then, as long as x is less than 10, we continue iterating using a while-loop. We use an if statement inside the loop to see if x equals 3. If so, the loop's current iteration is skipped in favor of the following one using the next statement. If not, we use the $x - x + 1$ expression to increment x by 1 and output a message stating the current value of x.

The next line instructs R to move on to the next iteration of the loop and skip the current one, based on a condition, over a subset of the loop's iterations without ever leaving the loop.

R While Loop with If .. Else

- R

```
x <- 1  
  
while (x <= 10) {  
  if (x == 3) {  
    next  
  }  
  print(paste("The current value of x is:", x))  
  x <- x + 1  
}
```

```
if (x %% 2 == 0) {  
  print(paste(x, "is even"))  
}  
else {  
  print(paste(x, "is odd"))  
}  
  
x <- x + 1  
}
```

Output

```
[1] "1 is odd"  
[1] "2 is even"  
[1] "3 is odd"  
[1] "4 is even"  
[1] "5 is odd"  
[1] "6 is even"  
[1] "7 is odd"  
[1] "8 is even"  
[1] "9 is odd"  
[1] "10 is even"
```

In this illustration, we initialize a variable `x` to 1, and then we iterate through the integers 1 through 10 using a while loop. We utilize an if-else statement inside the loop to determine whether `x` is even or odd. We publish a message stating that `x` is even if it is. We print a message noting that `x` is unusual if it is. Then, until `x` is more than 10, we increase `x` by 1 and loop till `x` is greater than 10.

R – Repeat loop

Repeat loop in R is used to iterate over a block of code multiple number of times. And also it executes the same code again and again until a break statement is found.

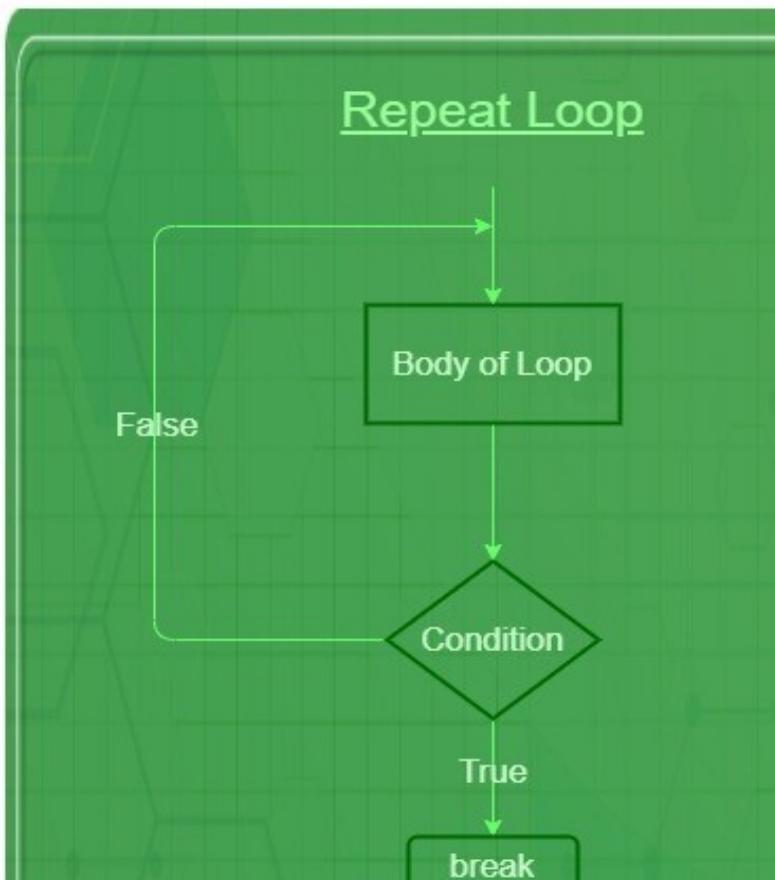
Repeat loop, unlike other loops, doesn't use a condition to exit the loop instead it looks for a **break** statement that executes if a condition within the loop body results to be true. An infinite loop in

R can be created very easily with the help of the Repeat loop. The keyword used for the repeat loop is 'repeat'.

Syntax:

```
repeat {  
  commands  
  if(condition) {  
    break  
  }  
}
```

Flowchart:



Example 1:

```
# R program to illustrate repeat loop
```

```
result <- c("Hello World")
```

```
i <- 1
```

```
# test expression
```

```
repeat {
```

```
    print(result)
```

```
    # update expression
```

```
    i <- i + 1
```

```
    # Breaking condition
```

```
    if(i >5) {
```

```
        break
```

```
}
```

```
}
```

Output:

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

Example 2:

```
# R program to illustrate repeat loop
```

```
result <- 1
```

```
i <- 1
```

```
# test expression
```

```
repeat {
```

```
    print(result)
```

```
    # update expression
```

```
    i <- i + 1
```

```
    result = result + 1
```

```
    # Breaking condition
```

```
    if(i > 5) {
```

```
        break
```

```
    }
```

```
}
```

Output:

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

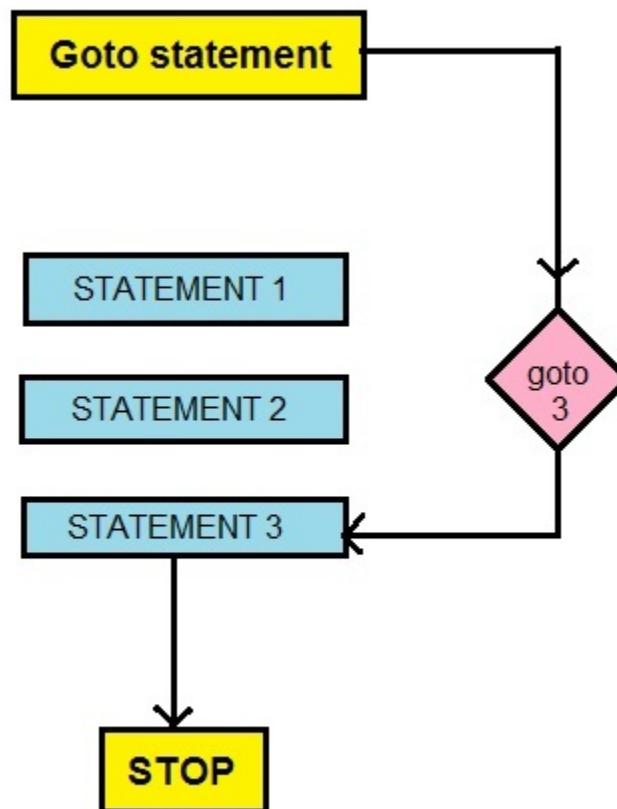
goto statement in R Programming

Goto statement in a general programming sense is a command that takes the code to the specified line or block of code provided to it. This is helpful when the need is to jump from one programming section to the other without the use of functions and without creating an abnormal shift.

Unfortunately, [R](#) doesn't support goto but its algorithm can be easily converted to depict its application. By using following methods this can be carried out more smoothly:

- Use of if and else
- Using break, next and return

Flowchart



1. Goto encountered
2. Jump to the specified line number/ name of the code block
3. Execute code

Example 1: Program to check for even and odd numbers

```
a <- 4  
if ((a %% 2) == 0)  
{  
    print("even")  
}  
else  
{  
    print("odd")  
}
```

Output:

```
[1] "even"
```

Explanation:

- **With goto:**

1. Two blocks named EVEN and ODD
2. Evaluate for a
3. if even, goto block named EVEN
4. if odd, goto block named ODD

- **Without goto:**

1. Evaluate for a
2. if even, run the statement within if block
3. if odd, run the statement within else block

Example 2: Program to check for prime numbers

```
a <- 16
b <- a/2
flag <- 0
i <- 2
repeat
{
  if ((a %% i)== 0)
  {
    flag <- 1
    break
  }
}

if (flag == 1)
{
  print("composite")
}
else
{
  print("prime")
}
```

Output:

```
[1] "composite"
```

Explanation:

- **With goto :**

1. This doesn't require flag and if statement to check flag.
 2. Evaluate for a
 3. If a factor is found take the control to the line number that has the print statement – print("composite").
 4. If not take it to the line number that has statement – print("prime")
- **Without goto:**
 1. Evaluate for a
 2. If factor found, change flag
 3. when the loop is complete check flag
 4. Print accordingly

Note: Since R doesn't have the concept of the goto statement, the above examples were made using simple if-else and break statements.

Break and Next statements in R

In R programming, we require a control structure to run a block of code multiple times. Loops come in the class of the most fundamental and strong programming concepts. A loop is a control statement that allows multiple executions of a statement or a set of statements. The word 'looping' means cycling or iterating.

Jump statements are used in loops to terminate the loop at a particular iteration or to skip a particular iteration in the loop. The two most commonly used jump statements in loops are:

- Break Statement
- Next Statement

Note: In R language continue statement is referred to as the next statement.

The basic Function of Break and Next statement is to alter the running loop in the program and flow the control outside of the loop. In R language, repeat, for and while loops are used to run the statement or get the desired output N number of times until the given condition to the loop becomes false.

Sometimes there will be such a condition where we need to terminate the loop to continue with the rest of the program. In such cases R Break statement is used.

Sometimes there will be such condition where we don't want loop to perform the program for specific condition inside the loop. In such cases, R next statement is used.

Break Statement

The break keyword is a jump statement that is used to terminate the loop at a particular iteration.

Syntax:

```
if (test_expression) {  
  break  
}
```

Example 1: Using break in For-loop

```
# R program for break statement in For-loop  
  
no <- 1:10  
  
for (val in no)  
{  
  if (val == 5)  
  {  
    print(paste("Coming out from for loop Where i = ", val))  
    break  
  }  
  print(paste("Values are: ", val))  
}
```

Output:

```
[1] "Values are: 1"  
[1] "Values are: 2"  
[1] "Values are: 3"  
[1] "Values are: 4"  
[1] "Coming out from for loop Where i = 5"
```

Example 2: Using break statement in While-loop

```
# R Break Statement Example

a<-1

while (a < 10)

{
  print(a)

  if(a==5)

    break

  a = a + 1

}
```

Output:

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

Next Statement

The next statement is used to skip the current iteration in the loop and move to the next iteration without exiting from the loop itself.

Syntax:

```
if (test_condition)

{
  next

}
```

Example 1: Using next statement in For-loop

```
# R Next Statement Example

no <- 1:10

for (val in no)
{
  if (val == 6)
  {
    print(paste("Skipping for loop Where i = ", val))
    next
  }
  print(paste("Values are: ", val))
}
```

Output:

```
[1] "Values are: 1"
[1] "Values are: 2"
[1] "Values are: 3"
[1] "Values are: 4"
[1] "Values are: 5"
[1] "Skipping for loop Where i = 6"
[1] "Values are: 7"
[1] "Values are: 8"
[1] "Values are: 9"
[1] "Values are: 10"
```

Example 2: Using next statement in While-loop

```
# R Next Statement Example
```

```
x <- 1  
while(x < 5)  
{  
  x <- x + 1;  
  if (x == 3)  
    next;  
  print(x);  
}
```

Output:

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

List and Data Frames: **Recursive Lists**

Lists can be recursive, meaning that you can have lists within lists. Here's an example:

```
> b <- list(u = 5, v = 12)  
> c <- list(w = 13)  
> a <- list(b,c)  
> a
```

```
[[1]]  
[[1]]$u  
[1] 5
```

```
[[1]]$v  
[1] 12
```

```
[[2]]  
[[2]]$w  
[1] 13
```

```
> length(a)  
[1] 2
```

This code makes `a` into a two-component list, with each component itself also being a list.

The concatenate function `c()` has an optional argument `recursive`, which controls whether *flattening* occurs when recursive lists are combined.

```
> c(list(a=1,b=2,c=list(d=5,e=9)))  
$a  
[1] 1
```

```
$b  
[1] 2
```

```
$c
```

```
$c$d
```

```
[1] 5
```

```
$c$e
```

```
[1] 9
```

```
> c(list(a=1,b=2,c=list(d=5,e=9)),recursive=T)
```

```
a b c.d c.e
```

```
1 2 5 9
```

In the first case, we accepted the default value of recursive, which is FALSE, and obtained a recursive list, with the c component ...

creating a datframe:

apply(), lapply(), sapply(), and tapply() in R

¶ Read

¶ Discuss

¶ Courses

¶ Practice

-
-
-

In this article, we will learn about the apply(), lapply(), sapply(), and tapply() functions in the [R Programming Language](#).

The apply() collection is a part of R essential package. This family of functions helps us to apply a certain function to a certain data frame, list, or vector and return the result as a list or vector depending on the function we use. There are these following four types of function in apply() function family:

apply() function

The apply() function lets us apply a function to the rows or columns of a matrix or data frame. This function takes matrix or data frame as an argument along with function and whether it has to be applied by row or column and returns the result in the form of a vector or array or list of values obtained.

Syntax: `apply(x, margin, function)`

Parameters:

- **x:** determines the input array including matrix.
- **margin:** If the margin is 1 function is applied across row, if the margin is 2 it is applied across the column.
- **function:** determines the function that is to be applied on input data.

Example:

Here, is a basic example showcasing the use of apply() function along rows as well as columns.

- R

```
# create sample data
sample_matrix <- matrix(C<-(1:10),nrow=3, ncol=10)

print( "sample matrix:")
sample_matrix

# Use apply() function across row to find sum
print("sum across rows:")
apply( sample_matrix, 1, sum)

# use apply() function across column to find mean
```

```
print("mean across columns:")
apply( sample_matrix, 2, mean)
```

Output:

```
[1] "sample matrix:"
```

1	4	7	10	3	6	9	2	5	8
2	5	8	1	4	7	10	3	6	9
3	6	9	2	5	8	1	4	7	10

```
[1] "sum across rows:"
```

```
55 55 55
```

```
[1] "mean across columns:"
```

lapply() function

The [lapply\(\) function](#) helps us in applying functions on list objects and returns a list object of the same length. The lapply() function in the R Language takes a list, vector, or data frame as input and gives output in the form of a list object. Since the lapply() function applies a certain operation to all the elements of the list it doesn't need a MARGIN.

Syntax: *lapply(x, fun)*

Parameters:

- *x: determines the input vector or an object.*
- *fun: determines the function that is to be applied to input data.*

Example:

Here, is a basic example showcasing the use of the lapply() function to a vector.

- R

```

# create sample data

names <- c("priyank", "abhiraj","pawananjani",
         "sudhanshu","devraj")

print( "original data:")

names

# apply lapply() function

print("data after lapply():")

lapply(names, toupper)

```

Output:

```

[1] "original data:"  

'priyank'  'abhiraj'  'pawananjani'  'sudhanshu'  'devra  

[1] "data after lapply():"  

1. 'PRIYANK'  

2. 'ABHIRAJ'  

3. 'PAWANANJANI'  

4. 'SUDHANSHU'

```

sapply() function

The [sapply\(\) function](#) helps us in applying functions on a list, vector, or data frame and returns an array or matrix object of the same length. The sapply() function in the R Language takes a list, vector, or data frame as input and gives output in the form of an array or matrix object. Since the sapply() function applies a certain operation to all the elements of the object it doesn't need a MARGIN. It is the same as lapply() with the only difference being the type of return object.

Syntax: `sapply(x, fun)`

Parameters:

- *x*: determines the input vector or an object.
- *fun*: determines the function that is to be applied to input data.

Example:

Here, is a basic example showcasing the use of the sapply() function to a vector.

- R

```
# create sample data
sample_data<- data.frame( x=c(1,2,3,4,5,6),
                           y=c(3,2,4,2,34,5))
print( "original data:")
sample_data

# apply sapply() function
print("data after sapply():")
sapply(sample_data, max)
```

Output:

```
[1] "original data:"
```

x	y
1	3
2	2
3	4
4	2
5	34
6	5

```
[1] "data after sapply():"
```

tapply() function

The [tapply\(\)](#) helps us to compute statistical measures (mean, median, min, max, etc..) or a self-written function operation for each factor variable in a vector. It helps us to create a subset of a vector and then apply some functions to each of the subsets. For example, in an organization, if we have data of salary of employees and we want to find the mean salary for male and female, then we can use tapply() function with male and female as factor variable gender.

Syntax: `tapply(x, index, fun)`

Parameters:

- **x:** determines the input vector or an object.
- **index:** determines the factor vector that helps us distinguish the data.
- **fun:** determines the function that is to be applied to input data.

Example:

Here, is a basic example showcasing the use of the tapply() function on the diamonds dataset which is provided by the tidyverse package library.

- R

```
# load library tidyverse  
library(tidyverse)  
  
# print head of diamonds dataset  
print(" Head of data:")  
head(diamonds)  
  
# apply tapply function to get average price by cut  
print("Average price for each cut of diamond:")  
tapply(diamonds$price, diamonds$cut, mean)
```

Output:

```
[1] "Head of data:"
```

carat	cut	color	clarity	depth	table	price	x	y
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84
0.23	Good	E	VS1	56.9	65	327	4.05	4.07
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23
0.31	Good	J	SI2	63.3	58	335	4.34	4.35
0.24	Very Good	J	VVS2	62.8	57	336	3.94	3.96

```
[1] "Average price for each cut of diamond:"
```

Fair	4358.75776397516
Good	3928.86445169181

Object oriented programming with R

R – Object Oriented Programming

In this article, we will discuss Object-Oriented Programming (OOPs) in [R Programming Language](#). We will discuss the S3 and S4 classes, the [inheritance](#) in these classes, and the methods provided by these classes.

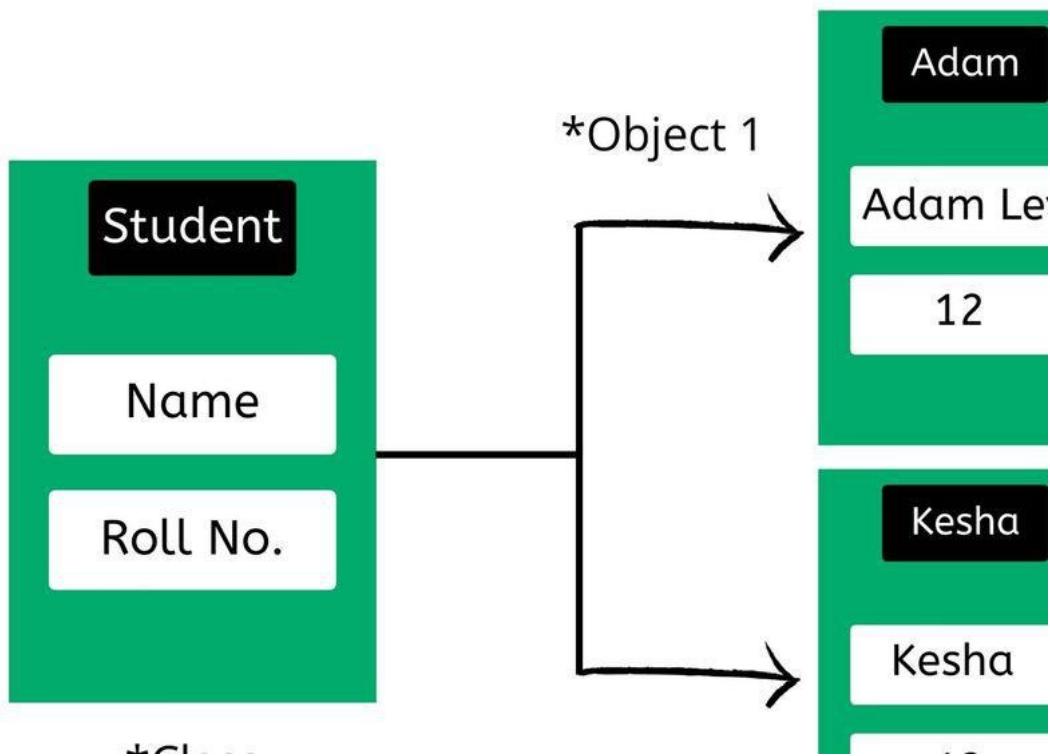
OOPs in R Programming Language:

In R programming, OOPs in R provide classes and objects as its key tools to reduce and manage the complexity of the program. R is a functional language that uses concepts of OOPs. We can think of a class as a sketch of a car. It contains all the details about the model_name, model_no, engine, etc. Based on these descriptions we select a car. The car is the object. Each car object has its own characteristics and features. An object is also called an instance of a class and the process of creating this object is

called instantiation. In R [S3](#) and S4 classes are the two most important classes for object-oriented programming. But before going discussing these classes let's see a brief about classes and objects.

Class and Object

Class is the blueprint or a prototype from which objects are made by encapsulating data members and functions. An object is a data structure that contains some methods that act upon its attributes.



R – Object Oriented Programming

S3 Class

[S3 class](#) does not have a predefined definition and is able to dispatch. In this class, the generic function makes a call to the method. Easy implementation of S3 is possible because it differs from the traditional programming language Java, C++, and C# which implements Object Oriented message passing.

Creating S3 Class

To create an object of this class we will create a list that will contain all the class members. Then this list is passed to the class() method as an argument.

Syntax:

`variable_name <- list(attribute1, attribute2, attribute3....attributeN)`

Example:

In the following code, a Student class is defined. An appropriate class name is given having attributes student's name and roll number. Then the object of the student class is created and invoked.

- R

```
# List creation with its attributes name  
  
# and roll no.  
  
a <- list(name="Adam", Roll_No=15)  
  
  
# Defining a class "Student"  
  
class(a) <- "Student"  
  
  
# Creation of object  
  
a
```

Output:

```
$name
```

```
[1] "Adam"
```

```
$Roll_No
```

```
[1] 15
```

```
attr(, "class")
```

```
[1] "Student"
```

Generic Functions

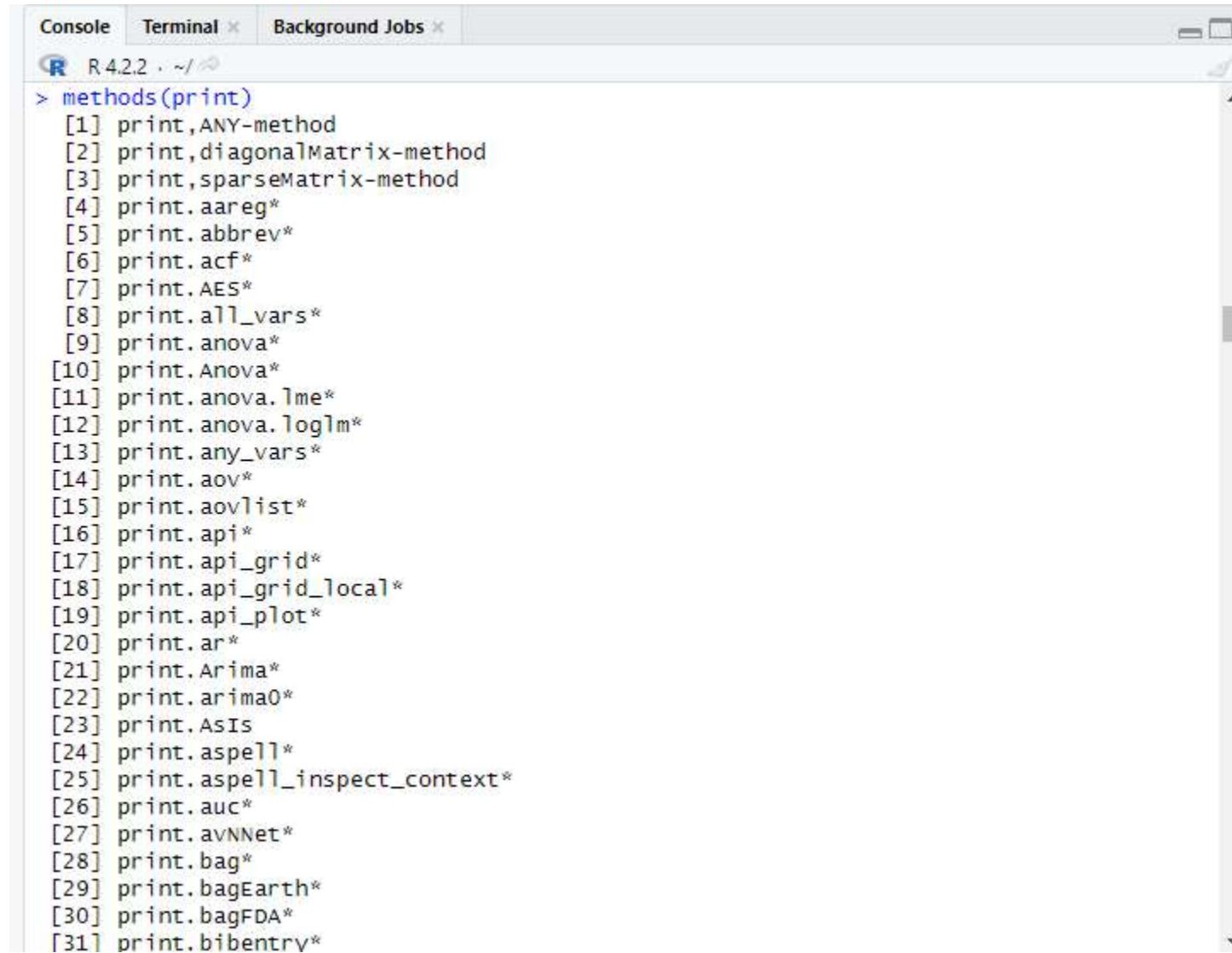
The generic functions are a good example of polymorphism. To understand the concept of generic functions consider the print() function. The print() function is a collection of various print functions that are created for different data types and data structures in the R Programming Language. It calls the appropriate function depending on the type of object passed as an argument. We can see the various implementation of print functions using the methods() function.

Example: Seeing different types of the print function

- R

```
methods(print)
```

Output:



The screenshot shows an R console window with three tabs: "Console", "Terminal", and "Background Jobs". The "Console" tab is active, displaying the command "methods(print)" followed by a list of 31 methods. The methods listed are: [1] print,ANY-method, [2] print,diagonalMatrix-method, [3] print,sparseMatrix-method, [4] print.aareg*, [5] print.abbrev*, [6] print.acf*, [7] print.AES*, [8] print.all_vars*, [9] print.anova*, [10] print.Anova*, [11] print.anova.lme*, [12] print.anova.loglm*, [13] print.any_vars*, [14] print.aov*, [15] print.aovlist*, [16] print.api*, [17] print.api_grid*, [18] print.api_grid_local*, [19] print.api_plot*, [20] print.ar*, [21] print.Arima*, [22] print.arima0*, [23] print.ASIS, [24] print.aspell*, [25] print.aspell_inspect_context*, [26] print.auc*, [27] print.avNNet*, [28] print.bag*, [29] print.bagEarth*, [30] print.bagFDA*, [31] print.bibentry*. The method at index 31 is highlighted in blue.

```
> methods(print)
[1] print,ANY-method
[2] print,diagonalMatrix-method
[3] print,sparseMatrix-method
[4] print.aareg*
[5] print.abbrev*
[6] print.acf*
[7] print.AES*
[8] print.all_vars*
[9] print.anova*
[10] print.Anova*
[11] print.anova.lme*
[12] print.anova.loglm*
[13] print.any_vars*
[14] print.aov*
[15] print.aovlist*
[16] print.api*
[17] print.api_grid*
[18] print.api_grid_local*
[19] print.api_plot*
[20] print.ar*
[21] print.Arima*
[22] print.arima0*
[23] print.ASIS
[24] print.aspell*
[25] print.aspell_inspect_context*
[26] print.auc*
[27] print.avNNet*
[28] print.bag*
[29] print.bagEarth*
[30] print.bagFDA*
[31] print.bibentry*
```

Print methods

Now let's create a generic function of our own. We will create the print function for our class that will print all the members in our specified format. But before creating a print function let's create what the print function does to our class.

Example:

- R

```
# List creation with its attributes name  
  
# and roll no.  
  
a = list(name="Adam", Roll_No=15)  
  
  
# Defining a class "Student"  
  
class(a) = "Student"  
  
  
# Creation of object  
  
print(a)
```

Output:

\$name

[1] "Adam"

\$Roll_No

[1] 15

attr(,"class")

[1] "Student"

Now let's print all the members in our specified format. Consider the below example –

Example:

- R

```
print.Student <- function(obj){
```

```
cat("name: " ,obj$name, "\n")
cat("Roll No: ", obj$Roll_No, "\n")
}

print(a)
```

Output:

name: Adam

Roll No: 15

Attributes

[Attributes](#) of an object do not affect the value of an object, but they are a piece of extra information that is used to handle the objects. The function attributes() can be used to view the attributes of an object.

Examples: An S3 object is created and its attributes are displayed.

- R

```
attributes(a)
```

Output:

\$names

'name' 'Roll_No'

\$class

'Student'

Also, you can add attributes to an object by using attr.

- R

```
attr(a, "age")<-c(18)
```

```
attributes(a)
```

Output:

\$names

'name"Roll_No'

\$class

'Student'

\$age

18

Inheritance in S3 Class

Inheritance is an important concept in OOP(object-oriented programming) which allows one class to derive the features and functionalities of another class. This feature facilitates code-reusability.

S3 class in R programming language has no formal and fixed definition. In an S3 object, a list with its class attribute is set to a class name. S3 class objects inherit only methods from their base class.

Example:

In the following code, inheritance is done using S3 class, firstly the object is created of the class student.

- R

```
# student function with argument  
  
# name(n) and roll_no(r)  
  
student <- function(n, r) {  
  
  value <- list(name=n, Roll=r)  
  
  attr(value, "class") <- "student"  
  
  value  
  
}
```

Then, the method is defined to print the details of the student.

- R

```
# 'print.student' method created

print.student <- function(obj) {

  # 'cat' function is used to concatenate

  # strings

  cat("Name:", obj$name, "\n")

  cat("Roll", obj$roll, "\n")}
```

Now, inheritance is done while creating another class by doing ***class(obj) <- c(child, parent)***.

- R

```
s <- list(name="Kesha", Roll=21, country="India")

# child class 'Student' inherits

# parent class 'student'

class(s) <- c("Student", "student")

s
```

Output:

Name: Kesha

Roll: 21

The following code overwrites the method for class students.

- R

```
# 'Student' class object is passed

# in the function of class 'student'
```

```
print.student <- function(obj) {  
  cat(obj$name, "is from", obj$country, "\n")  
}  
  
s
```

Output:

Kesha is from India

S4 Class

S4 class has a predefined definition. It contains functions for defining methods and generics. It makes multiple dispatches easy. This class contains auxiliary functions for defining methods and generics.

Creating S4 class and object

setClass() command is used to create S4 class. Following is the syntax for setclass command which denotes myclass with slots containing name and rollno.

Syntax:

```
setClass("myclass", slots=list(name="character", Roll_No="numeric"))
```

The **new()** function is used to create an object of the S4 class. In this function, we will pass the class name as well as the value for the slots.

Example:

- R

```
# Function setClass() command used  
  
# to create S4 class containing list of slots.  
  
setClass("Student", slots=list(name="character",  
                                Roll_No="numeric"))
```

```
# 'new' keyword used to create object of  
  
# class 'Student'  
  
a <- new("Student", name="Adam", Roll_No=20)  
  
  
# Calling object  
  
a
```

Output:

```
Slot "name":  
  
[1] "Adam"
```

```
Slot "Roll_No":  
  
[1] 20
```

Create S4 objects from the generator function

`setClass()` returns a generator function that helps in creating objects and it acts as a constructor.

Example:

- R

```
stud <- setClass("Student", slots=list(name="character",  
  
                               Roll_No="numeric"))  
  
  
# Calling object  
  
stud
```

Output:

```
class generator function for class "Student" from package '.GlobalEnv'  
  
function (...)
```

```
new("Student", ...)
```

Now the above-created stud function will act as the constructor for the Student class. It will behave as the new() function.

Example:

- R

```
stud(name="Adam", Roll_No=15)
```

Output:

An object of class "Student"

Slot "name":

```
[1] "Adam"
```

Slot "Roll_No":

```
[1] 15
```

Inheritance in S4 class

S4 class in R programming have proper definition and derived classes will be able to inherit both attributes and methods from its base class. For this, we will first create a base class with appropriate slots and will create a generic function for that class. Then we will create a derived class that will inherit using the contains parameter. The derived class will inherit the members as well as functions from the base class.

Example:

- R

```
# Define S4 class  
  
setClass("student",  
  
  slots=list(name="character",  
  
            age="numeric", rno="numeric"))
```

```

)
}

# Defining a function to display object details

setMethod("show", "student",
  function(obj){
    cat(obj@name, "\n")
    cat(obj@age, "\n")
    cat(obj@rno, "\n")
  }
)

# Inherit from student

setClass("InternationalStudent",
  slots=list(country="character"),
  contains="student"
)

# Rest of the attributes will be inherited from student

s <- new("InternationalStudent", name="Adam",
  age=22, rno=15, country="India")

show(s)

```

Output:

Adam

22

15

The reasons for defining both S3 and S4 classes are as follows:

- S4 class alone will not be seen if the S3 generic function is called directly. This will be the case, for example, if some function calls **unique()** from a package that does not make that function an S4 generic.
- However, primitive functions and operators are exceptions: The internal C code will look for S4 methods if and only if the object is an S4 object. S4 method dispatch would be used to dispatch any binary operator calls where either of the operands was an S4 object.
- S3 class alone will not be called if there is any eligible non-default S4 method.

So if a package defined an S3 method for unique for an S4 class but another package defined an S4 method for a superclass of that class, the superclass method would be chosen, probably not what was intended.

Classes in R Programming

Classes and Objects are basic concepts of Object-Oriented Programming that revolve around the real-life entities. Everything in R is an object. An **object** is simply a data structure that has some methods and attributes. A **class** is just a blueprint or a sketch of these objects. It represents the set of properties or methods that are common to all objects of one type.

Unlike most other programming languages, R has a three-class system. These are S3, S4, and Reference Classes.

S3 Class

S3 is the simplest yet the most popular OOP system and it lacks formal definition and structure. An object of this type can be created by just adding an attribute to it. Following is an example to make things more clear:

Example:

```
# create a list with required components  
  
movieList <- list(name = "Iron man", leadActor = "Robert Downey Jr")  
  
# give a name to your class
```

```
class(movieList) <- "movie"
```

```
movieList
```

Output:

```
$name
```

```
[1] "Iron man"
```

```
$leadActor
```

```
[1] "Robert Downey Jr"
```

In S3 systems, methods don't belong to the class. They belong to generic functions. It means that we can't create our own methods here, as we do in other programming languages like C++ or Java. But we can define what a generic method (for example print) does when applied to our objects.

```
print(movieList)
```

Output:

```
$name
```

```
[1] "Iron man"
```

```
$leadActor
```

```
[1] "Robert Downey Jr"
```

Example: Creating a user-defined print function

```
# now let us write our method
```

```
print.movie <- function(obj)
```

```
{
```

```
  cat("The name of the movie is", obj$name, ".\n")
```

```
  cat(obj$leadActor, "is the lead actor.\n")
```

```
}
```

Output:

The name of the movie is Iron man .

Robert Downey Jr is the lead actor.

S4 Class

Programmers of other languages like C++, Java might find S3 to be very much different than their normal idea of classes as it lacks the structure that classes are supposed to provide. S4 is a slight improvement over S3 as its objects have a proper definition and it gives a proper structure to its objects.

Example:

```
library(methods)

# definition of S4 class

setClass("movies", slots=list(name="character", leadActor = "character"))

# creating an object using new() by passing class name and slot values

movieList <- new("movies", name="Iron man", leadActor = "Robert Downey Jr")

movieList
```

Output:

An object of class "movies"

Slot "name":

[1] "Iron man"

Slot "leadActor":

[1] "Robert Downey Jr"

As shown in the above example, **setClass()** is used to define a class and **new()** is used to create the objects.

The concept of methods in S4 is similar to S3, i.e., they belong to generic functions. The following example shows how to create a method:

```
movieList
```

Output:

An object of class "movies"

Slot "name":

```
[1] "Iron man"
```

Slot "leadActor":

```
[1] "Robert Downey Jr"
```

Example:

```
# using setMethod to set a method

setMethod("show", "movies",
          function(object)
{
  cat("The name of the movie is ", object@name, ".\n")
  cat(object@leadActor, "is the lead actor.\n")
}
)

movieList
```

Output:

```
[1] "show"
```

The name of the movie is Iron man .

Robert Downey Jr is the lead actor.

Reference Class

Reference Class is an improvement over S4 Class. Here the methods belong to the classes. These are much similar to object-oriented classes of other languages.

Defining a Reference class is similar to defining S4 classes. We use **setRefClass()** instead of **setClass()** and “fields” instead of “slots”.

Example:

```
library(methods)

# setRefClass returns a generator

movies <- setRefClass("movies", fields = list(name = "character",
                                               leadActor = "character", rating = "numeric"))

#now we can use the generator to create objects

movieList <- movies(name = "Iron Man",
                      leadActor = "Robert downey Jr", rating = 7)

movieList
```

Output:

Reference class object of class "movies"

Field "name":

[1] "Iron Man"

Field "leadActor":

[1] "Robert downey Jr"

Field "rating":

[1] 7

Now let us see how to add some methods to our class with an example.

Example

```
library(methods)

movies <- setRefClass("movies", fields = list(name = "character",
                                              leadActor = "character", rating = "numeric"), methods = list(
    increment_rating = function()
    {
        rating <<- rating + 1
    },
    decrement_rating = function()
    {
        rating <<- rating - 1
    }
))

movieList <- movies(name = "Iron Man",
                     leadActor = "Robert downey Jr", rating = 7)

# print the value of rating
movieList$rating

# increment and then print the rating
movieList$increment_rating()

movieList$rating
```

```
# decrement and print the rating  
  
movieList$decrement_rating()  
  
movieList$rating
```

Output:

[1] 7

[1] 8

[1] 7

R – Objects

¶ Read

¶ Discuss

¶ Courses

¶ Practice

-
-
-

Every programming language has its own data types to store values or any information so that the user can assign these data types to the variables and perform operations respectively. Operations are performed accordingly to the data types.

These data types can be character, integer, float, long, etc. Based on the data type, memory/storage is allocated to the variable. For example, in C language character variables are assigned with 1 byte of memory, integer variable with 2 or 4 bytes of memory and other data types have different memory allocation for them.

Unlike other programming languages, variables are assigned to objects rather than data types in [R programming](#).

Type of Objects

There are 5 basic types of objects in the R language:

Vectors

Atomic [vectors](#) are one of the basic types of objects in R programming. Atomic vectors can store homogeneous data types such as character, doubles, integers, raw, logical, and complex. A single element variable is also said to be vector.

Example:

```
# Create vectors  
  
x <- c(1, 2, 3, 4)  
  
y <- c("a", "b", "c", "d")  
  
z <- 5  
  
  
# Print vector and class of vector  
  
print(x)  
  
print(class(x))  
  
  
print(y)  
  
print(class(y))  
  
  
print(z)  
  
print(class(z))
```

Output:

```
[1] 1 2 3 4  
  
[1] "numeric"  
  
[1] "a" "b" "c" "d"  
  
[1] "character"  
  
[1] 5  
  
[1] "numeric"
```

Lists

[List](#) is another type of object in R programming. List can contain heterogeneous data types such as vectors or another lists.

Example:

```
# Create list  
ls <- list(c(1, 2, 3, 4), list("a", "b", "c"))
```

```
# Print  
print(ls)  
print(class(ls))
```

Output:

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

```
[[2]]  
[[2]][[1]]  
[1] "a"
```

```
[[2]][[2]]  
[1] "b"
```

```
[[2]][[3]]  
[1] "c"  
  
[1] "list"
```

Matrices

To store values as 2-Dimensional array, matrices are used in R. Data, number of rows and columns are defined in the `matrix()` function.

Syntax:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Example:

```
x <- c(1, 2, 3, 4, 5, 6)
```

```
# Create matrix
```

```
mat <- matrix(x, nrow = 2)
```

```
print(mat)
```

```
print(class(mat))
```

Output:

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

```
[1, ] 1 3 5
```

```
[2, ] 2 4 6
```

```
[1] "matrix"
```

Factors

[Factor](#) object encodes a vector of unique elements (levels) from the given data vector.

Example:

```
# Create vector
```

```
s <- c("spring", "autumn", "winter", "summer",
```

```
"spring", "autumn")
```

```
print(factor(s))
```

```
print(nlevels(factor(s)))
```

Output:

```
[1] spring autumn winter summer spring autumn
```

```
Levels: autumn spring summer winter
```

```
[1] 4
```

Arrays

array() function is used to create n-dimensional array. This function takes dim attribute as an argument and creates required length of each dimension as specified in the attribute.

Syntax:

```
array(data, dim = length(data), dimnames = NULL)
```

Example:

```
# Create 3-dimensional array
```

```
# and filling values by column
```

```
arr <- array(c(1, 2, 3), dim = c(3, 3, 3))
```

```
print(arr)
```

Output:

```
,, 1
```

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

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

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

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

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

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

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

```
[3, ]  3  3  3,, 3
```

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

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

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

```
[3, ]  3  3  3
```

Data Frames

[Data frames](#) are 2-dimensional tabular data object in R programming. Data frames consists of multiple columns and each column represents a vector. Columns in data frame can have different modes of data unlike matrices.

Example:

```
# Create vectors  
  
x <- 1:5  
  
y <- LETTERS[1:5]  
  
z <- c("Albert", "Bob", "Charlie", "Denver", "Elie")
```

```
# Create data frame of vectors
```

```
df <- data.frame(x, y, z)
```

```
# Print data frame
```

```
print(df)
```

Output:

```
x y z  
1 1 A Albert  
2 2 B Bob  
3 3 C Charlie  
4 4 D Denver  
5 5 E Elie
```

Polymorphism in R Programming

[R language](#) is evolving and it implements parametric polymorphism, which means that methods in R refer to functions, not classes. Parametric polymorphism primarily lets you define a generic method or function for types of objects you haven't yet defined and may never do. This means that one can use the same name for several functions with different sets of arguments and from various classes. R's method call mechanism is generics which allows registering certain names to be treated as methods in R, and they act as dispatchers.

Generic Functions

Polymorphism in R can be obtained by the generics. It allows certain names to be treated as methods and they act as dispatchers. Let's understand with the help of `plot()` function and `summary` function. In R programming the `plot()` and [summary\(\)](#) functions return different results depending on the objects being passed to them and that's why they are generic functions that implement polymorphism.

`plot()` in R

`plot()` is one of the generic functions that implement polymorphism in R. It produces a different graph if it is given a [vector](#), a [factor](#), a [data frame](#), etc. But have you ever wondered how does the class of vectors or factors determine the method used for plotting? Let's see the code for the `plot` function.

Example: Code of `plot` function

- R

```
plot
```

Output:

```
function (x, y, ...)
```

```
UseMethod("plot")
```

We can see that the body of the plot function contains only one expression and that is `UseMethod("plot")`. Let's see the definition of `UseMethod` with the help of `help()` function.

Example: Definition of the `help()` method.

- R

```
help(UseMethod)
```

Output:

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the argument to the generic function or of the object supplied as an argument to `UseMethod` or `NextMethod`.

Usage

```
UseMethod(generic, object)  
NextMethod(generic = NULL, object = NULL, ...)
```

Arguments

generic: a character string naming a function (and not a built-in operator). Required for `UseMethod`.

From the above output, we can see that `UseMethod` takes two parameters `generic` and `object`.

- The `generic` is the string name which is the name of the function (`plot` in this case).
- This is an object whose class will determine the method that will be “dispatched,” It means the object for which the generic method will be called.

The `UseMethod` then searches for the suitable `plot` function that is needed to be called by creating a string of the type `plot.object`. We can also see all the available methods for the `plot` function.

Example:

- R

```
methods(plot)
```

Output:

```
[1] plot.acf*          plot.data.frame*    plot.decomposed.ts*
[4] plot.default       plot.dendrogram*   plot.density*
[7] plot.ecdf          plot.factor*      plot.formula*
[10] plot.function     plot.hclust*      plot.histogram*
[13] plot.HoltWinters* plot.isoreg*      plot.lm*
[16] plot.medpolish*   plot.mlm*        plot.ppr*
[19] plot.prcomp*       plot.princomp*   plot.profile.nls*
[22] plot.raster*       plot.spec*       plot.stepfun
[25] plot.stl*          plot.table*      plot.ts
[28] plot.tskernel*    plot.TukeyHSD*
see '?methods' for accessing help and source code
```

Let's see how **plot()** function taking arguments and displaying different outputs

Input is one numeric vector

In this example let's take a single numeric vector inside **plot()** function as a parameter.

- R

```
# R program to illustrate  
  
# polymorphosim  
  
# X Window System Graphics (X11)  
  
X11(width = 15, height = 13)  
  
# The runif() function generates  
  
# random deviates of the uniform distribution  
  
x <- 3 * runif(40) + (1:30)  
  
par(mar = c(20, 20, 1, 1))
```

```

# type='l' is used to connect the points

# of the scatter plots with lines.

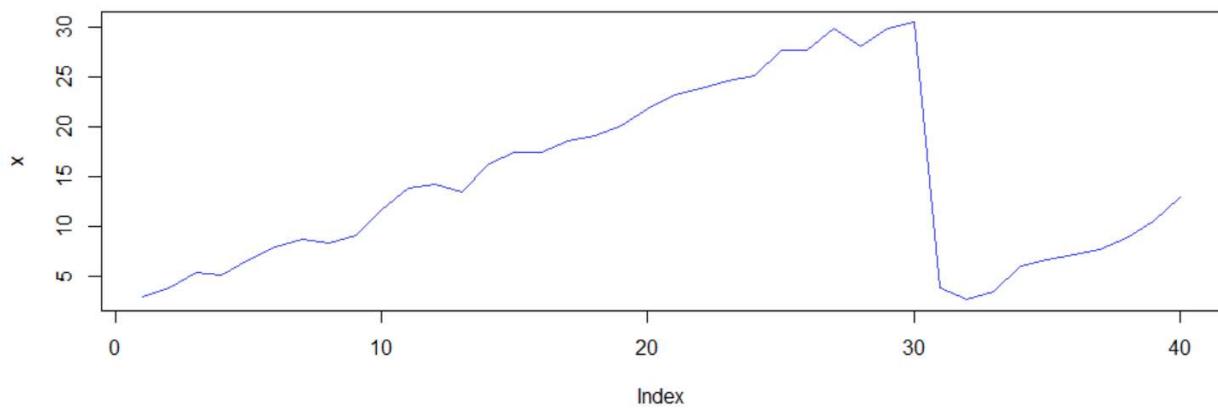
plot(x, type = 'l', col = '#343deb')

# We can do mouse click or enter pressed

z <- locator(1)

```

Output:



Inputs are two numeric vectors

We need to pass two vector parameters and it produces a scatter plot accordingly.

- R

```

# R program to illustrate

# polymorphosim

# X Window System Graphics (X11)

X11(width = 5, height = 3)

# The runif() function generates random

```

```
# deviates of the uniform distribution

x <- runif(20)

y <- runif(20) * x

par(mar = c(2, 2, 0.3, 0.3))

# type = 'p' means as points, the output comes as scattered

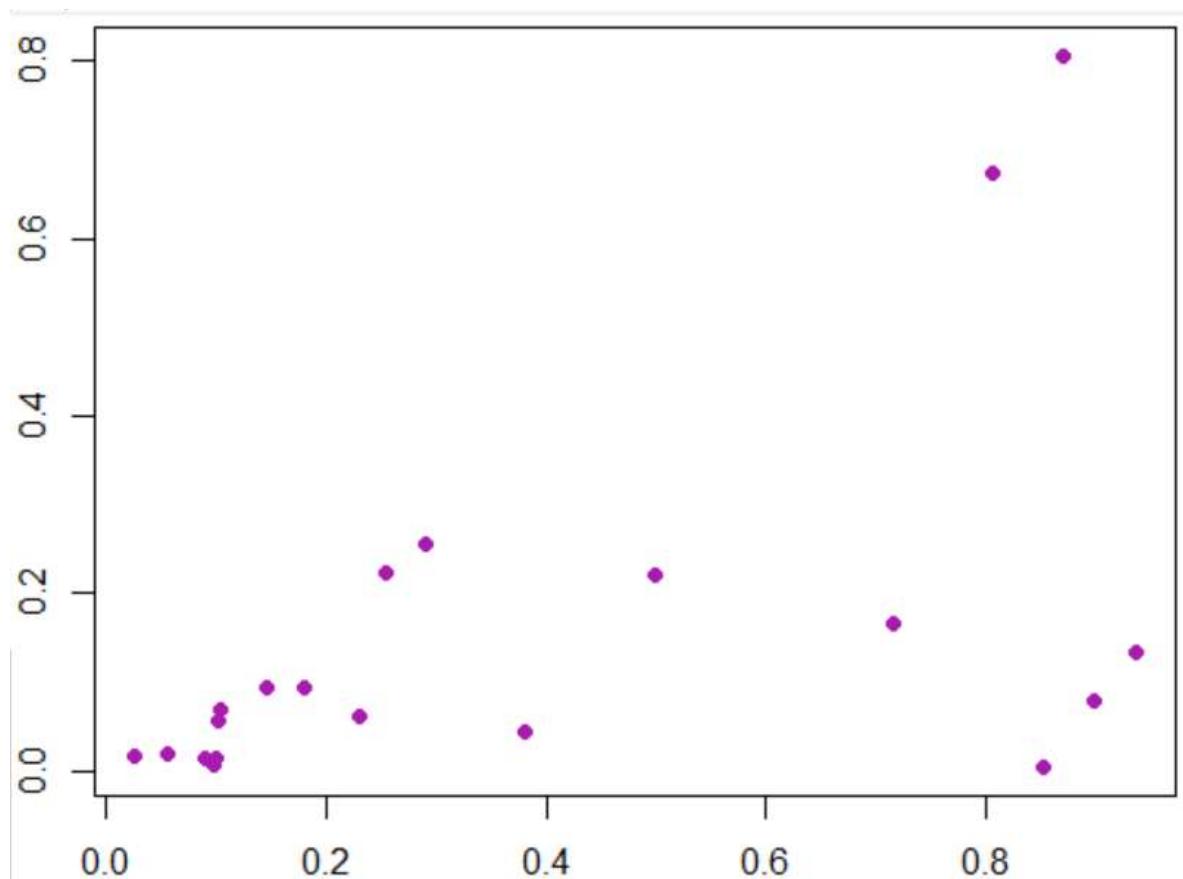
# pch stands for plot character. pch = 16 we get . character

plot(x, y, type = 'p', pch = 16, col = '#ab1ab0')

#Either mouse press or enter key press wait

z <- locator(1)
```

Output:



Input is a factor

If we passed factor as arguments then we get a bar chart pattern.

- R

```
# R program to illustrate  
# polymorphosim  
  
# X Window System Graphics (X11)  
X11(width = 5, height = 8)  
  
# here fruits names are passed and barchart is produced as output  
f<- factor(c('apple', 'orange', 'apple', 'pear', 'orange',
```

```
'apple', 'apple', 'orange'))
```

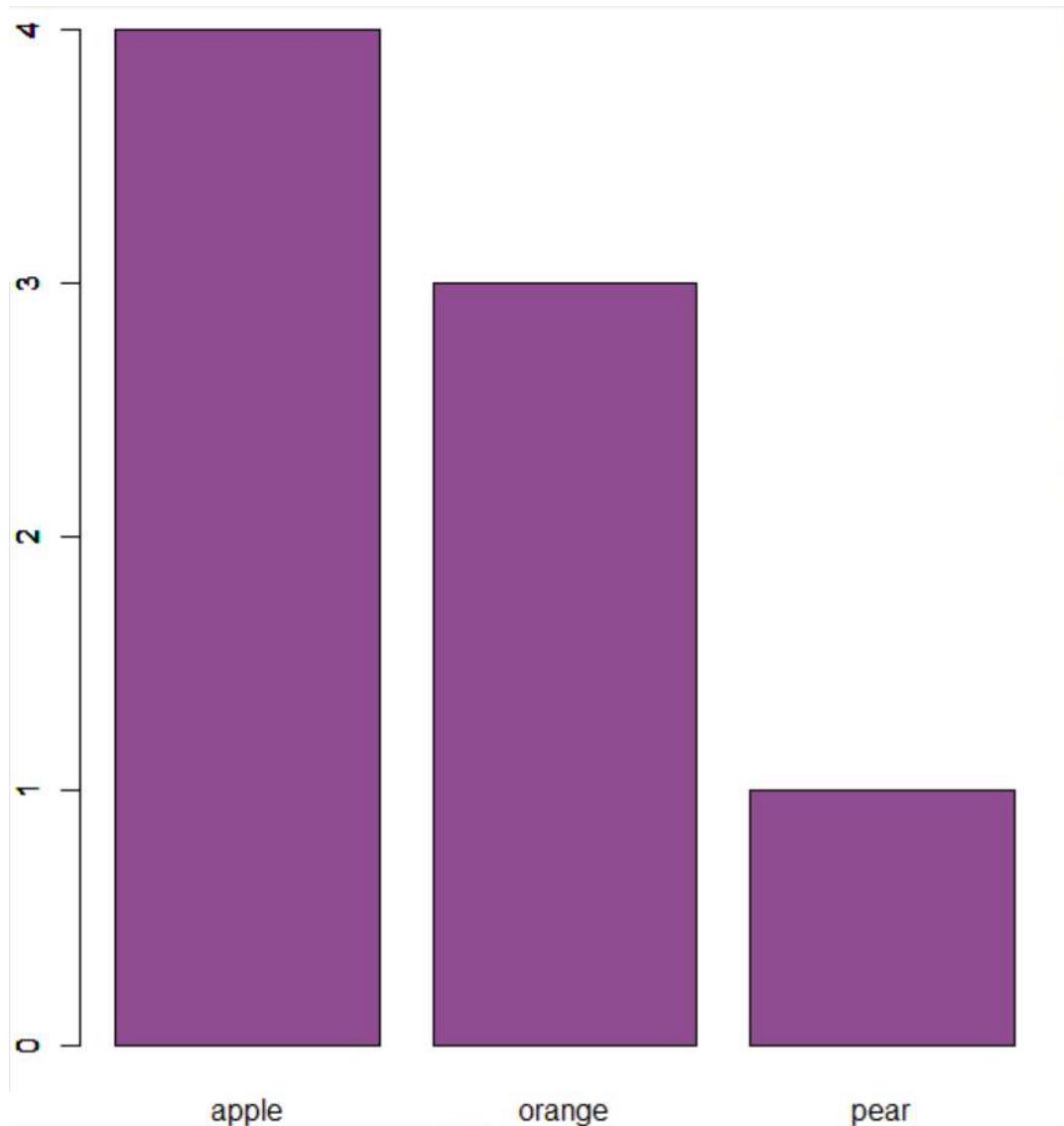
```
par(mar = c(2, 2, 0.6, 0.6))
```

```
# Using plot()
```

```
plot(f, col = '#8f4c91')
```

```
z <- locator(1)
```

Output:



Input is a data frame

The Plot function takes the data frame as an argument and each variable of the data frame is plotted against each other.

- R

```
# R program to illustrate

# polymorphosim

# X Window System Graphics (X11)

X11(width = 6, height = 6)

set.seed(280870)

x <- c(4, 3, 1, 2, 2, 4, 6, 4, 5, 5,
      4, 4, 5, 4, 4, 8, 4, 1, 2, 7)

y <- x * rnorm(20, 1, 0.3)

z <- x * y

# Taking a data frame

df <- data.frame(x, y, z)

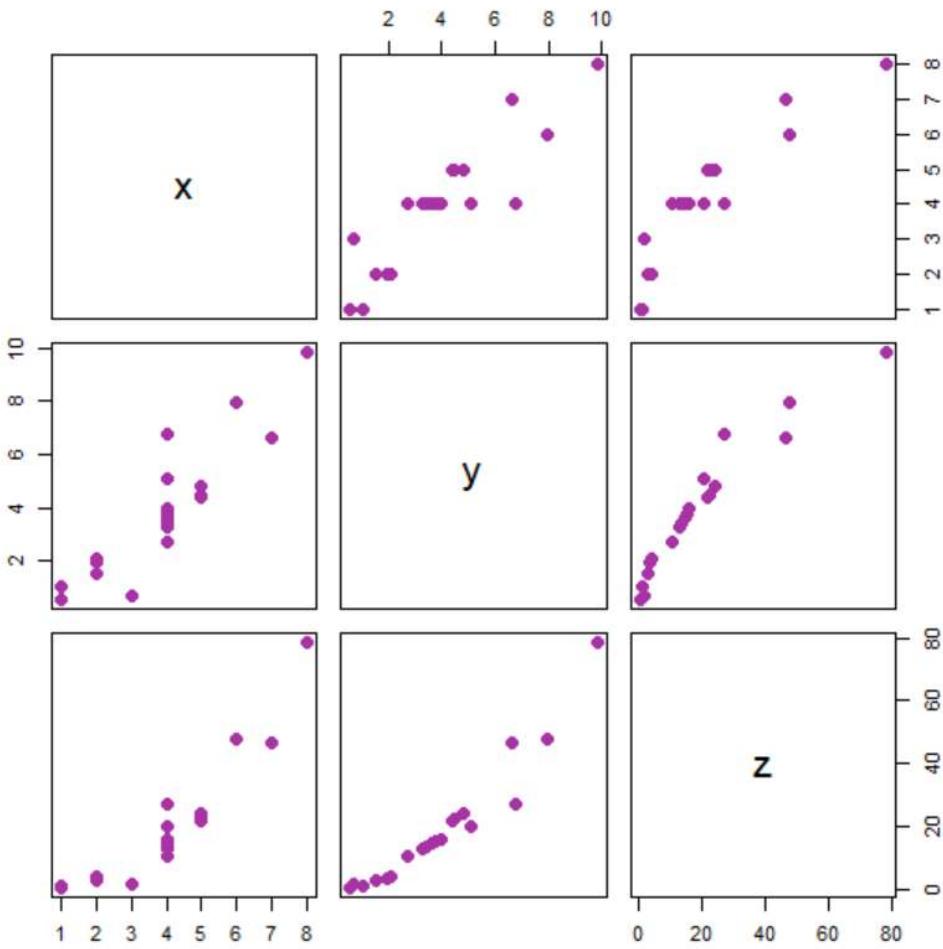
par(mar = c(0.1, 0.1, 0.1, 0.1))

# Using plot()

plot(df, col = '#a832a6', pch = 16, cex = 1.5)

z <- locator(1)
```

Output:



summary() in R

It is also a generic function that implements polymorphism in R. It is used to produce result summaries of the results of various model fitting functions.

Example 1:

- R

```
# R program to illustrate
# polymorphosim

# Rainbow colors and let us see summary of it
```

```
colors <- c("violet", "indigo", "blue", "green",
          "yellow", "orange", "red")

summary(colors)
```

Output:

```
Length Class Mode
7 character character
```

Example 2:

Let us check for summarized results for state.region. In R it usually displays what are the regions available under “Northeast”, “South”, “North Central”, and “West”. Using **summary()** function either one can pass state.region as 1st parameter and as a second step, (optionally) pass “maxsum” argument. “maxsum” indicates how many levels should be shown for factors in output.

- R

```
# R program to illustrate

# polymorphosim

state.region

# Provides summarised results under each region

summary(state.region)

# As maxsum is given as 3, totally we should have 3 regions

# But here we have 4 regions and hence highest count region,
# next highest count region is displayed and the other
# regions are clubbed under Other
```

```
summary(state.region, maxsum = 3)
```

Output:

```
> state.region
```

```
[1] South      West      West      South      West      West  
[7] Northeast   South     South     South     West      West  
[13] North Central North Central North Central North Central South     South  
[19] Northeast   South     Northeast   North Central North Central South  
[25] North Central West      North Central West      Northeast   Northeast  
[31] West       Northeast   South     North Central North Central South  
[37] West       Northeast   Northeast   South     North Central South  
[43] South      West      Northeast   South     West      South  
[49] North Central West
```

```
Levels: Northeast South North Central West
```

```
> summary(state.region)
```

```
Northeast   South North Central      West  
9          16      12      13
```

```
> summary(state.region, maxsum = 3)
```

```
South  West (Other)
```

```
16  13  21
```

Example 3:

If the data set is very large then let's have a look at how the **summary()** function works.

- R

```
# R program to illustrate  
  
# polymorphosim  
  
  
# 10 different data sets are taken using stats::rnorm  
  
x <- stats::rnorm(10)  
  
x  
  
  
# Let us cut the dataset to lie between -3 and 3 and  
  
# in this case, it will be  
  
# (-3,-2] (-2,-1] (-1,0] (0,1] (1,2] (2,3]  
  
c <- cut(x, breaks = -3:3)  
  
c  
  
  
# Summarized the available dataset under the given levels  
  
summary(c)
```

Output:

```
> x  
  
[1] 0.66647846 -0.29140286 -0.29596477 -0.23432541 -0.02144178 1.56640107 0.64575227  
[8] -0.23759734 0.73304657 -0.04201218  
  
  
> c  
  
[1] (0,1] (-1,0] (-1,0] (-1,0] (-1,0] (1,2] (0,1] (-1,0] (0,1] (-1,0]  
  
Levels: (-3,-2] (-2,-1] (-1,0] (0,1] (1,2] (2,3]  
  
  
> summary(c)
```

```
(-3,-2] (-2,-1] (-1,0] (0,1] (1,2] (2,3]
```

```
0 0 6 3 1 0
```

Till now, we have described the **plot()** and **summary()** function which is a polymorphic function. By means of different inputs, **plot()** function behavior is changing and producing outputs. Here we can see the polymorphism concept. Similarly, in **summary()**, function, by means of varying parameters, the same method is applicable to provide different statistical outputs. Now let's create our own generic methods.

Creating Generic Method

Let's create a class bank and let's try to create our own **display()** method that will use the **print()** method and will display the content of the class in the format specified by us.

For doing this we first have to create a generic **display()** function that will use the **UseMethod** function.

- R

```
display <- function(obj){  
  UseMethod("print")  
}
```

After creating the generic **display** function let's create the **display** function for our class bank.

- R

```
print.bank<-function(obj)  
{  
  cat("Name is ", obj$name, "\n")  
  cat(obj$account_no, " is the Acc no of the holder\n")  
  cat(obj$saving, " is the amount of saving in the account \n")  
  cat(obj$withdrawn, " is the withdrawn amount\n")  
}
```

Now, let's see the output given by this function

- R

```

x <- list(name ="Arjun", account_no = 1234,
          saving = 1500, withdrawn = 234)

class(x)<-"bank"

display <- function(obj){

  UseMethod("print")

}

print.bank<-function(obj)

{

  cat("Name is ", obj$name, "\n")

  cat(obj$account_no, " is the Acc no of the holder\n ")

  cat(obj$saving, " is the amount of saving in the account \n ")

  cat(obj$withdrawn, " is the withdrawn amount\n")

}

display(x)

```

Output:

```

Name is Arjun

1234 is the Acc no of the holder

1500 is the amount of saving in the account

234 is the withdrawn amount

```

R – Inheritance

 **Read**

 **Discuss**

[Courses](#)

[Practice](#)

-
-
-

Inheritance is one of the concept in [object oriented programming](#) by which new classes can be derived from existing or base classes helping in re-usability of code. Derived classes can be the same as a base class or can have extended features which creates a hierarchical structure of classes in the programming environment. In this article, we'll discuss how inheritance is followed out with three different types of classes in [R programming](#).

Inheritance in S3 Class

S3 class in R programming language has no formal and fixed definition. In an S3 object, a list with its class attribute is set to a class name. S3 class objects inherit only methods from its base class.

Example:

```
# Create a function to create an object of class

student <- function(n, a, r){

  value <- list(name = n, age = a, rno = r)

  attr(value, "class") <- student

  value

}
```

```
# Method for generic function print()

print.student <- function(obj){

  cat(obj$name, "\n")

  cat(obj$age, "\n")

  cat(obj$rno, "\n")
```

```

}

# Create an object which inherits class student

s <- list(name = "Utkarsh", age = 21, rno = 96,
          country = "India")

# Derive from class student

class(s) <- c("InternationalStudent", "student")

cat("The method print.student() is inherited:\n")

print(s)

# Overwriting the print method

print.InternationalStudent <- function(obj){

  cat(obj$name, "is from", obj$country, "\n")

}

cat("After overwriting method print.student():\n")

print(s)

# Check inheritance

cat("Does object 's' is inherited by class 'student' ?\n")

inherits(s, "student")

```

Output:

The method print.student() is inherited:

Utkarsh

21

96

After overwriting method print.student():

Utkarsh is from India

Does object 's' is inherited by class 'student' ?

[1] TRUE

Inheritance in S4 Class

S4 class in R programming have proper definition and derived classes will be able to inherit both attributes and methods from its base class.

Example:

```
# Define S4 class

setClass("student",

  slots = list(name = "character",

               age = "numeric", rno = "numeric"))

)
```

```
# Defining a function to display object details

setMethod("show", "student",

  function(obj){

    cat(obj@name, "\n")

    cat(obj@age, "\n")

    cat(obj@rno, "\n")

  }

)
```

```

# Inherit from student

setClass("InternationalStudent",
slots = list(country = "character"),
contains = "student"
)

# Rest of the attributes will be inherited from student

s <- new("InternationalStudent", name = "Utkarsh",
age = 21, rno = 96, country="India")

show(s)

```

Output:

Utkarsh

21

96

Inheritance in Reference Class

Inheritance in reference class is almost similar to the S4 class and uses setRefClass() function to perform inheritance.

Example:

```

# Define class

student <- setRefClass("student",
fields = list(name = "character",
age = "numeric", rno = "numeric"),
methods = list(

```

```
inc_age <- function(x) {  
  age <- age + x  
}  
  
dec_age <- function(x) {  
  age <- age - x  
}  
)  
)  
  
# Inheriting from Reference class  
  
InternStudent <- setRefClass("InternStudent",  
  fields = list(country = "character"),  
  contains = "student",  
  methods = list(  
    dec_age <- function(x) {  
      if((age - x) < 0) stop("Age cannot be negative")  
      age <- age - x  
    }  
  )  
)  
  
# Create object  
  
s <- InternStudent(name = "Utkarsh",  
  age = 21, rno = 96, country = "India")
```

```
cat("Decrease age by 5\n")
```

```
s$dec_age(5)
```

```
s$age
```

```
cat("Decrease age by 20\n")
```

```
s$dec_age(20)
```

```
s$age
```

Output:

```
[1] 16
```

```
Error in s$dec_age(20) : Age cannot be negative
```

```
[1] 16
```

Abstraction in R Programming

¶ Read

¶ Discuss

¶ Courses

¶ Practice

-
-
-

People who've been using the [R language](#) for any period of time have likely grown to be conversant in passing features as arguments to other functions. However, people are a whole lot much less probably to go back functions from their personal custom code. This is simply too horrific because doing so can open up a whole new international of abstraction that may greatly lower the quantity and complexity of the code vital to finish sure styles of duties. Here we offer a few short examples of ways R programmers can make use of lexical closures to encapsulate both records and strategies.



Implementation in R

To begin with, an easy instance, assume you want a function that provides `add_2()` to its argument. You could probably write something like this:

- R

```
add_2 <- function(y) { 2 + y }
```

Which does precisely what you'll anticipate:

```
> add_2(1:10)  
[1] 3 4 5 6 7 8 9 10 11 12
```

Now suppose you need every other feature that rather provides 7 to its argument. The heretical issue to do could be to write down any other characteristic, much like `add_2`, where the 2 is replaced with a 7. But this would be grossly inefficient: if within the future you discover that you made a mistake and also you in truth want to multiply the values instead of adding them, you will be pressured to trade the code in places. In this trivial instance, that won't be plenty of hassle, but for greater complicated projects, duplicating code is a recipe for catastrophe. A higher concept could be to put in writing a characteristic that takes one argument, `x`, that returns every other function which provides its argument, `y`, to `x`. In different words, something like this:

- R

```
add_x <- function(x) {  
  function(y) { x + y }
```

```
}
```

Now, while you name **add_x** with an argument, you may get back a feature that does precisely what you need:

- R

```
add_2 <- add_x(2)
```

```
add_7 <- add_x(7)
```

```
> add_2(1:10)
```

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

```
> add_7(1:10)
```

```
[1] 8 9 10 11 12 13 14 15 16 17
```

So this doesn't seem too earth-shattering. But if you look closely at the definition of **add_x**, you may notice something odd: how does the return characteristic realize in which to discover x when it's referred to as at a later point?

It turns out that R is lexically scoped, which means that features deliver with them a connection with the environment within which they were described. In this case, when you call **add_x**, the x argument you offer receives attached to the environment for the return characteristic. In different phrases, on this simple instance, you may think about R as simply changing all instances of the x variable within the feature to be lower back with the value you specify whilst you known as **add_x**. Ok, so this may be a neat trick, however, how this can be used extra productively? For a slightly extra complicated instance, think you're doing some complex bootstrapping, and, for efficiency, you pre-allocate container vectors to keep the results. This is easy if you have just a single vector of effects—all you need to do is take into account to iterate an index counter whenever you upload an end result to the vector.

- R

```
for (i in 1:nboot) {  
  bootmeans[i] <- mean(sample(data, length(data),  
    replace = TRUE))  
}
```

```
> mean(data)  
[1] 0.0196  
  
> mean(bootmeans)  
[1] 0.0188
```

But think you need to track several extraordinary statistics, every requiring you to maintain track of a unique index variable. If your bootstrapping ordinary is even a little bit complicated, this could be tedious and vulnerable to blunders. By the use of closures, you may summary away all of this bookkeeping. Here is a constructor function that wraps a pre-allocated container vector:

- R

```
make_container <- function(n) {  
  
  x <- numeric(n)  
  
  i <- 1  
  
  
  function(value = NULL) {  
  
    if (is.null(value)) {  
  
      return(x)  
  
    }  
  
    else {  
  
      x[i] <- value  
  
      i <- i + 1  
  
    }  
  
  }  
  
}
```

When you call **make_container** with an issue, it pre-allocates a numeric vector of the specified period, n, and returns a feature that permits you to feature statistics to that vector while not having to fear approximately keeping the music of an index. If you don't the argument to that return feature is NULL, the entire vector is the lower back.

- R

```
bootmeans <- make_container(nboot)

for (i in 1:nboot)
  bootmeans(mean(sample(data, length(data),
    replace = TRUE)))

> mean(data)
[1] 0.0196

> mean(bootmeans())
[1] 0.0207
```

Here **make_container** is tremendously easy, but it may be as complicated as you need. For example, you could want to have the constructor function carry out some expensive calculations which you could instead no longer do on every occasion the character is known as. In reality, that is what I even have completed within the **boolean3** package deal to decrease the range of calculations performed at each new release of the optimization habitual.

Looping over Objects in R Programming

 **Read**

 **Discuss**

 **Courses**

 **Practice**

-
-
-

Prerequisite: [Data Structures in R Programming](#)

One of the biggest issues with the “for” loop is its memory consumption and its slowness in executing a repetitive task. And when it comes to dealing with large data set and iterating over it, for loop is not advised. [R](#) provides many alternatives to be applied to vectors for looping operations that are pretty

useful when working interactively on a command line. In this article, we deal with **apply()** function and its variants:

- **apply()**
- **lapply()**
- **sapply()**
- **tapply()**
- **mapply()**

Let us see what each of these functions does.

Looping Function	Operation
apply()	Applies a function over the margins of an array or matrix
lapply()	Apply a function over a list or a vector
sapply()	Same as lapply() but with simplified results
tapply()	Apply a function over a ragged array
mapply()	Multivariate version of lapply()

- **apply():** This function applies a given function over the margins of a given array.

apply(array, margins, function, ...)

array = list of elements

margins = dimension of the array along which the function needs to be applied

function = the operation which you want to perform

Example:

```
# R program to illustrate  
  
# apply() function  
  
  
# Creating a matrix  
  
A = matrix(1:9, 3, 3)  
  
print(A)  
  
  
# Applying apply() over row of matrix  
  
# Here margin 1 is for row  
  
r = apply(A, 1, sum)  
  
print(r)  
  
  
# Applying apply() over column of matrix  
  
# Here margin 2 is for column  
  
c = apply(A, 2, sum)  
  
print(c)
```

Output:

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

[1] 12 15 18

[1] 6 15 24

- **lapply()**: This function is used to apply a function over a list. It always returns a list of the same length as the input list.

lapply(list, function, ...)

list = Created list

function = the operation which you want to perform

Example:

```
# R program to illustrate

# lapply() function

# Creating a matrix
A = matrix(1:9, 3, 3)

# Creating another matrix
B = matrix(10:18, 3, 3)

# Creating a list
myList = list(A, B)

# applying lapply()
determinant = lapply(myList, det)
print(determinant)
```

Output:

[1]]

[1] 0

[[2]]

[1] 5.329071e-15

- **sapply()**: This function is used to simplify the result of lapply(), if possible. Unlike lapply(), the result is not always a list. The output varies in the following ways:-
 - If output is a list containing elements having length 1, then a vector is returned.
 - If output is a list where all the elements are vectors of same length(>1), then a matrix is returned.
 - If output contains elements which cannot be simplified or elements of different types, a list is returned.

sapply(list, function, ...)

list = Created list

function = the operation which you want to perform

Example:

```
# R program to illustrate
```

```
# sapply() function
```

```
# Creating a list
```

```
A = list(a = 1:5, b = 6:10)
```

```
# applying sapply()
```

```
means = sapply(A, mean)
```

```
print(means)
```

Output:

a b

3 8

A vector is returned since the output had a list with elements of length 1.

- **tapply()**: This function is used to apply a function over subset of vectors given by a combination of factors.

tapply(vector, factor, function, ...)

vector = Created vector

factor = Created factor

function = the operation which you want to perform

Example:

```
# R program to illustrate

# tapply() function

# Creating a factor
Id = c(1, 1, 1, 1, 2, 2, 2, 3, 3)

# Creating a vector
val = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

# applying tapply()
result = tapply(val, Id, sum)
print(result)
```

Output:

1 2 3

10 18 17

How does the above code work?

Id	val
1	1
1	2
1	3
1	4
2	5
2	6
2	7
3	8
3	9

sum = 10

sum = 18

sum= 17

- **mapply()**: It's a multivariate version of lapply(). This function can be applied over several list simultaneously.

mapply(function, list1, list2, ...)

function = the operation which you want to perform

list1, list2 = Created lists

Example:

```
# R program to illustrate
```

```
# mapply() function
```

```
# Creating a list
```

```
A = list(c(1, 2, 3, 4))
```

```
# Creating another list
```

```
B = list(c(2, 5, 1, 6))
```

```
# Applying mapply()  
  
result = mapply(sum, A, B)  
  
print(result)
```

Output:

[1] 24

S3 class in R Programming

¶ Read

¶ Discuss

¶ Courses

¶ Practice

-
-
-

All things in the [R language](#) are considered objects. Objects have attributes and the most common attribute related to an object is class. The command class is used to define a class of an object or learn about the classes of an object.

Class is a vector and this property allows two things:

- Objects are allowed to inherit from numerous classes
- Order of inheritance can be specified for complex classes

Example: Checking the class of an object

- Python3

```
# Creating a vector x consisting of type of genders
```

```
x<-c("female", "male", "male", "female")  
  
# Using the command <code>class()</code>  
  
# to check the class of the vector  
  
class(x)
```

Output:

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

Example: Appending the class of an object

- Python3

```
# Creating a vector x consisting of type of genders  
  
x<-c("female", "male", "male", "female")  
  
# Using the command <code>class()</code>  
  
# to append the class of the vector  
  
class(x)<-append(class(x), "Gender")  
  
class(x)
```

Output:

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

Managing Memory

While doing object-oriented programming the programmer can have doubts about which class to use- S3 OR S4?

When comparing both the classes, S4 has a more structured approach while S3 is considered a flexible

class.

Memory environments are responsible for flexibility in S3 classes.

An environment is like a local scope and has a variable set associated with it. These variables are accessible if 'ID' associated with the environment is known.

To know or set the values of a variable in an environment, commands like **assign** and **get** are used.

Example: Assigning and getting values of a variable within the environment

- Python3

```
# Creating a vector x consisting of type of genders
```

```
# Creating a vector for age
```

```
age<-c(12, 10, 09)
```

```
# The command environment() can be used
```

```
# to bring the pointer to current environment
```

```
e <- environment()
```

```
e
```

```
# Setting the value of the variable
```

```
assign("age", 3, e)
```

```
ls()
```

```
# Getting the values of the variable
```

```
get("age", e)
```

Output:

```
[1] "age" "e" "x"
```

```
[1] 3
```

Environments can be easily created. They can also be embedded in other environments.

Creating an S3 class

An S3 class is the most prevalent and used class in R programming. It is easy to implement this class and most of the predefined classes are of this type.

An S3 object is basically a list with its class attributes assigned some names. And the member variable of the object created is the components of the list.

For creating an S3 object there are two main steps:

- Create a list(say x) with the required components
- Then the class can be formed by command class(x) and a name should be assigned to this class

Examples: An S3 object of bank account details can be created easily.

- Python3

```
x <- list(name ="Arjun", account_no = 1234,  
          saving = 1500, withdrawn = 234)  
  
class(x)<- "bank"  
  
x
```

Output:

```
$name
```

```
[1] "Arjun"
```

```
$account_no
```

```
[1] 1234
```

```
$saving
```

```
[1] 1500
```

```
$withdrawn
```

```
[1] 234
```

```
attr(, "class")
```

```
[1] "bank"
```

Examples: An S3 object of a person's resume can be created easily.

- Python3

```
x <- list(name ="Arjun", percentage = 95,  
          school_name ="ST Xavier")  
class(x)<-"resume"  
x
```

Output:

```
$name
```

```
[1] "Arjun"
```

```
$percentage
```

```
[1] 95
```

```
$school_name
```

```
[1] "ST Xavier"
```

```
attr(, "class")
```

```
[1] "resume"
```

Other languages like- Python, Java, C++, etc have a proper definition for class and the objects have proper defined methods and attributes.

But in the R language in the S3 class system, it is flexible and you can even modify or convert them (object of the same class can be different).

The S3 system in R language consists of three main components

- Generic function
- method
- attributes

Generic Functions

R uses print() function very often. If you type the name of the class, its internals will be printed or you can use the command print(name of the class). But the same function print() is used to print dissimilar things like – vectors, data frames, matrices, etc.

How does the print() recognize this variety of inputs?

The function print() is a generic function and hence is a collection of methods. These methods can further be checked by typing the command methods(print).

- Python3

```
methods(print)
```

Output:

```
[1] print.AES*
```

```
[2] print.Arima*
```

```
[3] print.AsIs
```

```
[4] print.Bibtex*
```

```
[5] print.CRAN_package_reverse_dependencies_and_views*
```

In the above long list there are important methods like print.factor(). When we print a factor through function print(), the call would automatically dispatch to print.factor()

The class created as – bank, would search for a method named print.bank(), and since no such method exists print.default() is used.

Generic functions have a default method which is used when no match is available.

Creating your own method

Creating your own method is possible.

Now if the class – ‘bank’ searches for print.bank(), it will find this method and use it if we have already created it.

- Python3

```
x <- list(name ="Arjun", account_no = 1234,  
          saving = 1500, withdrawn = 234)  
  
class(x)<- "bank"  
  
print.bank<-function(obj)  
{  
  cat("Name is ", obj$name, "\n")  
  cat(obj$account_no, " is the Acc no of the holder\n")  
  cat(obj$saving, " is the amount of saving in the account \n")  
  cat(obj$withdrawn, " is the withdrawn amount\n")  
}  
  
x
```

Output:

Name is Arjun

1234 is the Acc no of the holder

1500 is the amount of saving in the account

234 is the withdrawn amount

In a general way creating methods can be now easily defined and understood.

- Firstly define a function(in a generic way) existing out of the class.
- Secondly, defining the function specifics to a given class.

Based on the class names of an argument to the function and the suffix written in the names of the associated functions, The R environments determine which function to use.

- Python3

```
# Defining a function

indian <- function(eatslunch = TRUE, myFavorite = "daal")

{

  me <- list(haslunch = eatslunch,
            favoritelunch = myFavorite)

  # Set the name for the class

  class(me) <- append(class(me), "indian")

  return(me)
}

# Reserving the name of the function and
# by using the command <code>UseMethod</code>
# R will search for the appropriate function.
```

```
setHaslunch <- function(e, newValue)

{
  print("Calling the base setHaslunch function")

  UseMethod("setHaslunch", e)

  print(" this is not executed")

}

setHaslunch.default <- function(e, newValue)

{
  print("This objects is unable to be handled.")

  return(e)

}

setHaslunch.indian <- function(e, newValue)

{
  print("R is in setHaslunch.indian and is setting the value")

  e$haslunch <- newValue

  return(e)

}

# objects calling functions

foodie <- indian()

foodie$haslunch

foodie <- setHaslunch(foodie, FALSE)
```

```
foodie$haslunch
```

Output:

```
[1] TRUE
```

```
[1] "Calling the base setHaslunch function"
```

```
[1] "R is in setHaslunch.indian and is setting the value"
```

```
[1] FALSE
```

Attributes

Attributes of an object do not affect the value of an object, but they are a piece of extra information which is used to handle the objects.

The function **attributes()** can be used to view the attributes of an object.

Examples: An S3 object is created and its attributes are displayed.

- Python3

```
# Defining a function  
x <- list(name ="Arjun", percentage = 95,  
          school_name ="ST Xavier")  
  
attributes(x)
```

Output:

```
$names
```

```
[1] "name"      "percentage" "school_name"
```

```
$class
```

```
[1] "resume"
```

Also, you can add attributes to an object by using attr.

- Python3

```
# Defining a function  
  
x <- list(name ="Arju", percentage = 95,  
          school_name ="ST Xavie")  
  
attr(x, "age")<-c(18)  
  
attributes(x)
```

Output:

```
$names
```

```
[1] "name"      "percentage" "school_name"
```

```
$age
```

```
[1] 18
```

The S3 has been named so as it originated in the third version of S language. S is a programming language that later modified into R and S plus.