R Language Fundamentals

Why Do We Need Analytics?

Banking :

Large amount of customer data is generated everyday in Banks. While dealing with millions of customers on regular basis, it becomes hard to track their mortgages.

Solution:

R builds a custom model that maintains the loans provided to every individual customer which helps us to decide the amount to be paid by the customer over time

Insurance:

Insurance extensively depends on forecasting. It is difficult to decide which policy to accept or reject.

Solution:

By using the continuous credit report as input, we can create a model in R that will not only assess risk appetite but also make a predictive forecast as well.

Healthcare:

Every year millions of people are admitted in hospital and billions are spent annually just in the admission process.

Solution:

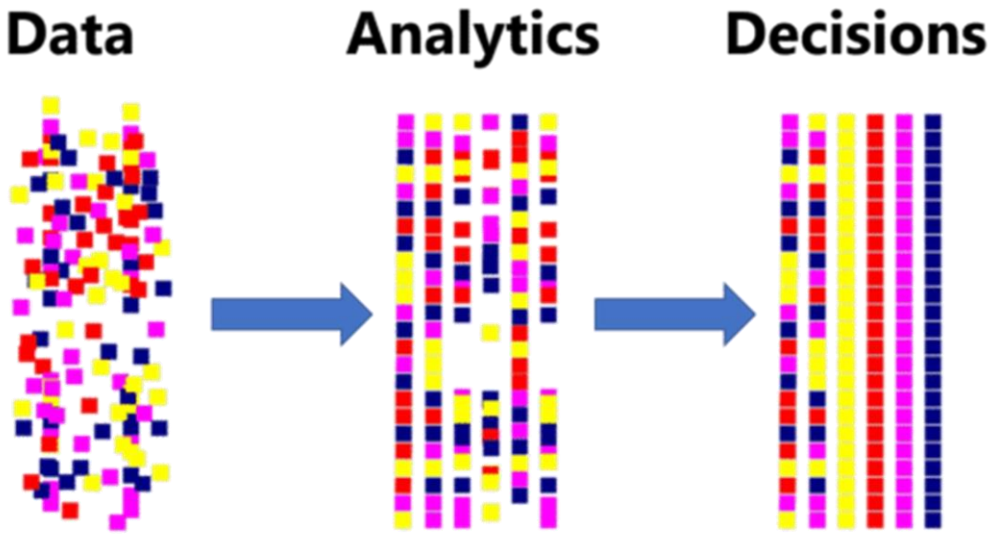
Given the patient history and medical history, a predictive model can be built to identify who is at risk for hospitalization and to what extent the medical equipment should be scaled.

Now we know how data analytics helps organizations to harness their data and use it to identify new opportunities. If we talk about the need for analytics in an organization, you must come across these 4 aspects:

What is Business Analytics?

Business analytics is a process of examining large sets of data and achieving hidden patterns, correlations and other insights. It basically helps you understand all the data that you have gathered, be it organisational data, market or product research data or any other kind of data. It becomes easy for you to make better decisions, better products, better marketing strategies etc.

Refer to the below image for better understanding:



If you look at the above figure, your data in the first image is scattered. Now, if you want something specific such as a particular record in a database, it becomes cumbersome. To simplify this, you need analysis. With analysis, it becomes easy to strike a correlation between the data. Once you have established what to do, it becomes quite easy for you to make decisions such as, which path you want to follow or in terms of business analytics, which path will lead to the betterment of your organization. But

you can’t expect people in the chain above to always understand the raw data that you are providing them after analytics. So to overcome this gap, we have a concept of data visualization.

Data visualization:

Data visualization is a visual access to huge amounts of data that you have generated after analytics. The human mind processes visual images and visual graphics are more better than compare to raw data. Its always easy for us to understand a pie chart or a bar graph compare to raw numbers. Now you may be wondering how can you achieve this data visualization from the data you have already analyzed?

There are various tools available in the market for Data Visualization:

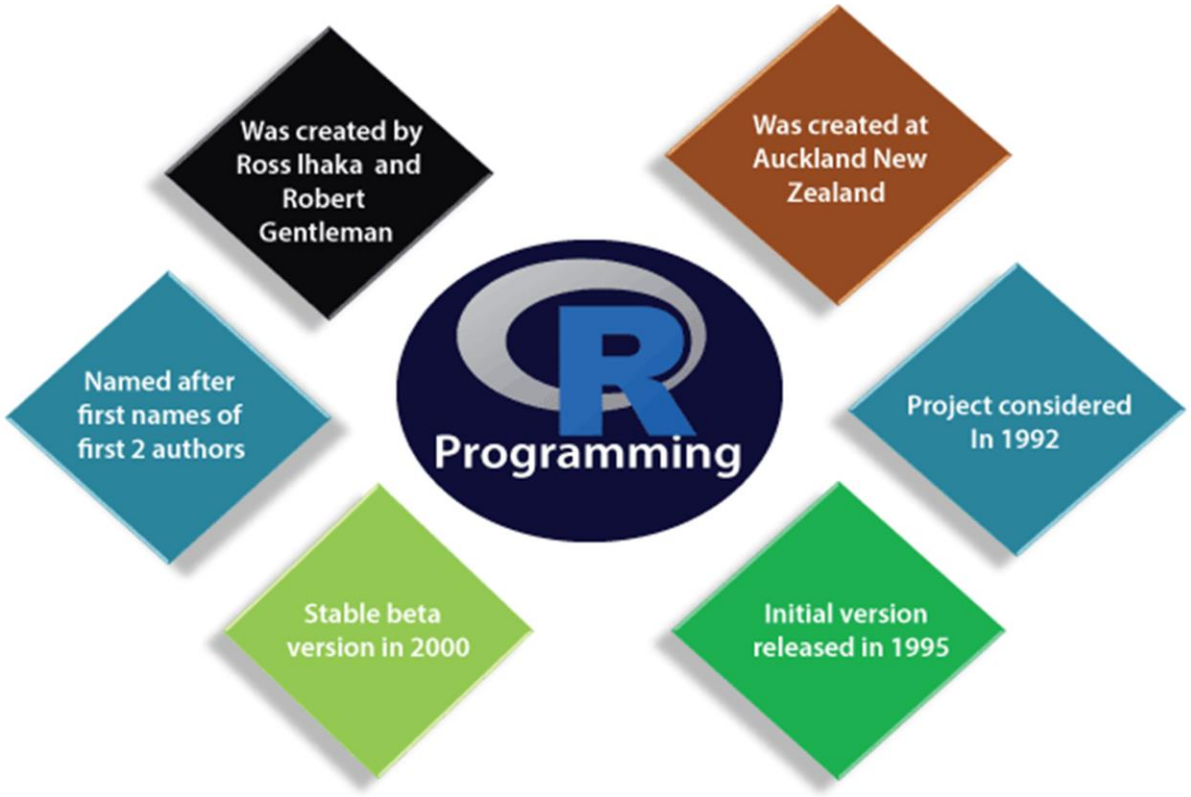
Overview Of R :

R is a open source programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.R is free software distributed under a GNU-style copy left, and an official part of the GNU project called GNU S.

R is an example of a FLOSS (Free Libre and Open Source Software) where one can freely distribute copies of this software, read it's source code, modify it, etc.

Evolution of R

- R was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.
- A large group of individuals has contributed to R by sending code and bug reports.
- Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.



What is R used for?

- Statistical inference
- Data analysis
- Machine learning algorithm

Data analysis with R is done in a series of steps; programming, transforming, discovering, modeling and communicate the results

Program: R is a clear and accessible programming tool

Transform: R is made up of a collection of libraries designed specifically for data science

Discover: Investigate the data, refine your hypothesis and analyze them

Model: R provides a wide array of tools to capture the right model for your data

Communicate: Integrate codes, graphs, and outputs to a report with R Markdown or build Shiny apps to share with the world

Features of R programming

R is a domain-specific programming language which aims to do data analysis. It has some unique features which make it very powerful. The most important arguably being the notation of vectors. These vectors allow us to perform a complex operation on a set of values in a single command. There are the following features of R programming:

- It is a simple and effective programming language which has been well developed.
- It is data analysis software.
- It is a well-designed, easy, and effective language which has the concepts of user-defined, looping, conditional, and various I/O facilities.
- It has a consistent and incorporated set of tools which are used for data analysis.
- For different types of calculation on arrays, lists and vectors, R contains a suite of operators.
- It provides effective data handling and storage facility.
- It is an open-source, powerful, and highly extensible software.
- It provides highly extensible graphical techniques.
- It allows us to perform multiple calculations using vectors.
- R is an interpreted language.
- R is a comprehensive programming language that provides support for procedural programming involving functions as well as object-oriented programming with generic functions.
- There are more than 10,000 packages in the repository of R programming. With these packages, one can make use of functions to facilitate easier programming.
- R has extensive community support that provides technical assistance, seminars and several boot camps to get you started with R.
- R is cross-platform compatible. R packages can be installed and used on any OS in any software environment without any changes.
- R facilitates complex operations with vectors, arrays, data frames as well as other data objects that have varying sizes.

Applications of R Programming in Real World

Data Science

Harvard Business Review named data scientist the “sexiest job of the 21st century”. Glassdoor named it the “best job of the year” for 2016. With the advent of IoT devices creating terabytes and terabytes of data that can be used to make better decisions, data science is a field that has no other way to go but up.Simply explained, a data scientist is a statistician with an extra asset: computer programming skills. Programming languages like R give a data scientist superpowers that allow them to collect data in realtime, perform statistical and predictive analysis, create visualizations and communicate actionable results to stakeholders.

Most courses on data science include R in their curriculum because it is the data scientist’s favourite tool.

Statistical computing

R is the most popular programming language among statisticians. In fact, it was initially built by statisticians for statisticians. It has a rich package repository with more than 9100 packages with every statistical function you can imagine.R’s expressive syntax allows researchers – even those from non computer science backgrounds to quickly import, clean and analyze data from various data sources.

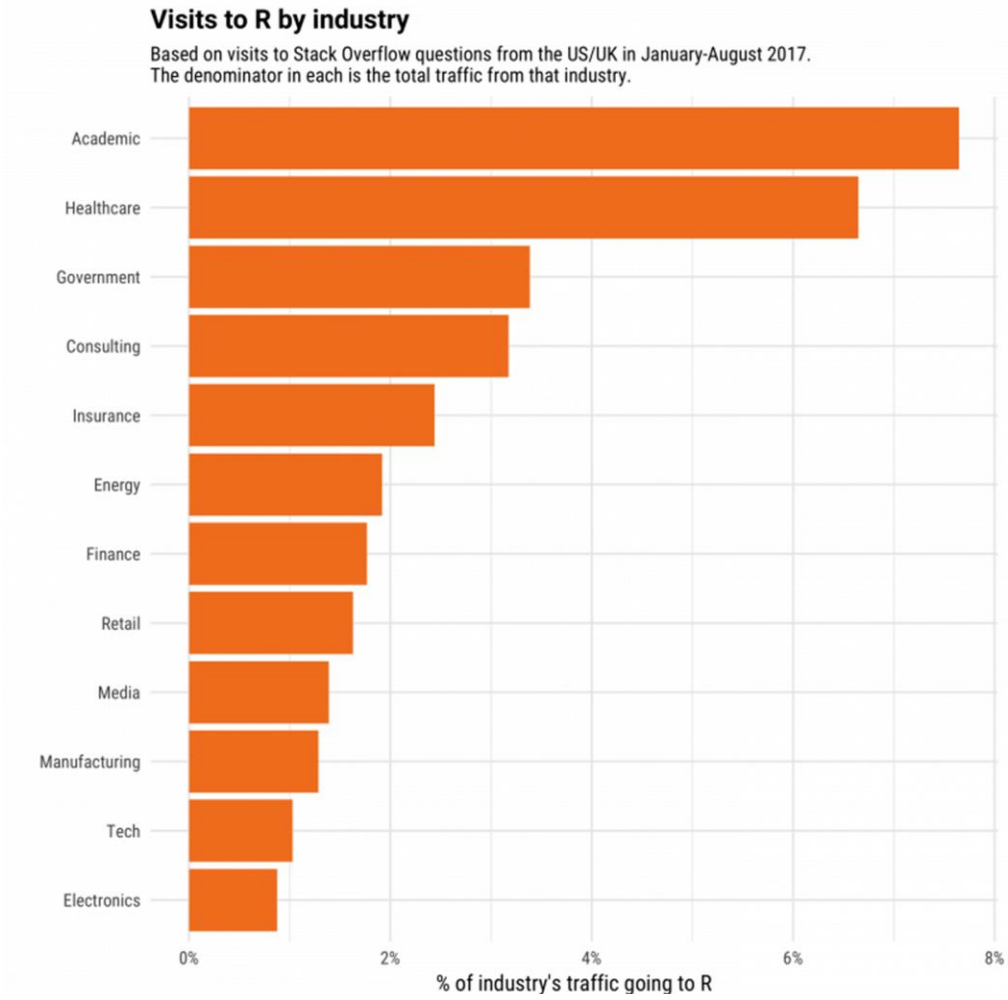
R also has charting capabilities, which means you can plot your data and create interesting visualizations from any dataset.

Machine Learning

R has found a lot of use in predictive analytics and machine learning. It has various package for common ML tasks like linear and non-linear regression, decision trees, linear and non-linear classification and many more.Everyone from machine learning enthusiasts to researchers use R to implement machine learning algorithms in fields like finance, genetics research, retail, marketing and health care.

R by Industry

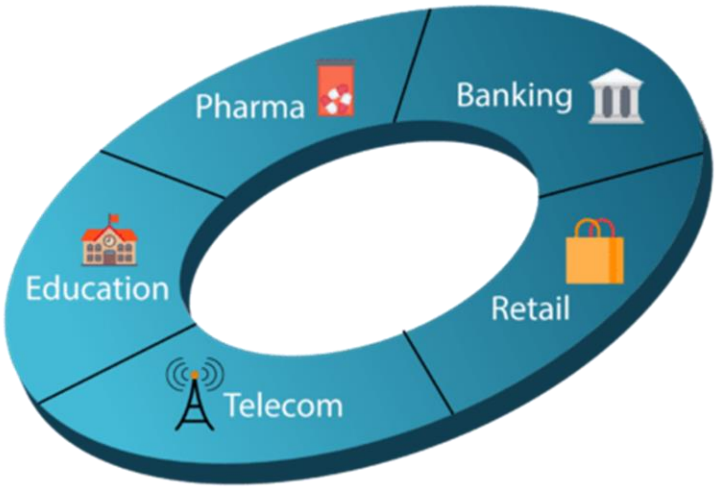
If we break down the use of R by industry, we see that academics come first. R is a language to do statistic. R is the first choice in the healthcare industry, followed by government and consulting.



Who uses R?

1. The Consumer Financial Protection Bureau uses R for data analysis
2. Statisticians at John Deere use R for time series modeling and geospatial analysis in a reliable and reproducible way.
3. Bank of America uses R for reporting.
4. R is part of technology stack behind Foursquare's famed recommendation engine.
5. ANZ, the fourth largest bank in Australia, using R for credit risk analysis.
6. Google uses R to predict Economic Activity.
7. Mozilla, the foundation responsible for the Firefox web browser, uses R to visualize Web activity.
8. Weather Service uses R to predict severe flooding.
9. Social networking companies are using R to monitor their user experience.
10. Newspapers companies are using R to create infographics and interactive data journalism applications.

Below are some of the domains where R is used:

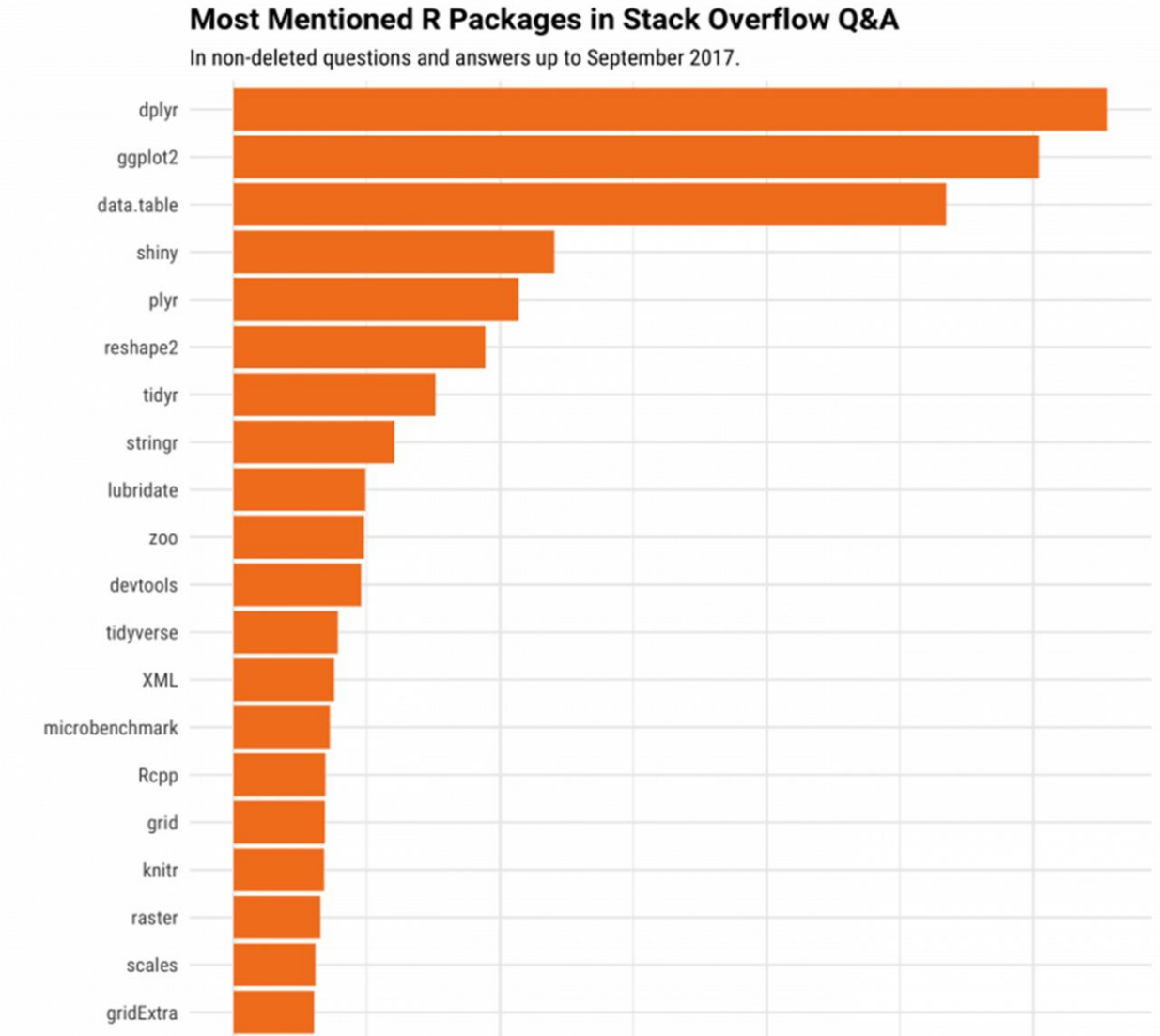


What is CRAN?

The “Comprehensive R Archive Network” (CRAN) is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.
The CRAN master site at WU (Wirtschaftsuniversität Wien) in Austria can be found at the URL https://CRAN.R-project.org/

R package

The primary uses of R is and will always be, statistic, visualization, and machine learning. The picture below shows which R package got the most questions in Stack Overflow. In the top 10, most of them are related to the workflow of a data scientist: data preparation and communicate the results.



All the libraries of R, almost 12k, are stored in CRAN. CRAN is a free and open source. You can download and use the numerous libraries to perform Machine Learning or time series analysis.

How R is better than Other Technologies

There are certain unique aspects of R programming which makes it better in comparison with other technologies:

- **Graphical Libraries** – R stays ahead of the curve through its aesthetic graphical libraries. Libraries like ggplot2, plotly facilitate appealing libraries for making well-defined plots.
- **Availability / Cost** – R is completely free to use which means widespread availability.
- **Advancement in Tool** – R supports various advanced tools and features that allow you to build robust statistical models.
- **Job Scenario** – As stated above, R is the primary tool for Data Science. With the immense growth in Data Science and rise in demand, R has become the most in-demand programming language of the world today.
- **Customer Service Support and Community** – With R, you can enjoy strong community support.
- **Portability** – R is highly portable. Many different programming languages and software frameworks can easily combine with the R environment for the best results.

Companies Using R

Some of the companies that are using R programming are as follows:

- Facebook
- Google
- Linkedin
- IBM
- Twitter
- Uber
- Airbnb
- Ford Motor company
- Microsoft

Following are the companies with their application areas:

Company	Application/Contribution
Twitter	Monitor user experience
Ford	Analyse social media to support design decisions for their cars
New York Times	Infographics, data journalism
Microsoft	Released Microsoft R Open, an enhanced R distribution and Microsoft R server after acquiring Revolution Analytics in 2015
Human Rights Data Analysis Group	Measure the impact of war
Google	Created the R style guide for the R user community inside Google

Applications of R Programming

- R is used in finance and banking sectors for detecting fraud, reducing customer churn rate and for making future decisions.
- R is also used by bioinformatics to analyse strands of genetic sequences, for performing drug discovery and also in computational neuroscience.
- R is used in social media analysis to discover potential customers in online advertising. Companies also use social media information to analyse customer sentiments for making their products better.
- E-Commerce companies make use of R to analyse the purchases made by the customers as well as their feedbacks.
- Manufacturing companies use R to analyze customer feedback. They also use it to predict future demand to adjust their manufacturing speeds and maximize profits.

There are several-applications available in real-time. Some of the popular applications are as follows:

- Facebook
- Google
- Twitter
- HRDAG
- Sunlight Foundation
- RealClimate
- NDAA
- XBOX ONE
- ANZ
- FDA

Why use R for statistical computing and graphics?

1. **R is open source and free**
2. **R is free to download-** as it is licensed under the terms of GNU General Public license.You can look at the source to see what's happening under the hood. There's more, most R packages are available under the same license so you can use them, even in commercial applications without having to call your lawyer.
3. **R is popular** – and increasing in popularity IEEE publishes a list of the most popular programming languages each year. R was ranked 5th in 2016, up from 6th in 2015. It is a big deal for a domain-specific language like R to be more popular than a general purpose language like C#.This not only shows the increasing interest in R as a programming language, but also of the fields like Data Science and Machine Learning where R is commonly used.
4. **R runs on all platforms -** You can find distributions of R for all popular platforms – Windows, Linux and Mac. R code that you write on one platform can easily be ported to another without any issues. Cross-platform interoperability is an important feature to have in today's computing

world – even Microsoft is making its coveted .NET platform available on all platforms after realizing the benefits of technology that runs on all systems.

5. **Learning R will increase your chances of getting a job -** According to the Data Science Salary Survey conducted by O'Reilly Media in 2014, data scientists are paid a median of $98,000 worldwide. The figure is higher in the US – around $144,000.Of course, knowing how to write R programs won't get you a job straight away, a data scientist has to juggle a lot of tools to do their work. Even if you are applying for a software developer position, R programming experience can make you stand out from the crowd.
6. **R is being used by the biggest tech giants -** Adoption by tech giants is always a sign of a programming language's potential. Today's companies don't make their decisions on a whim. Every major decision has to be backed by concrete analysis of data.

IDE's For R Language :

There are a variety of ide's available for r language:

- RStudio
- Spyder
- R Tools for Visual Studio
- Rattle
- StatET for R
- ESS
- Tinn-R
- R AnalyticalFlow
- Radiant
- RBox
- NVim-R
- r4intelliJ

R Scripts

R is the primary statistical programming language for performing modeling and graphical tasks. With its extensive support for performing matrix computation, R is used for a variety of tasks that involve complex datasets. There is the entropy of freedom for carrying out the selection of editing tools to perform an interaction with the native console. In order to perform scripting in R, you can simply import packages and then use the provided functions to achieve results with minimal lines of code.

There are several editors and IDEs that facilitate GUI features for executing R scripts. Some of the useful editors that are most commonly used in the R programming language are:

- RGui (R Graphical User Interface)
- Rstudio – It is a comprehensive environment for R scripting and has more features than Rstudio.

R Graphical User Interface (R GUI)

R GUI is the standard GUI platform for working in R. The R Console Window forms an essential part of the R GUI. In this window, we input various instructions, scripts and several other important operations. This console window has several tools embedded in it to facilitate ease of operations. This console appears whenever we access the R GUI.In the main panel of R GUI, go to the 'File' menu and select the 'New Script' option. This will create a new script in R. In order to quit the active R session, you can type the following code after the R prompt '>' as follows:

RStudio

RStudio is an integrated and comprehensive Integrated Development Environment for R. It facilitates extensive code editing, development as well as various features that make R an easy language to implement.

Features of RStudio

- RStudio provides various tools and features that allow you to boost your code productivity.
- It can also be accessed over the web and is cross-platform in nature.
- It facilitates automatic checking of updates so that you don't have to check for them manually.
- It provides support for recovery in case of file loss.
- With RStudio, you can manage the data more efficiently.

Components of RStudio

Source – In the top left corner of the screen is the text editor that allows you to work within source scripting. You can enter multiple lines in this source. Furthermore, users can save the R scripts to files that are stored in local memory.

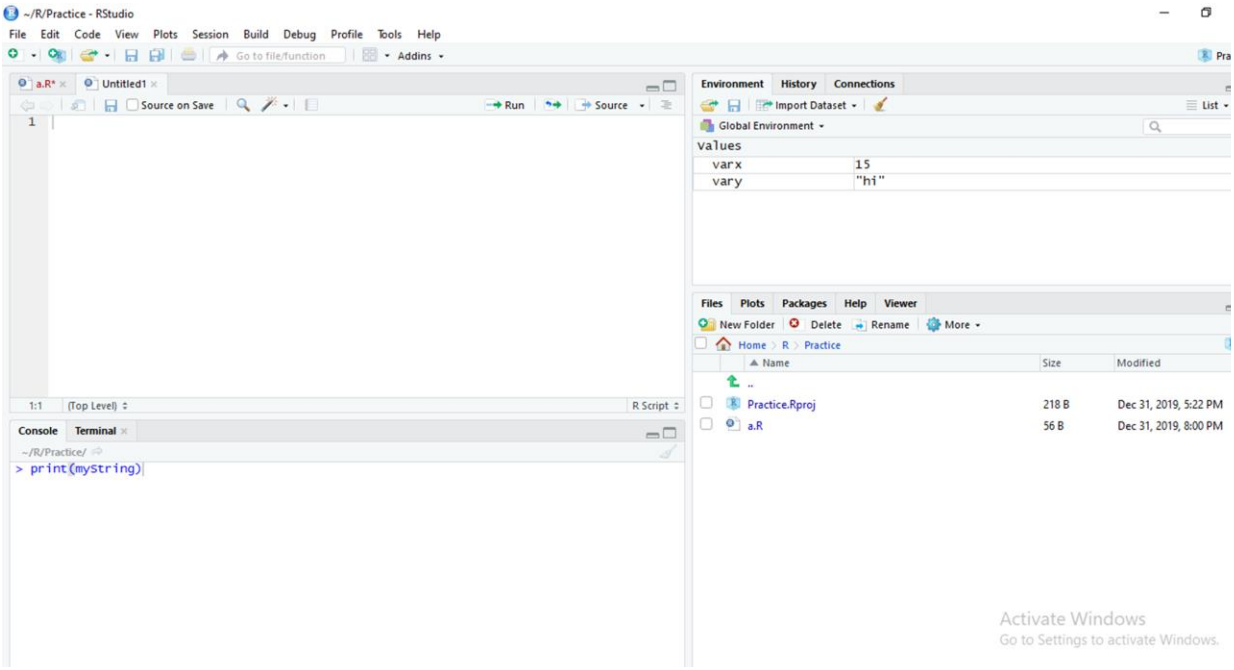
Console – This is present on the bottom left corner of the main window of R Studio. It facilitates interactive scripting in R.

Workspace and History – In the top right corner, you will find the R workspace and the history window. This will give you the list of all the variables that were created in the environment session. Furthermore, you can also view the list of past commands that were executed by R.

Files, Plots, Package, and Help at the bottom right corner gives access to the following tools:

- Files – A user can browse the various files and folders on a computer.
- Plots – We obtain the user plots here.

- Packages – Here, we can view the list of all the installed packages.
- Help – We can browse the built-in help system of R with this command.



Scripting in R

We will create a script to print “Hello world!” in R. To create scripts in R, you need to perform the following steps: Here in R, you will have to enclose some commands in print() to get the same output as on the command line. So you need to type below command: This takes “Hello World” as input in R.

```
1.  print("Hello World")
```

R Command Prompt

Once you have R environment setup, then it’s easy to start your R command prompt by just typing the following command at your command prompt −

$ R

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows −

> myString <- "Hello, World!"

> print (myString)

[1] "Hello, World!"

Sourcing a Script in R

While R console provides an interactive method to perform R programming, R Studio also provides various features to develop a script in the external editors and source the script into the console.
An advantage of writing into the R editor is that multiple lines can be written at once without prompting R to evaluate them individually. You can source the script in the following ways:

In the case of R Studio,

In order to execute a selected line of code:

Hold and press Ctrl+Enter

In order to execute the whole code:

Hold and press Ctrl+Shift+ Enter.

Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows −

My first program in R Programming

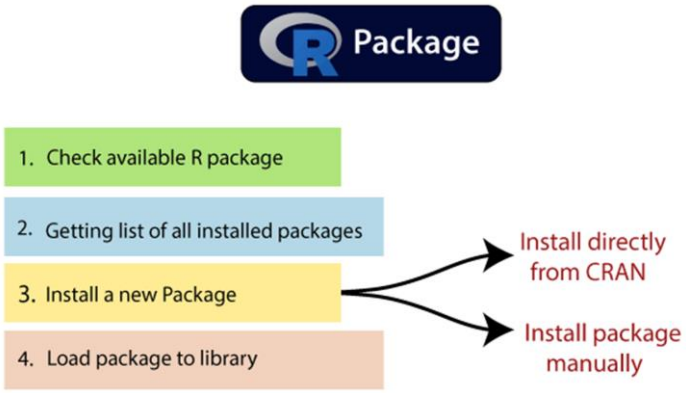
R does not support multi-line comment but you can perform a trick and its code will look something like this:

```
if(FALSE) {
"This is an example of how to write multi-line comments.
}
newStr <- "Hello - World!"
print ( newStr)
```

It is to be noted that the strings while using Multi-line comment should have to be put inside either Single quote or Double Quote.

R Packages

R packages are the collection of R functions, sample data, and compile codes. In the R environment, these packages are stored under a directory called "library." During installation, R installs a set of packages. We can add packages later when they are needed for some specific purpose. Only the default packages will be available when we start the R console. Other packages which are already installed will be loaded explicitly to be used by the R program. There is the following list of commands to be used to check, verify, and use the R packages.



Check Available R Packages

To check the available R Packages, we have to find the library location in which R packages are contained. R provides libPaths() function to find the library locations.

libPaths()

When the above code executes, it produces the following project, which may vary depending on the local settings of our PCs & Laptops.

[1] "C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6"

[2] "C:/Program Files/R/R-3.6.1/library"

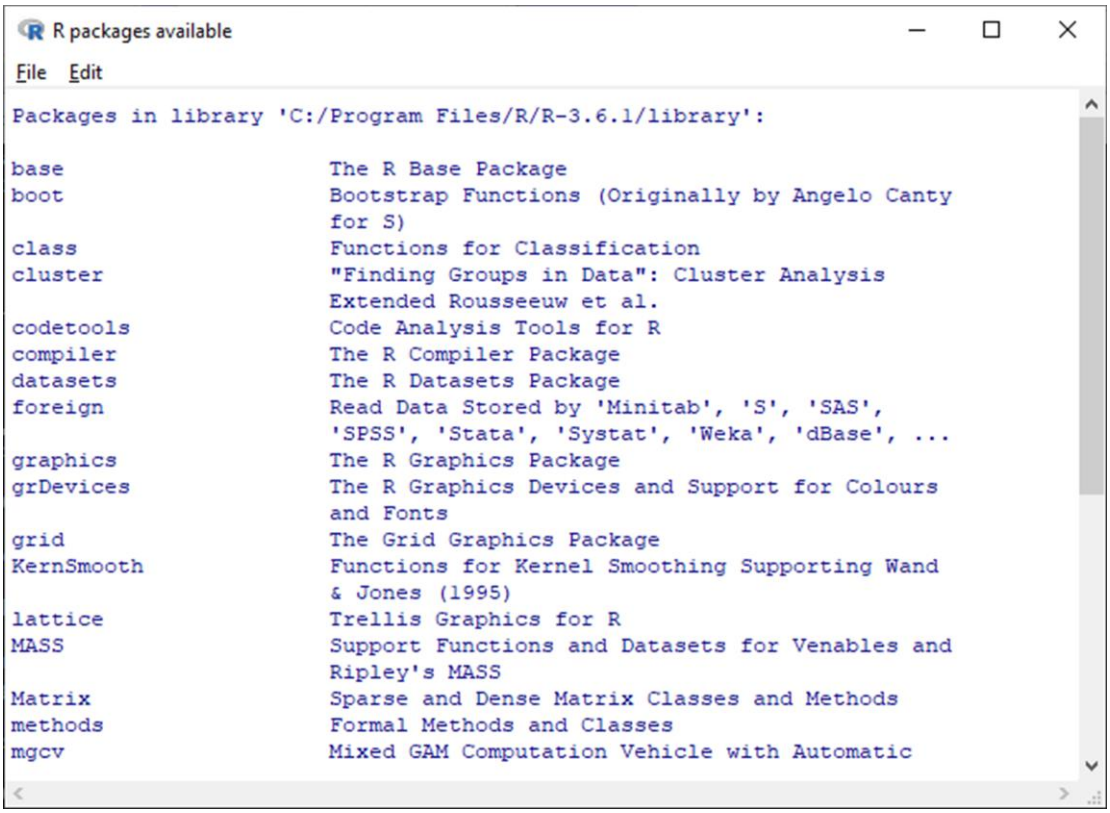
Getting the list of all the packages installed

R provides library() function, which allows us to get the list of all the installed packages.

library()

When we execute the above function, it produces the following result, which may vary depending on the local settings of our PCs or laptops.

Packages in library 'C:/Program Files/R/R-3.6.1/library':



Like library() function, R provides search() function to get all packages currently loaded in the R environment.

search()

When we execute the above code, it will produce the following result, which may vary depending on the local settings of our PCs and laptops:

```
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Install a New Package

In R, there are two techniques to add new R packages. The first technique is installing package directly from the CRAN directory, and the second one is to install it manually after downloading the package to our local system.

Install directly from CRAN

The following command is used to get the packages directly from CRAN webpage and install the package in the R environment. We may be prompted to choose the nearest mirror. Choose the one appropriate to our location.

install.packages("Package Name")

The syntax of installing XML package is as follows:

install.packages("XML")

Install package manually

To install a package manually, we first have to download it from https://cran.r-project.org/web/packages/available_packages_by_name.html. The required package will be saved as a .zip file in a suitable location in the local system.

Once the downloading has finished, we will use the following command:

install.packages(file_name_with_path, repos = NULL, type = "source")

Install the package named "XML"

install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")

Load Package to Library

We cannot use the package in our code until it will not be loaded into the current R environment. We also need to load a package which is already installed previously but not available in the current environment.

There is the following command to load a package:

library("package Name", lib.loc = "path to library")

Command to load the XML package

install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")

R Reserved Words

Reserved words in R programming are a set of words that have special meaning and cannot be used as an identifier (variable name, function name etc.).

Here is a list of reserved words in the R's parser.

Reserved words in R

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

This list can be viewed by typing help(reserved) or ?reserved at the R command prompt as follows.

> ?reserved

Among these words, if, else, repeat, while, function, for, in, next and break are used for conditions, loops and user defined functions.They form the basic building blocks of programming in R.

`TRUE` and `FALSE` are the logical constants in R.

`NULL` represents the absence of a value or an undefined value.

`Inf` is for "Infinity", for example when 1 is divided by 0 whereas `NaN` is for "Not a Number", for example when 0 is divided by 0.

`NA` stands for "Not Available" and is used to represent missing values.

R is a case sensitive language. Which mean that `TRUE` and `True` are not the same.

Identifier In R Language :

Identifier names are a combination of alphabets, digits, period (.) and also underscore (_). It is mandatory to start an identifier with a letter or a period. Another thing is if it starts with a period / dot operator then it you cannot write digit following it.

Rules For Writing Identifiers In R

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_).
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.
3. Reserved words in R cannot be used as identifiers.

Valid identifiers in R

`total`, `Sum`, `.fine.with.dot`, `this_is_acceptable`, `Number5`

Invalid identifiers in R

`tot@l`, `5um`, `_fine`, `TRUE`, `.0ne`

R Data Types

You need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory. You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are –

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

Basics types

- 4.5 is a decimal value called numerics.
- 4 is a natural value called integers. Integers are also numerics.
- TRUE or FALSE is a Boolean value called logical.
- The value inside " " or ' ' are text (string). They are called characters.

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

Example 1 :Integer Data Type

```
# Declare variables of different types

# Numeric

x <- 28

class(x)
```

Example 2: String Data Type

```
# String

y <- "R is Fantastic"

class(y)
```

Example 3: Boolean Data Type

```
# Boolean

z <- TRUE

class(z)
```

Another way of using interger data type

If you want to create any integer variable in R, you have to invoke the as.integer() function to define any integer type data. You can be certain that y is definitely an integer by applying the is.integer() function.

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

> s          # print the value of s
```

Example 4: Complex Data Type

```
v <- 2+5i

print(class(v))

it produces the following result –

[1] "complex"
```

Example 5: Character Data Type

```
v <- "TRUE"

print(class(v))

it produces the following result –

[1] "character"
```

Variables

Variables store values and are an important component in programming, especially for a data scientist. A variable can store a number, an object, a statistical result, vector, dataset, a model prediction basically anything R outputs. We can use that variable later simply by calling the name of the variable. Unique name has to be given to variable (also for functions and objects) is identifier.

Assignment of variable

In R programming, there are three operators which we can use to assign the values to the variable. We can use leftward, rightward, and equal_to operator for this purpose.

There are two functions which are used to print the value of the variable i.e., print() and cat(). The cat() function combines multiples values into a continuous print output.

```
# Assignment using equal operator.
variable.1 = 124

# Assignment using leftward operator.
variable.2 <- "Learn R Programming"

# Assignment using rightward operator.
133L -> variable.3

print(variable.1)
cat ("variable.1 is ", variable.1 ,"\n")
cat ("variable.2 is ", variable.2 ,"\n")
cat ("variable.3 is ", variable.3 ,"\n")
```

Finding Variables

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

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

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

List the variables starting with the pattern "var".

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

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

[1] "my var" "my_new_var" "my_var" "var.1"

[5] "var.2" "var.3" "var.name" "var_name2."

[9] "var_x" "varname"

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

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

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

[1] ".cars" ".Random.seed" ".var_name" ".varname" ".varname2"

[6] "my var" "my_new_var" "my_var" "var.1" "var.2"

[11]"var.3" "var.name" "var_name2." "var_x"

Deleting Variables

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

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

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

[1] "var.3"

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

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

rm(list = ls())

print(ls())

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

character(0)

Constants In R

Constants are entities within a program whose value can't be changed. There are 2 basic types of constant. These are numeric constants and character constants.All numbers fall under this category. They can be of type integer, double or complex.It can be checked with the typeof() function. Numeric constants followed by L are regarded as integer and those followed by i are regarded as complex.

```
> typeof(5)
[1] "double"
> typeof(5L)
[1] "integer"
> typeof(5i)
[1] "complex"
```

Numeric constants preceded by 0x or 0X are interpreted as hexadecimal numbers.

```
> 0xff
[1] 255
> 0XF + 1
[1] 16
```

Character Constants

Character constants can be represented using either single quotes (') or double quotes (") as delimiters.

```
> 'ray'
[1] "ray"

> typeof("karlos")
[1] "character"
```

Built-in Constants

Some of the built-in constants defined in R along with their values is shown below.

```
> LETTERS

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
"R" "S"

[20] "T" "U" "V" "W" "X" "Y" "Z"

> letters

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
"r" "s"

[20] "t" "u" "v" "w" "x" "y" "z"

> pi
```

```
[1] 3.141593

> month.name

[1] "January"   "February"  "March"     "April"     "May"       "June"

[7] "July"      "August"    "September" "October"   "November"  "December"

> month.abb

[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
"Dec"
```

Operators In R

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

Types of Operators

We have the following types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R

Arithmetic Operators in R

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

Example 1:

```
> x <- 5
> y <- 16
> x+y
[1] 21
> x-y
[1] -11
> x*y
[1] 80
> y/x
[1] 3.2
> y%/%x
[1] 3
> y%%x
[1] 1
> y^x
[1] 1048576
```

Example 2: Arithmetic Operators On Vectors

Addition :

```
g <- c (4, 6.5, 6)
s <- c (8, 3, 5)
print (g + s)
```

Subtration

```
g <- c ( 2, 5.5, 6)
s <- c (8, 3, 4)
print (g - s)
```

R Relational Operators

Relational operators are used to compare between values. Here is a list of relational operators available in R.

Relational Operators in R

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Example 1:

```
> y <- 16
> x<y
[1] TRUE
> x>y
[1] FALSE
> x<=5
[1] TRUE
> y>=20
[1] FALSE
> y == 16
[1] TRUE
> x != 5
[1] FALSE
```

Example 2: Relational Operators On Vectors

Greater than :

```
g <- c (2,5.5,6,9)

s <- c (8,2.5,14,9)

print (g > s)
```

Less Than :

```
g <- c (2, 5.6, 6,9)

s <- c(8,2.5,14,9)

print (g < s)
```

R Logical Operators

The logical operators allow a program to make a decision on the basis of multiple conditions. In the program, each operand is considered as a condition which can be evaluated to a false or true value. The value of the conditions is used to determine the overall value of the op1 operator op2. Logical operators are applicable to those vectors whose type is logical, numeric, or complex. The logical operator compares each element of the first vector with the corresponding element of the second vector. Logical operators are used to carry out Boolean operations like AND, OR etc.

Logical Operators in R

Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Element Wise Logical And Operator

```
g <- c(3, 1, TRUE, 2+3i)

s <- c(4,1,FALSE, 2+3i)
```

```
print (g & s)
```

It unites each element of the 1st vector with the equivalent element of the 2nd vector and returns TRUE or FALSE.

Element Wise Logical Or Operator

```
g <- c(3,0, TRUE, 2+2i)

s <- c(4,0, FALSE, 2+3i)

print (g | s)
```

It unites each element of the 1st vector with the equivalent element of the 2nd vector.

Logical Not Operator

```
k <- c (3,0, TRUE, 2+2i)

print (!k)
```

Logical And Operator

```
g <- c(3,0,TRUE,2+2i)

s <- c(1,3,TRUE,2+3i)

print (g && s)
```

Logical And Operator

```
g <- c (0,0,TRUE,2+2i)

s <- c (0,3,TRUE,2+3i)

print (g||s)
```

Assignment Operators

There are three types of operators used for assigning values to vectors.

```
g1 <- c (2,1,TRUE, 2+3i)

g2 <<- c (2,1,TRUE, 2+3i)

g3 = c (2,1, TRUE, 2+3i)

print (g1)
```

```
print (g2)

print (g3)
```

User Input of Strings in R

When we are working with R in an interactive session, we can use readline() function to take input from the user (terminal).This function will return a single element character vector.

So, if we want numbers, we need to do appropriate conversions.

R Program to Take Input From User

```
my.name <- readline(prompt="Enter name: ")
my.age <- readline(prompt="Enter age: ")
# convert character into integer
my.age <- as.integer(my.age)
print(paste("Hi,", my.name, "next year you will be", my.age+1, "years old."))
```

Here, we see that with the prompt argument we can choose to display an appropriate message for the user.In the above example, we convert the input age, which is a character vector into integer using the function as.integer(). This is necessary for the purpose of doing further calculations.

Versions Of R Language

The following table shows the release date, version, and description of R language:

Version-Release	Date	Description
0.49	1997-04-23	First time R's source was released, and CRAN (Comprehensive R Archive Network) was started.
0.60	1997-12-05	R officially gets the GNU license.
0.65.1	1999-10-07	update.packages and install.packages both are included.
1.0	2000-02-29	The first production-ready version was released.
1.4	2001-12-19	First version for Mac OS is made available.
2.0	2004-10-04	The first version for Mac OS is made available.
2.1	2005-04-18	Add support for UTF-8encoding, internationalization, localization etc.
2.11	2010-04-22	Add support for Windows 64-bit systems.
2.13	2011-04-14	Added a function that rapidly converts code to byte code.
2.14	2011-10-31	Added some new packages.
2.15	2012-03-30	Improved serialization speed for long vectors.
3.0	2013-04-03	Support for larger numeric values on 64-bit systems.
3.4	2017-04-21	The just-in-time compilation (JIT) is enabled by default.
3.5	2018-04-23	Added new features such as compact internal representation of integer sequences, serialization format etc.

Factors of R Language

1. Comprehensive Language

R is a comprehensive programming language, meaning that it provides services for statistical modeling as well as for software development. R is the primary language for Data Science as well as for developing web applications through its robust package RShiny. R is also an object-oriented programming language which is an addition to its procedure programming feature.

2. Provides a Wide Array of Packages

R is most widely used because of its wide availability of libraries. R has CRAN, which is a repository holding more than 10,0000 packages. These packages appeal to every functionality and different fields that deal with data. Based on user requirements and preferences, these packages provide different features to their users.

3. Possesses a Number of Graphical Libraries

The most important feature of R that sets it apart from other programming languages of Data Science is its massive collection of graphical libraries like ggplot2, plotly, etc. that are capable of making aesthetic and quality visualizations.

4. Open-source

R is an open-source programming language. This means that it is free of cost and requires no license. Furthermore, you can contribute towards the development of R, customize its packages and add more features.

5. Cross-Platform Compatibility

R supports cross-platform compatibility. It can be run on any OS in any software environment. It can also be run on any hardware configuration without any extra workarounds.

6. Facilities for Various Industries

Almost every industry that makes use of data, utilizes the R language. While only the academic areas made use of R in the past, it is now being heavily used in industries that require to mine insights from the data.The health industry makes use of R for drug design and analyzing genomic strands. Manufacturing industries like Ford use it in their optimization procedures. Furthermore, Airbnb and social media companies like Twitter use it to analyze its users.

7. No Need for a Compiler

R language is interpreted instead of compiled. Therefore, it does not need a compiler to compile code into an executable program. The R code is interpreted one step at a time and directly converted into machine level calls. This makes running an R script much less time-consuming.

8. Performs Fast Calculations

Through R, you can perform a wide variety of complex operations on vectors, arrays, data frames and other data objects of varying sizes. Furthermore, all these operations operate at a lightning speed. It provides various suites of operators to perform these miscellaneous calculations.

9. Can Handle all Sorts of Data

R provides excellent data handling and storage facilities.

In conjunction with data platforms like Hadoop, R facilitates the handling of structured as well as unstructured data that imparts a comprehensive data capability. Furthermore, R provides various data modeling and data operation facilities that are often a result of active interaction with the storage facility. It also provides extensions for SQL and Big Data.

10. Integration with Other Technologies

R can be integrated with a number of different technologies, frameworks, software packages, and programming languages. It can be paired with Hadoop to use its distributed computing ability. It can also be integrated with programs in other programming languages like C, C++, Java, Python, and FORTRAN.

11. R has an Active Community

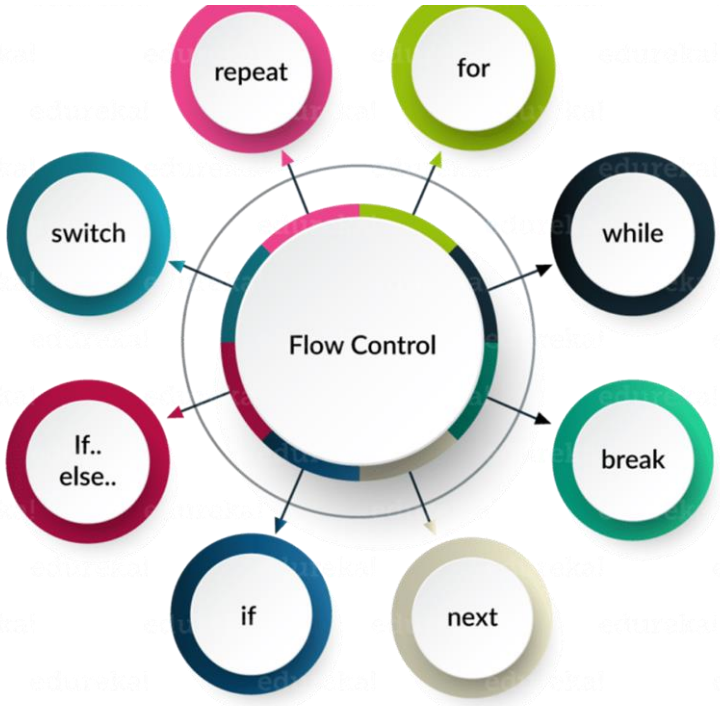
R is continuously evolving. The contribution is fuelled by the growing number of users who are using R on a daily basis. As mentioned above, R is an open-source library that is supported and maintained by a large user-base. Not only this, R has an engaging community that organizes seminars, boot camps and other training sessions of R. Once you start your journey in R, you will never feel alone.

12. Machine Learning with R

Earlier R had different packages for different machine learning algorithms. This may be considered inefficient and, therefore, the MLR package which stands for Machine Learning in R has become highly popular. This package is useful for all machine learning algorithms and provides other tools that help with machine learning as well.

Control Statements In R

Flow control statements play a very important role as they allow you to control the flow of execution of a script inside a function. The most commonly used flow control statements are represented in the below image:



R Decision Making

There are a lot of situations where you do not just want to execute one statement after another: in fact you have to control the flow of execution also. Usually this means that you merely want to execute some code if a condition is fulfilled. In that case control flow statements are implemented within R Program.

Decision making is an important aspect in any programming language. It helps to take a different path in the code based on the dynamic results. This helps a lot in process automation kind of stuff, where at some point in the process there could be multiple separate paths, and the state at that point decides the path to be taken. Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions. Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the

outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

R if Statement

The if statement consists of the Boolean expressions followed by one or more statements. The if statement is the simplest decision-making statement which helps us to take a decision on the basis of the condition.

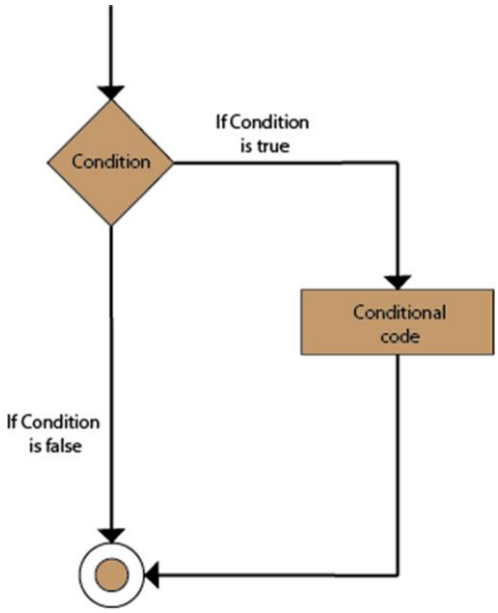
The if statement is a conditional programming statement which performs the function and displays the information if it is proved true.

The block of code inside the if statement will be executed only when the boolean expression evaluates to be true. If the statement evaluates false, then the code which is mentioned after the condition will run.

The syntax of if statement in R is as follows:

```
if(boolean_expression) {

  // If the boolean expression is true, then statement(s) will be executed.

}
```

Flow Chart of If Statement



Example 1

```
x <-24L
y <- "shubham"
if(is.integer(x))
{
   print("x is an Integer")
}
```

Example 2

```
x <-20
y<-24
count=0
if(x<y)
{
   cat(x,"is a smaller number\n")
   count=1
}
if(count==1){
   cat("Block is successfully execute")
}
```

Example 3

```
x <-24
if(x%%2==0){
   cat(x," is an even number")
}
if(x%%2!=0){
   cat(x," is an odd number")
}
```

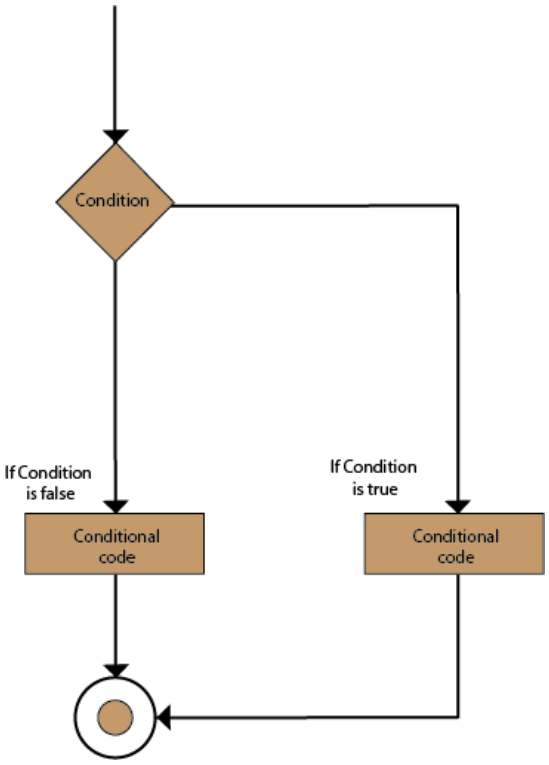
R if-else Statement

In the if statement, the inner code is executed when the condition is true. The code which is outside the if block will be executed when the if condition is false. There is another type of decision-making statement known as the if-else statement. An if-else statement is the if statement followed by an else statement. An if-else statement, else statement will be executed when the boolean expression will false. In simple words, If a Boolean expression will have true value, then the if block gets executed otherwise, the else block will get executed.R programming treats any non-zero and non-null values as true, and if the value is either zero or null, then it treats them as false.

The syntax of if –else statement in R is as follows:

```
if(boolean_expression) {

  // statement(s) will be executed if the boolean expression is true.

} else {

  // statement(s) will be executed if the boolean expression is false.

}
```

Flow Chart of If-else Statement



Example 1

```
# local variable definition
a<- 100
#checking boolean condition
if(a<20){
   # if the condition is true then print the following
   cat("a is less than 20\n")
}else{
   # if the condition is false then print the following
   cat("a is not less than 20\n")
}
cat("The value of a is", a)
```

Example 2

```
x <- c("Hardwork","is","the","key","of","success")


if("key" %in% x) {
  print("key is found")
} else {
  print("key is not found")
}
```

R else if statement

This statement is also known as nested if-else statement. The if statement is followed by an optional else if..... else statement. This statement is used to test various condition in a single if.....else if statement. There are some key points which are necessary to keep in mind when we are using the if.....else if.....else statement. These points are as follows:

if statement can have either zero or one else statement and it must come after any else if's statement.

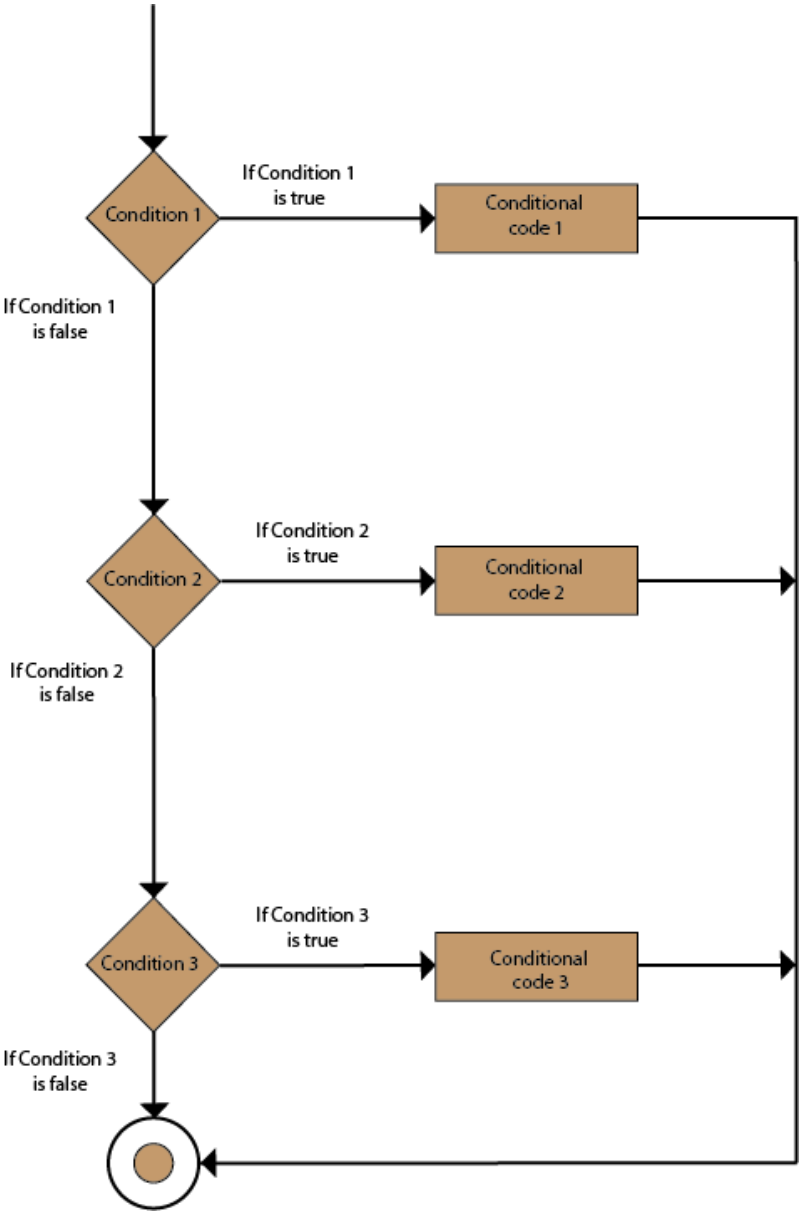
if statement can have many else if's statement and they come before the else statement.

Once an else if statement succeeds, none of the remaining else if's or else's will be tested.

The syntax of nested if –else statement in R is as follows:

```
if(boolean_expression 1) {
   // This block executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
   // This block executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
   // This block executes when the boolean expression 3 is true.
} else {
   // This block executes when none of the above condition is true.
}
```

Flow Chart of Nested If-else



Example 1

```
age <- readline(prompt="Enter age: ")
age <- as.integer(age)
if(age<18)
   print("You are child")
else if(age>30)
   print("You are old guy")
else
   print("You are adult")
```

Example 2

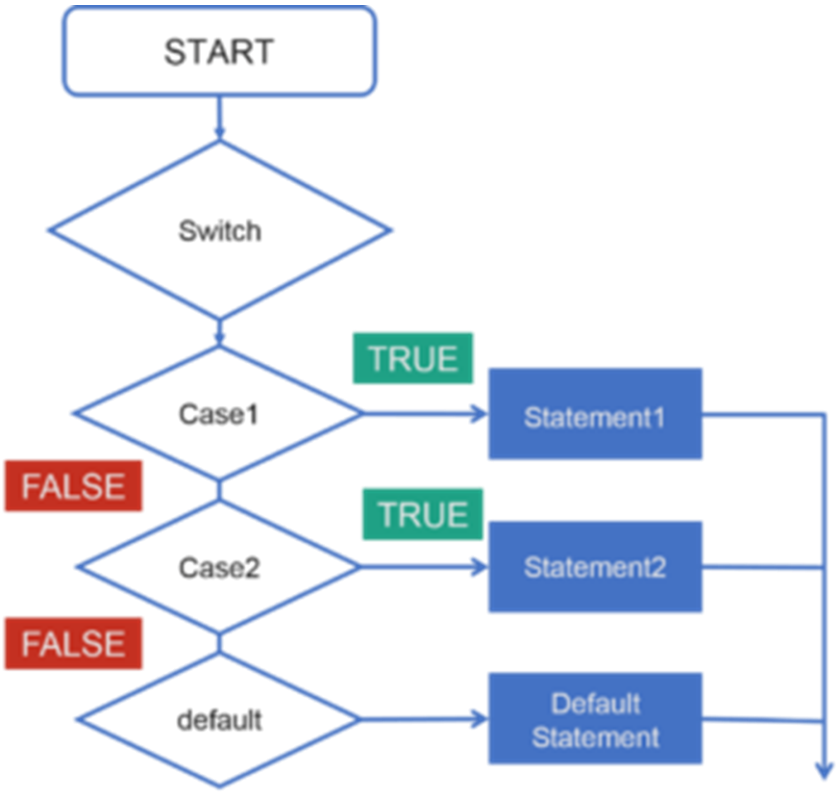
```
marks=83;
if(marks>75){
   print("First class")
}else if(marks>65){
   print("Second class")
}else if(marks>55){
   print("Third class")
}else{
   print("Fail")
}
```

Example 3

```
n1=4
n2=87
n3=43
n4=74
if(n1>n2){
   if(n1>n3&&n1>n4){
      largest=n1
   }
}else if(n2>n3){
   if(n2>n1&&n2>n4){
      largest=n2
   }
}else if(n3>n4){
   if(n3>n1&&n3>n2){
      largest=n3
   }
}else{
   largest=n4
}
cat("Largest number is =",largest)
```

R Switch Statement

A switch statement is a selection control mechanism that allows the value of an expression to change the control flow of program execution via map and search. The switch statement is used in place of long if statements which compare a variable with several integral values. It is a multi-way branch statement which provides an easy way to dispatch execution for different parts of code. This code is based on the value of the expression.



This statement allows a variable to be tested for equality against a list of values.

- If expression type is a character string, the string is matched to the listed cases.
- If there is more than one match, the first match element is used.
- No default case is available.
- If no case is matched, an unnamed case is used.

Example 1:

```
x <- switch(
3,
"Shubham",
"Nishka",
"Gunjan",
"Sumit"
)
```

```
print(x)
```

Example 2:

```
ax= 1
bx = 2
y = switch(
   ax+bx,
   "Hello, Shubham",
   "Hello Arpita",
   "Hello Vaishali",
   "Hello Nishka"
)
print (y)
```

Example 3:

```
y = "18"
x = switch(
   y,
   "9"="Hello Arpita",
   "12"="Hello Vaishali",
   "18"="Hello Nishka",
   "21"="Hello Shubham"
)
  print (x)
```

Example 4:

```
x= "2"
y="1"
a = switch(
   paste(x,y,sep=""),
   "9"="Hello Arpita",
   "12"="Hello Vaishali",
   "18"="Hello Nishka",
```

```
  "21"="Hello Shubham"
)
  print (a)
```

Example 5:

```
y = "18"
a=10
b=2
x = switch(
   y,
   "9"=cat("Addition=",a+b),
   "12"=cat("Subtraction =",a-b),
   "18"=cat("Division= ",a/b),
   "21"=cat("multiplication =",a*b)
)
  print (x)
```

Looping Statements

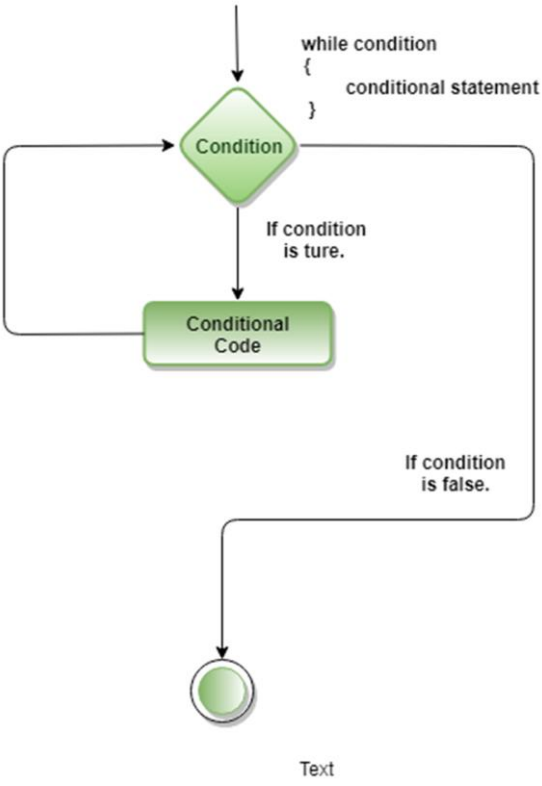
Loops help you to repeat certain set of actions so that you don't have to perform them repeatedly. Imagine you need to perform an operation 10 times, if you start writing the code for each time, the length of the program increases and it would be difficult for you to understand it later. But at the same time by using a loop, if I write the same statement inside a loop, it saves time and makes easier for code readability. It also gets more optimized with respect to code efficiency.

R while loop

A while loop is a type of control flow statements which is used to iterate a block of code several numbers of times. The while loop terminates when the value of the Boolean expression will be false.In while loop, firstly the condition will be checked and then after the body of the statement will execute. In this statement, the condition will be checked n+1 time, rather than n times.

The basic syntax of while loop is as follows:

```
while (test_expression) {
  statement
}
```



Text

Example 1:

```
v <- c("Hello","while loop","example")
c1 <- 2
while (c1 < 7) {
  print(v)
  c1= c1 + 1
}
}
```

```
n<-readline(prompt="please enter any integer value: ")
please enter any integer value: 12367906
n <- as.integer(n)
sum<-0
while(n!=0){
   sumsum=sum+(n%%10)
   n=as.integer(n/10)
}
cat("sum of the digits of the numbers is=",sum)
```

Example 3

```
n <- readline(prompt="Enter a four digit number please: ")
n <- as.integer(n)
num<-n
rev<-0
while(n!=0){
   rem<-n%%10
   rev<-rem+(rev*10)
   n<-as.integer(n/10)
}
print(rev)
if(rev==num){
   cat(num,"is a palindrome num")
}else{
   cat(num,"is not a palindrome number")
}
```

R For Loop

A for loop is the most popular control flow statement. A for loop is used to iterate a vector. It is similar to the while loop. There is only one difference between for and while, i.e., in while loop, the condition is checked before the execution of the body, but in for loop condition is checked after the execution of the body.

For loop in R Programming

In R, a for loop is a way to repeat a sequence of instructions under certain conditions. It allows us to automate parts of our code which need repetition. In simple words, a for loop is a repetition control structure. It allows us to efficiently write the loop that needs to execute a certain number of time.

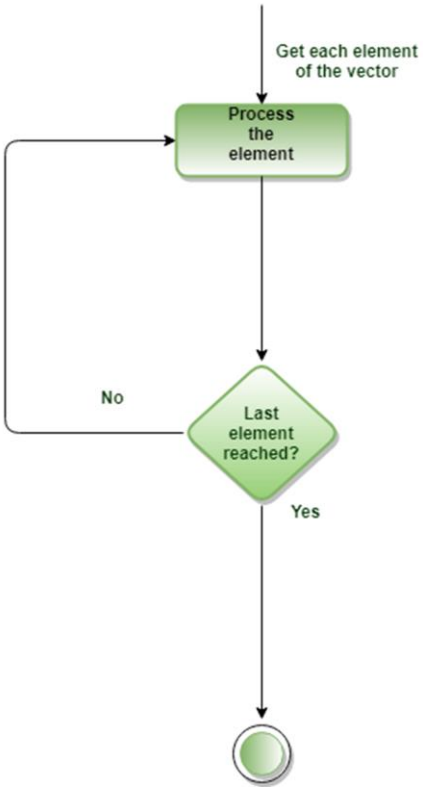
In R, a for loop is defined as :

- It starts with the keyword for like C or C++.
- Instead of initializing and declaring a loop counter variable, we declare a variable which is of the same type as the base type of the vector, matrix, etc., followed by a colon, which is then followed by the array or matrix name.
- In the loop body, use the loop variable rather than using the indexed array element.

There is a following syntax of for loop in R:

```
for (value in vector) {

  statements

}
```

Flow Chart



Example 1

```
fruit <- c('Apple', 'Orange',"Guava", 'Pinapple', 'Banana','Grapes')
# Create the for statement
for ( i in fruit)
{
   print(i)
}
```

Example 2:

```
v <- LETTERS[1:4]
for ( i in v)
{
  print(i)
}
```

Example 3:

```
x <- c(2,5,3,9,8,11,6,44,43,47,67,95,33,65,12,45,12)
count <- 0
for (val in x) {
if(val %% 2 == 0)  count = count+1
}
print(count)
```

R repeat loop

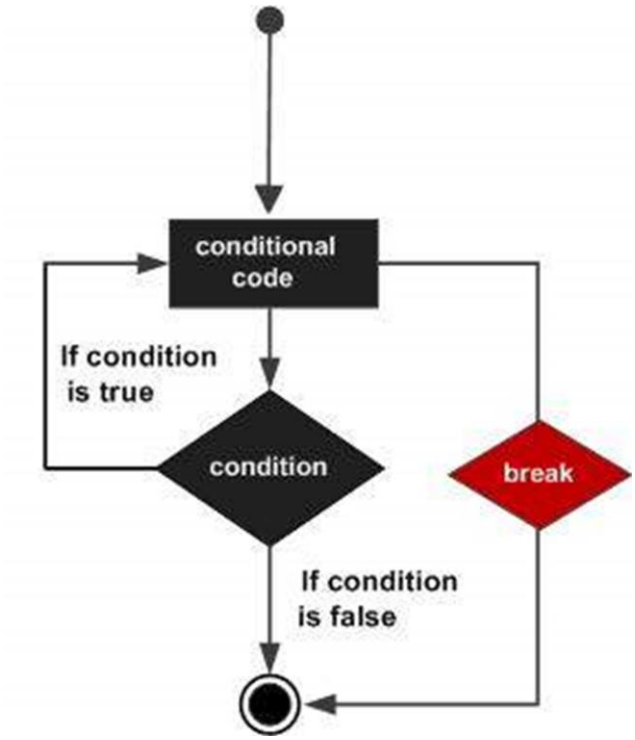
A repeat loop is used to iterate a block of code. It is a special type of loop in which there is no condition to exit from the loop. For exiting, we include a break statement with a user-defined condition. This

property of the loop makes it different from the other loops. A repeat loop constructs with the help of the repeat keyword in R. It is very easy to construct an infinite loop in R.

Syntax:

```
repeat {
  commands
  if(condition) {
    break
  }
}
```

Flow Chart



Example 1:

```
v <- c("Hello","python")

c2 <- 2

repeat {

  print(v)

  c2 <- c2+1

   if(c2 > 5) {

     break

  }

}
```

Example 2:

```
sum <- 0
{
   n1<-readline(prompt="Enter any integer value below 20: " )
   n1<-as.integer(n1)
}
repeat{
   sum<-sum+n1
   n1=n1+1
   if(n1>20){
      break
   }
}
cat("The sum of numbers from the repeat loop is: ",sum)
```

Example 3:

```
a <- 1
repeat {
 if(a == 10)
   break
 if(a == 7){
   a=a+1
   next
 }
 print(a)
 a <- a+1
}
```

Example 4:

```
terms<-readline(prompt="How many terms do you want ?")
terms<-as.integer(terms)
i<-1
repeat{
   print(paste("The cube of number",i,"is =",(i*i*i)))
   if(i==terms)
      break
   i<-i+1
}
```

R - Break Statement

In the R language, the break statement is used to break the execution and for an immediate exit from the loop. In nested loops, break exits from the innermost loop only and control transfer to the outer loop.

It is useful to manage and control the program execution flow. We can use it to various loops like: for, repeat, etc.

There are basically two usages of break statement which are as follows:

1. When the break statement is inside the loop, the loop terminates immediately and program control resumes on the next statement after the loop.
2. It is also used to terminate a case in the switch statement.

Syntax

There is the following syntax for creating a break statement in R

break

Example 1:

```
a <- 1
repeat {
  print("hello");
  if(a >= 5)
    break
  a<-a+1
}
```

Example 2:

```
a<-1
while (a < 10) {
  print(a)
  if(a==5)
    break
  a = a + 1
}
```

Example 3:

```
for (i in c(2,4,6,8)) {
  for (j in c(1,3)) {
     if (i==6)
       break
    print(i)
  }
}
```

R - Next Statement

The next statement is used to skip any remaining statements in the loop and continue executing. In simple words, a next statement is a statement which skips the current iteration of a loop without terminating it. When the next statement is encountered, the R parser skips further evaluation and starts the next iteration of the loop. This statement is mostly used with for loop and while loop.

Syntax

There is the following syntax for creating the next statement in R

Next

Example 1: Next in for loop with character abbr.

```
v <- LETTERS[1:6]
for ( i in v) {


  if (i == "D") {
     next
  }
  print(i)
}
```

Example 2:Next in repeat loop

```
a <- 1
repeat {
 if(a == 10)
  break
 if(a == 5){
  next
 }
 print(a)
 a <- a+1
}
```

Example 3:Next in While Loop

```
a<-1
while (a < 10) {
 if(a==5)
  next
print(a)
 a = a + 1
}
```

Example 4:Next in For Loop

```
x <- 1:10
for (val in x) {
if (val == 3){
next
}
print(val)
}
```

Example 5:Sum of Numbers

```
a1<- c(10L,-11L,12L,-13L,14L,-15L,16L,-17L,18L)
sum<-0
for(i in a1){
   if(i<0){
      next
   }
   sum=sum+i
}
cat("The sum of all positive numbers in array is=",sum)
```

Example 6:Generate the number series from 1 to 9

```
j<-0
while(j<10){
   if (j==7){
      j=j+1
      next
   }
   cat("\nnumber is =",j)
   j=j+1
}
```

R Functions

A set of statements which are organized together to perform a specific task is known as a function. R provides a series of in-built functions, and it allows the user to create their own functions. Functions are used to perform tasks in the modular approach.

Functions are used to avoid repeating the same task and to reduce complexity. To understand and maintain our code, we logically break it into smaller parts using the function. A function should be

- Written to carry out a specified task.
- May or may not have arguments
- Contain a body in which our code is written.
- May or may not return one or more output values.

"An R function is created by using the keyword function." There is the following syntax of R function:

Syntax:

function_name <- function (argu_1, argu_2, ... argu_N)

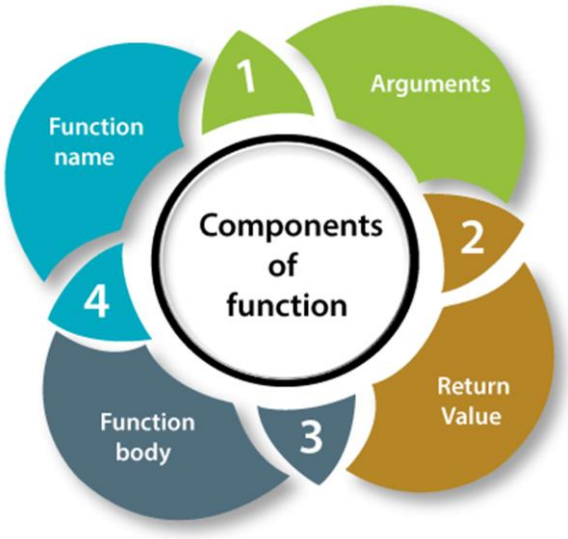
{

#Function body

}

Components of Functions:

There are four components of function, which are as follows:



Function Name

The function name is the actual name of the function. In R, the function is stored as an object with its name.

Arguments

In R, an argument is a placeholder. In function, arguments are optional means a function may or may not contain arguments, and these arguments can have default values also. We pass a value to the argument when a function is invoked.

Function Body

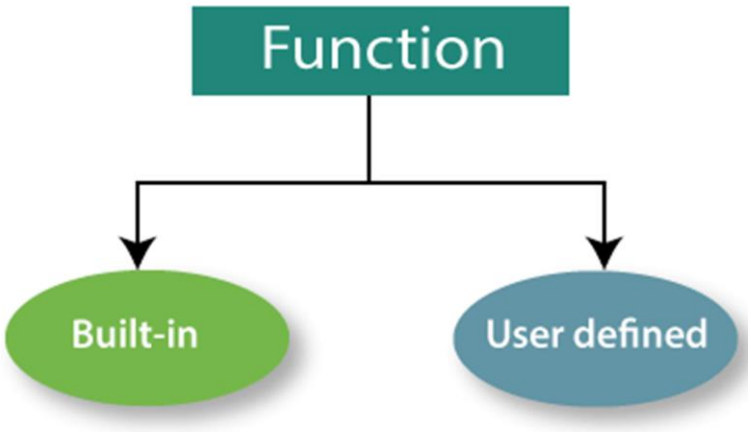
The function body contains a set of statements which defines what the function does.

Return value

It is the last expression in the function body which is to be evaluated.

Function Types:

Similar to the other languages, R also has two types of function, i.e. Built-in Function and User-defined Function. In R, there are lots of built-in functions which we can directly call in the program without defining them. R also allows us to create our own functions.



Built-in function

The functions which are already created or defined in the programming framework are known as built-in functions. User doesn't need to create these types of functions, and these functions are built into an application. End-users can access these functions by simply calling it. R have different types of built-in functions such as seq(), mean(), max(), and sum(x) etc.

```
# Creating sequence of numbers from 32 to 46.
print(seq(32,46))

# Finding the mean of numbers from 22 to 80.
print(mean(22:80))

# Finding the sum of numbers from 41 to 70.
print(sum(41:70))
```

User-defined function

R allows us to create our own function in our program. A user defines a user-define function to fulfill the requirement of user. Once these functions are created, we can use these functions like in-built function.

Example 1: Creating a function without an argument.

```
FUNCTION   DEFINITION
new.function <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
FUNCTION CALLING
 new.function()
```

Example 2: Function calling with an argument

We can easily call a function by passing an appropriate argument in the function. Let see an example to see how a function is called.

```
# Creating a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }

# Calling the function new.function supplying 10 as an argument.
new.function(10)
```

Example 3: Function calling with Argument Values(by position and by name)

```
# Creating a function with arguments.
new.function <- function(x,y,z) {
  result <- x * y + z
  print(result)
}
 # Calling the function by position of arguments.
new.function(11,13,9)
 # Calling the function by names of the arguments.
new.function(x = 2, y = 5, z = 3)
```

Example 4: Function calling with default arguments

```
# Creating a function with arguments.
new.function <- function(x = 11, y = 24) {
result <- x * y
print(result)
}
# Calling the function without giving any argument.
new.function()
# Calling the function with giving new values of the argument.
new.function(4,6)
```

Example 5: Generating the sum of the squares of 2 numbers:

```
sum_of_square <- function(x,y) {
x^2 + y^2
}
sum_of_sqares(3,4)
```

Return Value from Function

Many a times, we will require our functions to do some processing and return back the result. This is accomplished with the return() function in R.

Syntax of return()

return(expression)

The value returned from a function can be any valid object.

Example 1 :

```
check <- function(x) {
if (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
else {
result <- "Zero"
}
return(result)
}
```

Multiple Returns

The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

Example 1 :

```
multi_return <- function() {

my_list <- list("color" = "red", "size" = 20, "shape" = "round")

return(my_list)

}
```

R Environment and Scope

Environment can be thought of as a collection of objects (functions, variables etc.). An environment is created when we first fire up the R interpreter. Any variable we define, is now in this environment.

The top level environment available to us at the R command prompt is the global environment called R_GlobalEnv. **Global environment** can be referred to as .GlobalEnv in R codes as well.

We can use the ls() function to show what variables and functions are defined in the current environment. Moreover, we can use the environment() function to get the current environment.

```
a <- 2
> b <- 5
> f <- function(x) x<-0
> ls()
[1] "a" "b" "f"
> environment()
<environment: R_GlobalEnv>
> .GlobalEnv
<environment: R_GlobalEnv>
```

In the above example, we can see that a, b and f are in the R_GlobalEnv environment.

Notice that x (in the argument of the function) is not in this global environment. When we define a function, a new environment is created.

In the above example, the function f creates a new environment inside the global environment.

Actually an environment has a frame, which has all the objects defined, and a pointer to the enclosing (parent) environment.

Hence, x is in the frame of the new environment created by the function f. This environment will also have a pointer to R_GlobalEnv.

Example: Cascading of environments

```
f <- function(f_x){
g <- function(g_x){
print("Inside g")
print(environment())
print(ls())
}
g(5)
print("Inside f")
print(environment())
print(ls())
}
```

Here, we defined function g inside f and it is clear that they both have different environments with different objects within their respective frames.

R Programming Scope

```
outer_func <- function(){
b <- 20
inner_func <- function(){
c <- 30
}
}
a <- 10
```

Global variables

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

However, global variables also depend upon the perspective of a function.

For example, in the above example, from the perspective of inner_func(), both a and b are **global** variables.

However, from the perspective of outer_func(), b is a local variable and only a is global variable. The variable c is completely invisible to outer_func().

Local variables

On the other hand, Local variables are those variables which exist only within a certain part of a program like a function, and is released when the function call ends.

In the above program the variable c is called a local variable.

If we assign a value to a variable with the function inner_func(), the change will only be local and cannot be accessed outside the function.

This is also the same even if names of both global variable and local variables matches.

For example, if we have a function as below.

```
outer_func <- function(){
a <- 20
inner_func <- function(){
a <- 30
print(a)
}
inner_func()
print(a)

}
```

When we call it,

```
> a <- 10
> outer_func()
[1] 30
[1] 20
> print(a)

[1] 10
```

We see that the variable a is created locally within the environment frame of both the functions and is different to that of the global environment frame.

Accessing global variables

Global variables can be read but when we try to assign to it, a new local variable is created instead.

To make assignments to global variables, superassignment operator, <<-, is used.

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

If the variable is still not found, it is created and assigned at the global level.

```
outer_func <- function(){
inner_func <- function(){
a <<- 30
print(a)
}
inner_func()
print(a)
}
```

On running this function,

```
> outer_func()
[1] 30
[1] 30
> print(a)
[1] 30
```

When the statement `a <<- 30` is encountered within `inner_func()`, it looks for the variable `a` in `outer_func()` environment.

When the search fails, it searches in `R_GlobalEnv`.

Since, `a` is not defined in this global environment as well, it is created and assigned there which is now referenced and printed from within `inner_func()` as well as `outer_func()`.

R Recursive Function (Recursion)

What is Recursion?

In a recursive function (recursion), function calls itself. In this, to solve the problems, we break the programs into smaller sub-programs.

What is Recursive Function in R?

Recursive functions call themselves. They break down the problem into the smallest possible 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.

Examples of R Recursive Function

Example 1: Finding factorial of a number using the recursive function.

```
recur_factorial <- function(n) {
if(n <= 1) {
return(1)
} else {
return(n * recur_factorial(n-1))
}
}
```

Example 2: Finding the sum of natural numbers using the recursive function.

```
calculate_sum() <- function(n) {
if(n <= 1) {
return(n)
} else {
return(n + calculate_sum(n-1))
}
}
```

Example 3: Finding sum of series $1^2+2^2+3^2+.....+n^2$ using the recursive function.

```
Sum. Series <- function(number)
{
if(number == 0) {
return (0)
} else {
return ((number * number ) + Sum.Series(number - 1))
}
}
Sum.Series(3)
```

Using Functions as Arguments

In R, you can pass a function as an argument. You can also pass function code to an argument.

Then, you can assign the complete code of a function to a new object.

In the above example of rounding the value, you can pass the round() function as an argument to addPercent() function as below:

```
addPercent <- function(x, mult = 100, FUN = round, ...){
percent <- FUN(x * mult, ...)
paste(percent, "%", sep = "")
}
```

Using Anonymous Functions in R

Any function which does not have a name is called an anonymous function. They can be used for 1 liner code. You can add code as an argument in the anonymous function.

In the preceding example, you can use any function you want for the FUN argument. In fact, that function does not even need to have a name, because you copy the code as it is. So, instead of giving a function name, you can add the code as an argument in the form of a nameless or anonymous function.

Let us see this with an example:

Suppose, you have the quarterly profits of your company in a vector as follows:

> profits <- c(2100, 1430, 3580, 5230)

And, you want to report how much profit was made in each quarter relative to the total for the year. For this, you will have to use your new addPercent() function. To calculate the relative profits, you could write a rel.profit() function as follows:

>rel.profit <- function(x) round(x / sum(x) * 100)

Instead of using the function name, you can use the function body itself as an argument.

> addPercent(profits,FUN = function(x) round(x / sum(x) * 100)

R Strings

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes can not be inserted into a string starting and ending with double quotes.
- Single quote can not be inserted into a string starting and ending with single quote.

Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

```
a <- 'Start and end with single quote'
print(a)

b <- "Start and end with double quotes"
print(b)

c <- "single quote ' in between double quotes"
print(c)

d <- 'Double quotes " in between single quote'
print(d)
```

Examples of Invalid String

```
e <- 'Mixed quotes"
print(e)


f <- 'Single quote ' inside single quote'
print(f)


g <- "Double quotes " inside double quotes"
print(g)
```

String Manipulation

Concatenating Strings - paste() function

Many strings in R are combined using the paste() function. It can take any number of arguments to be combined together.

Syntax

The basic syntax for paste function is –

paste(..., sep = " ", collapse = NULL)

Following is the description of the parameters used −

- ... represents any number of arguments to be combined.
- sep represents any separator between the arguments. It is optional.
- collapse is used to eliminate the space in between two strings. But not the space within two words of one string.

Example 1

```
a <- "Hello"
b <- 'How'
c <- "are you? "
print(paste(a,b,c))
print(paste(a,b,c, sep = "-"))

print(paste(a,b,c, sep = "", collapse = ""))
```

Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using format() function.

Syntax

The basic syntax for format function is −

format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none")

Example 1:

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)

# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)

# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)

# Format treats everything as a string.
result <- format(6)
print(result)

# Numbers are padded with blank in the beginning for width.
result <- format(13.7, width = 6)
print(result)

# Left justify strings.
result <- format("Hello", width = 8, justify = "l")
print(result)

# Justfy string with center.
result <- format("Hello", width = 8, justify = "c")
print(result)
```

Counting number of characters in a string - nchar() function

This function counts the number of characters including spaces in a string.

Syntax

The basic syntax for nchar() function is −

nchar(x)

Example 1:

```
result <- nchar("Count the number of characters")
print(result)
```

Changing the case - toupper() & tolower() functions

These functions change the case of characters of a string.

Syntax

The basic syntax for toupper() & tolower() function is −

toupper(x)
tolower(x)

Example 1:

```
# Changing to Upper case.
result <- toupper("Changing To Upper")
print(result)

# Changing to lower case.
result <- tolower("Changing To Lower")
print(result)
```

Extracting parts of a string - substring() function

This function extracts parts of a String.

Syntax

The basic syntax for substring() function is −

substring(x,first,last)

Following is the description of the parameters used −

- x is the character vector input.
- first is the position of the first character to be extracted.
- last is the position of the last character to be extracted.

Example 1:

```
# Extract characters from 5th to 7th position.
result <- substring("Extract", 5, 7)
print(result)
```

toString() function

The toString function is a deviation of paste which is useful while printing vectors. This divides each element using a comma and a space. It can limit how much you want to print. Here is an example showing its use:

Example 1:

```
g <- (1:7) ^ 2
toString (g)
# [1] "1, 4, 9, 16, 25, 36, 49
```

Combining strings

To combine two or more strings, use str_c():

Example 1:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Use the sep argument to control how they're separated:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Like most other functions in R, missing values are contagious. If you want them to print as "NA", use str_replace_na():

```
x <- c("abc", NA)
str_c("|-", x, "-|")
```

As shown above, str_c() is vectorised, and it automatically recycles shorter vectors to the same length as the longest:

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Subsetting strings

You can extract parts of a string using str_sub(). As well as the string, str_sub() takes start and end arguments which give the (inclusive) position of the substring:

Example 1:

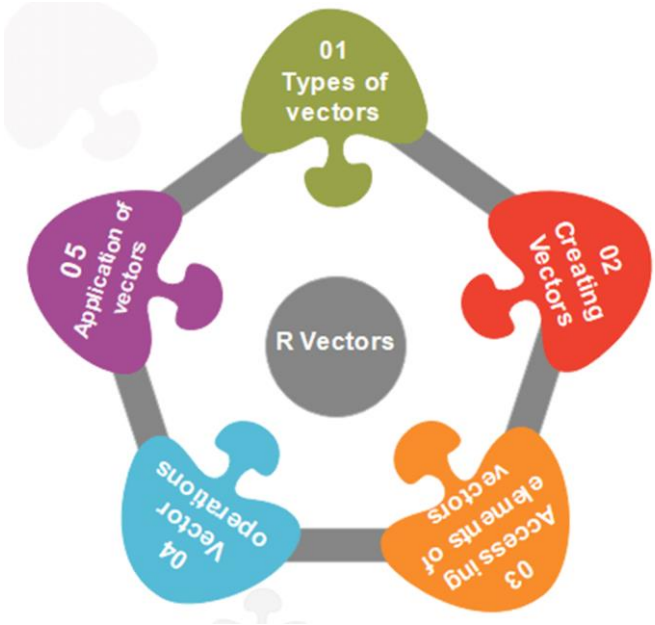
```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
```

```
# negative numbers count backwards from end
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

R - Vectors

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.Vector is a sequence of data elements which can store multiple values of similar data type.Members in a vectors are called Components Here, similar data type means all the values of a vector can be either numeric,character or logical.Vectors are dimensionless. dim() function is used to check the dimension of the vector.

Simplest way to create a vector is to use the command c().which stands for Combine.



Example:

- V1<-c(1,2,3,4,5,6,7,8,9) ----------Numeric Vector
- V2<-c("Red","Green","Yellow") ----------Character Vector
- V3<-c(TRUE,FALSE,TRUE,FALSE) ----------Logical Vector

Using the c() function

The non-character values are coerced to character type if one of the elements is a character.

The logical and numeric values are converted to characters.

```
s <- c('apple','red',5,TRUE)
```

print(s)

Nested Vector:

Vectors can be used inside a vector to create a vector.

Example 1:

vec<-c(vec1,vec2)

Example 2:

vec4<-c(vec2,11,12,13)

Creating a Numeric Vectors Using Sequence

Sequence of numbers as a vector can be created by using the colon(:)

Example:

vec6<-1:50

Sequence Function :

The seq() function can be used to generate the sequence of values.

Syntax:

Seq(from,to)

From:starting value of sequence.

To:end value of sequence.

Example 1 :Printing the values from 1 to 15

vec7<-seq(1,15)

Example 2:Printing the odd numbers from 1 to 20

Odd<-seq(1,20,2)

Example 3:Printing the even numbers from 1 to 20

Even<-seq(2,20,2)

Example 4:Printing the decremental numbers from 40 to 2

Even2<-seq(40,2,-2)

Example 5:Printing the even numbers from 1 to 20 by using the keywords

Even3<-seq(to=20,by=2,from=2)

Example 6 : Creating a sequence from 5 to 13.

```
v <- 5:13
print(v)
```

Example 7: Creating a sequence from 6.6 to 12.6.

```
v <- 6.6:12.6
print(v)
```

Example 8: If the final element specified does not belong to the sequence then it is discarded.

```
v <- 3.8:11.4
print(v)
```

Example 9: Create vector with elements from 5 to 9 incrementing by 0.4.

```
print(seq(5, 9, by = 0.4))
```

Assigning Names to the values

Names can be assigned to the components of the vector in two ways:

Method 1: Using combine method:

Names(odd)<-c('odd1' , 'odd2', 'odd3',…….'odd10')

Example 1:

Names(vec1)<-c("e1","e2","e3")

Example 2:

Names(even)<-paste0("even",1:10)

Length Function:

e length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.Length function is used to find the length of the vector.

Example 1 :

length(even)

Example 2:

names(v10)<-paste("E",1:length(v10))

Vector Creation

Single Element Vector

Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

```
# Atomic vector of type character.
print("abc");
# Atomic vector of type double.
print(12.5)
# Atomic vector of type integer.
print(63L)
# Atomic vector of type logical.
print(TRUE)
# Atomic vector of type complex.
print(2+3i)
```

Accessing the Values of the Vector

Value of the vector can be accessed in three ways:

1.Through their INDEX

2.Through their NAMES

3.Through their Logical Refrences

Method 1: Through their INDEX

Elements of a Vector are accessed using indexing. The [ ] brackets are used for indexing. Indexing starts with position 1. Giving a negative value in the index drops that element from result.TRUE, FALSE or 0 and 1 can also be used for indexing.

Vec_name[index]

Example 1:

Even[5]

Even[7]

Example 2:

Even[c(1,2,3)]

Even[c(3,5,9)]

Example 3:

Even[3:8]

Example 4:

Even[seq(1,10,2)]

Example 5:

Odd[seq(2,10,2)]

Example 6:

Odd[-5]

Odd[c(-3,-4,-5)]

Method 2: Through their NAMES

Example 1:

Odd[“odd5”]

Odd[c(“odd3”,”odd4”,”odd5”)]

Example 2:

Odd[paste0(“odd”,1:5)]

Method 3: Through their LOGICAL REFERENCES

True will return the value

False will surpress the value

Example 1:

Even[c(T,F,T,F,T,F,T,F,T,F)]

Even[c(T,F)]

Vector Manipulation

Vector arithmetic

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

Example:

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)
# Vector addition.
add.result <- v1+v2
print(add.result)
# Vector subtraction.
sub.result <- v1-v2
print(sub.result)
# Vector multiplication.
multi.result <- v1*v2
print(multi.result)
# Vector division.
divi.result <- v1/v2
print(divi.result)
```

Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

Example:

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11)
# V2 becomes c(4,11,4,11,4,11)
add.result <- v1+v2
print(add.result)
sub.result <- v1-v2
print(sub.result)
```

Vector Element Sorting

Elements in a vector can be sorted using the sort() function.

Example:

```
v <- c(3,8,4,5,0,11, -9, 304)
# Sort the elements of the vector.
sort.result <- sort(v)
print(sort.result)
# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
print(sort.result)
# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

Programs of Vectors

- Create a vector v1 with numbers from 1 to 15.
- Create a vector v2 with numbers 6 to 25.
- Create a vector v3 with names of Indian Cities.
- Find the 5th value of the v1 vector.
- Get the 5th,6th and 7th value of the v1 vector
- Get the 3rd,1st and 5th value of the v2 number.
- Find the last three values of the v2 vector.
- Exclude the 10th value from the v1 number.

Giving the names to the vector.

- Create a even vector from vector 1 to 30
- Create a odd vector from 1 to 30
- Assign the name even1,even2….. to the objects of the vector by using the combine command.
- Assign the name odd1,odd2….. to the objects of the vector by using the paste0() command.

Accessing the Values of the Vector.

Create a v3 vector with the series from 1 to 18

Create a v4 vector with the numbers from 4 to 23.

Accessing the values using the object names.

- Find the 5th value of the v3 vector.
- Get the 5th,6th and 7th value of the v4 vector.

Accessing the values using the Logical References.

- Find the last 3 values of the v4 vector.
- Exclude the 9th value from the v3 vector.

R Lists

In R, lists are the second type of vector. Lists are the objects of R which contain elements of different types such as number, vectors, string and another list inside it. It can also contain a function or a matrix as its elements. A list is a data structure which has components of mixed data types. We can say, a list is a generic vector which contains other objects.

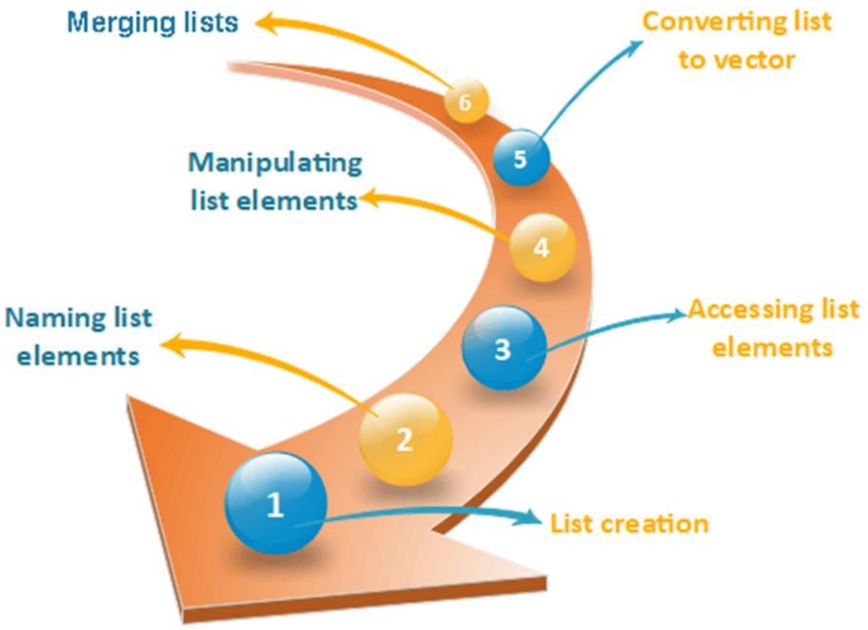
Example 1:

```
vec <- c(3,4,5,6)
char_vec<-c("shubham","nishka","gunjan","sumit")
logic_vec<-c(TRUE,FALSE,FALSE,TRUE)
out_list<-list(vec,char_vec,logic_vec)
print(out_list)
```

Example 2:

```
# Create a list containing strings, numbers, vectors and a logical
# values.
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

Lists in R programming



Lists creation

The process of creating a list is the same as a vector. In R, the vector is created with the help of c() function. Like c() function, there is another function, i.e., list() which is used to create a list in R. A list avoid the drawback of the vector which is data type. We can add the elements in the list of different data types.

Syntax

1. list()

Example 1:

```
Creating list with same data type
list_1<-list(1,2,3)
list_2<-list("Shubham","Arpita","Vaishali")
list_3<-list(c(1,2,3))
list_4<-list(TRUE,FALSE,TRUE)
list_1
list_2
list_3
```

```
list_4
```

Giving a name to list elements

R provides a very easy way for accessing elements, i.e., by giving the name to each element of a list. By assigning names to the elements, we can access the element easily. There are only three steps to print the list data corresponding to the name:

- Creating a list.
- Assign a name to the list elements with the help of names() function.
- Print the list data.

Example 1:

```
# Creating a list containing a vector, a matrix and a list.
list_data <- list(c("Ajay","Nishika","Gunjan",”Rupali”,”karan”), matrix(c(40,80,60,70,90,80), nrow = 2),
  list("BCA","MCA","B.tech"))
 # Giving names to the elements in the list.
names(list_data) <- c("Students", "Marks", "Course")
 # Show the list.
print(list_data)
```

Accessing List Elements

R provides two ways through which we can access the elements of a list. First one is the indexing method performed in the same way as a vector. In the second one, we can access the elements of a list with the help of names. It will be possible only with the named list.; we cannot access the elements of a list using names if the list is normal.

Example 1: Accessing elements using index

```
# Creating a list containing a vector, a matrix and a list.
list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),
  list("BCA","MCA","B.tech"))
# Accessing the first element of the list.
print(list_data[1])
 # Accessing the third element. The third element is also a list, so all its elements will be printed.
print(list_data[3])
```

Example 2: Accessing elements using names

```
# Creating a list containing a vector, a matrix and a list.
list_data <-
 list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),list("BCA","MCA","B.tech
"))
# Giving names to the elements in the list.
names(list_data) <- c("Student", "Marks", "Course")
# Accessing the first element of the list.
print(list_data["Student"])
print(list_data$Marks)
print(list_data)
```

Manipulation of list elements

R allows us to add, delete, or update elements in the list. We can update an element of a list from anywhere, but elements can add or delete only at the end of the list. To remove an element from a specified index, we will assign it a null value. We can update the element of a list by overriding it from the new value. Let see an example to understand how we can add, delete, or update the elements in the list.

Example 1:

```
# Creating a list containing a vector, a matrix and a list.
list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),
  list("BCA","MCA","B.tech"))

# Giving names to the elements in the list.
names(list_data) <- c("Student", "Marks", "Course")

# Adding element at the end of the list.
list_data[4] <- "Moradabad"
print(list_data[4])

# Removing the last element.
list_data[4] <- NULL

# Printing the 4th Element.
print(list_data[4])

# Updating the 3rd Element.
list_data[3] <- "Masters of computer applications"
print(list_data[3])
```

Converting list to vector

There is a drawback with the list, i.e., we cannot perform all the arithmetic operations on list elements. To remove this, drawback R provides unlist() function. This function converts the list into vectors. In some cases, it is required to convert a list into a vector so that we can use the elements of the vector for further manipulation.

The unlist() function takes the list as a parameter and change into a vector. Let see an example to understand how to unlist() function is used in R.

Example 1:

```
# Creating lists.
list1 <- list(10:20)
print(list1)

list2 <-list(5:14)
print(list2)

# Converting the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
print(v2)

adding the vectors
result <- v1+v2
print(result)
```

Merging Lists

R allows us to merge one or more lists into one list. Merging is done with the help of the list() function also. To merge the lists, we have to pass all the lists into list function as a parameter, and it returns a list which contains all the elements which are present in the lists. Let see an example to understand how the merging process is done.

Example 1:

```
# Creating two lists.
Even_list <- list(2,4,6,8,10)
Odd_list <- list(1,3,5,7,9)


# Merging the two lists.
merged.list <- list(Even_list,Odd_list)


# Printing the merged list.
print(merged.list)
```

How to Generate Lists in R

We can use a colon to generate a list of numbers.

Example 1:

> -5:5

c function in R

The c function in R combines the parameter into a list and converts them to the same type.

Example 1:

> c("April", 4)

> typeof("4")

Here 4 is converts into a string.

R Predefined Lists

Lists for letters and month names are predefined:

- letters
- LETTERS
- month.abb
- month.name