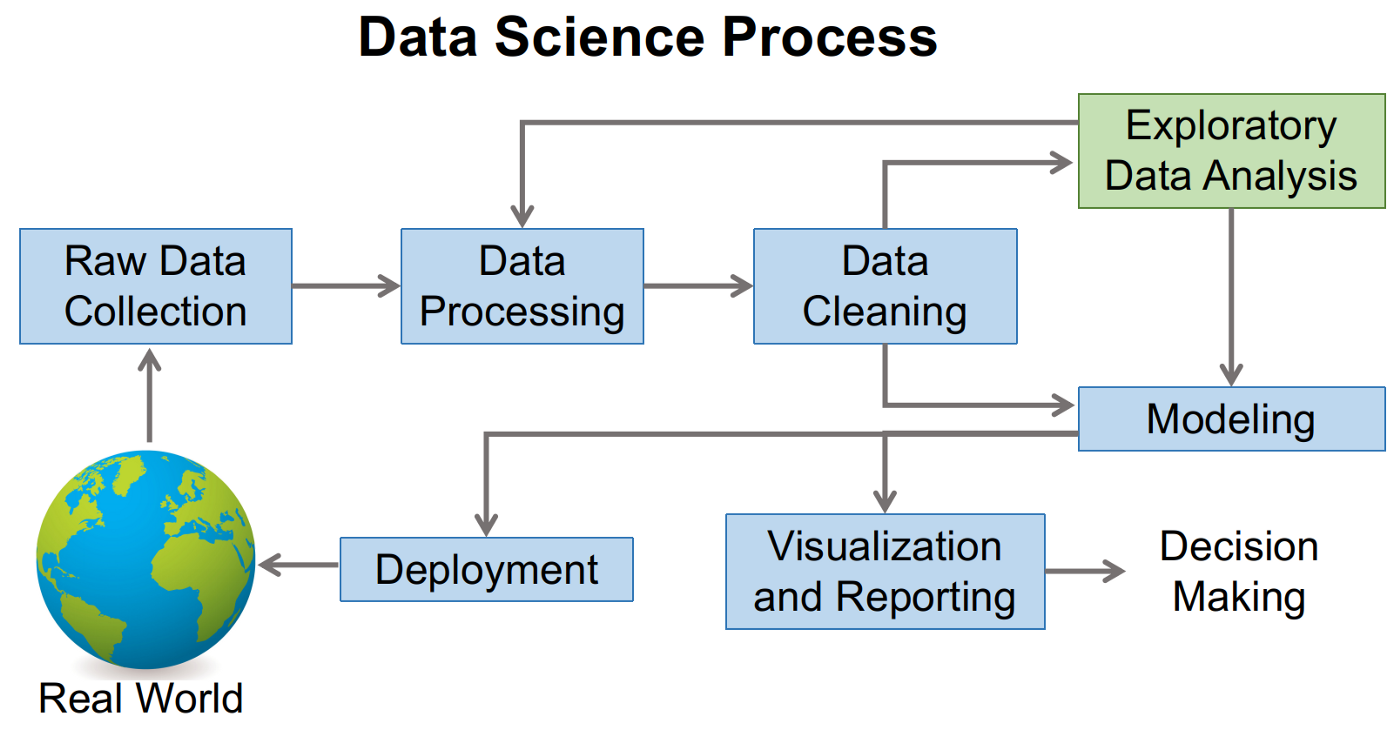
**Define Data science**

Data science is the area of study which involves **extracting knowledge from all the data** you can gather. Here we are translating a business problem into a research problem and then translating the result back into a practical solution. People doing all this are called data scientists.

**Data Science Life Cycle**

**Step 1:** Define Problem Statement: In meeting with clients, data scientist must ask relevant mquestions to understand and define objectives of the problem that need to be tackled.

**Step 2:** Data Collection: If there is no data available, then, you need to collect new data. This method is called primary data collection.

Another method is to use the data which is readily available. This is called secondary data collection. Here raw data is gathered from multiple sources like flat files and online repositories etc

The data collected need to be prepared for extracting knowledge

**Step 3:** Data preparation: The most essential part of any data science project is data preparation. It consumes 60% of the time spent on the project.

Steps in data preparation are

1. Data cleaning: It handles missing values, NULL or unwanted values, duplicate values, misspelt attributes, inconsistent data types. Handling outliers: Outliers are observations which are distant from the rest of the data. Outliers can be good or bad for the data. Outliers can be used for fraud detection. Scatter plots and box plots help to identify the outliers in the data.

2. Data transformation: turns raw data formats into desired outputs. It also normalizes the data. Normalisation is done in order to scale the data values in a specified range (-1.0 to 1.0 or 0.0 to 1.0)

For example : Consider the data below:

2001 pens 300

2001 pencils 400

2002 pens 800

2002 pencils 200

The above data can be transformed as shown below. This table format helps us to summarise quickly.

|  | 2001 | 2002 |
| --- | --- | --- |
| pens | 300 | 800 |
| pencils | 400 | 200 |

3. Data integration: When data from multiple sources are integrated, the data after integration must be accurate and reliable. Primary keys and foreign keys are handled while integrating data.

4. Data reduction: Here we reduce the size of data by eliminating duplicate columns , unnecessary columns etc..

**Step 4:** Data mining or Exploratory Data Analysis (EDA):

EDA helps us understand what we can actually do with the data. EDA helps understand the relationships between data and helps us in **selecting the variables** that will be used in model development. It also helps us in identifying the right algorithm.

Softwares available: tableau

**Step 5:** Model building: The model is built by selecting a machine learning algorithm that suits the data. Regression is used to predict continuous values

**Example:** predicting house prices, temperature etc..

and classification is used to predict discrete values. Example: classifying whether the email is spam or not. Customer will buy a product or not. **For modeling the data, we can use the tools: R , python or SAS**

**Step 6:** Visualization and communication: the business finding are communicated to business clients using simple and effective manner to convince the client. The visualization tools like tableau, powerBi, QlikView can be used to create powerful reports and dashboards

**Q) Data Science in various fields**

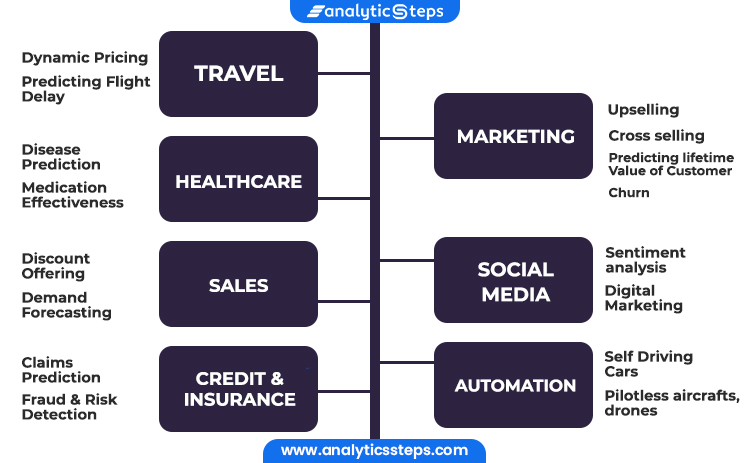
Business:

* Customer Segmentation: Grouping customers based on their behavior for targeted marketing campaigns.
* Fraud Detection: Identifying suspicious activities in financial transactions.
* Product Recommendation: Recommending products to customers based on their past purchases.

Agriculture:

* Crop Yield Prediction: Forecasting crop yields based on weather data and soil conditions.
* Disease Detection: Identifying plant diseases using image recognition techniques.
* Precision Farming: Optimizing resource usage based on real-time data from sensors.

<https://data-flair.training/blogs/data-science-applications/>





Data science in agriculture

Data science in manufacturing

Data science in health care

**Q) Impact of Data Science**

How data science impact society [**https://onlinemasters.ohio.edu/blog/data-science-for-social-good/**](https://onlinemasters.ohio.edu/blog/data-science-for-social-good/)

**Q) Data Analytics Life Cycle**

[**https://www.geeksforgeeks.org/life-cycle-phases-of-data-analytics/**](https://www.geeksforgeeks.org/life-cycle-phases-of-data-analytics/)

**Q) data science toolkit**

[**https://www.springboard.com/blog/data-science/open-source-data-science-tools/**](https://www.springboard.com/blog/data-science/open-source-data-science-tools/)

**Q) data science team**

Since, data is the most essential ingredient of any data science strategy the first people you need are Data Engineers. A data engineer is generally someone with good programming and hardware skills, and who can be build your data infrastructure. It depends on the size of your data but they are generally comfortable working with big data and cloud technologies, know how to build data pipelines, design databases, and pull data out of them. They will know how to do look at the data at a basic level, and do simple aggregations for sanity checks on the data quality, but are not necessarily experts in analyzing it.

After you’ve built out your data infrastructure and you will need people who can take that data, clean it, analyze it, run experiments on it and communicate the results. Depending on your business requirements the exact skills varies. Most of the time this work is done by Data Analysts who are skilled in the processing and cleaning data, creating statistical inference or predictive models, running experiments, plotting those results, creating reports and providing insights to stakeholders higher up. They’ll mostly be working in Jupyter notebook or Rstudio, and have a combination of programming, statistics and machine learning knowledge. We shouldn’t expect them to write production quality code though.

That brings me to the next role. If you are building a data product you will need Machine Learning Engineers in your team. These are not researchers who build machine learning algorithms per se but are data-focused software developers familiar with various data science libraries and know how to write production quality code based on the models developed by the analysts. To do this job they must work closely with Data Engineers, or it might be also done by a math savvy Data Engineer for smaller teams. Most developers looking to enter the data science field should consider this as a good career option.

Sometimes it might also be useful to have a more design-focused Data Visualization Specialist to create highly polished charts and reports to communicate the results of the analysis.

I tend to think of a Data Scientist as someone who is above average in all of the aforementioned roles, and also who knows how to work with the Domain Experts to deliver results. These are collaborators generally outside your team or organization that you bring in to leverage their subject-matter expertise in cases such as medicine, finance, economics, marketing, law etc.

If you are working on problems that require some custom or proprietary data science algorithms, it might then be time to hire someone with a PhD or core research background. They’ll probably have a deep understanding of the theory and algorithms behind an AI field like Conversational AI, Computer Vision, Robotics, Reinforcement Learning, Graphical Models etc. I tend to like the title Research Engineer or Research Scientist for a role like this.

Another important but less talked about role in a data science team is that of a Data Science Manager or Head of Data Science. For smaller teams it might be sufficient to have a senior member of the team who has a good knowledge of all the different roles lead the team. But once the team grows you will probably need someone with a strong technological and business strategy background.

Data Science managers are hands-on leaders who will build the foundation of your data science strategy, recruit and build your team, make sure everybody interacts with each other, have the data and information that they need, and develop the process that the whole team can follow. They are the data team’s interface to the rest of the organization, collaborators and executives. They translate the complex AI jargon to non-experts, and make sure their work is aligned with the strategy of the organization as a whole.

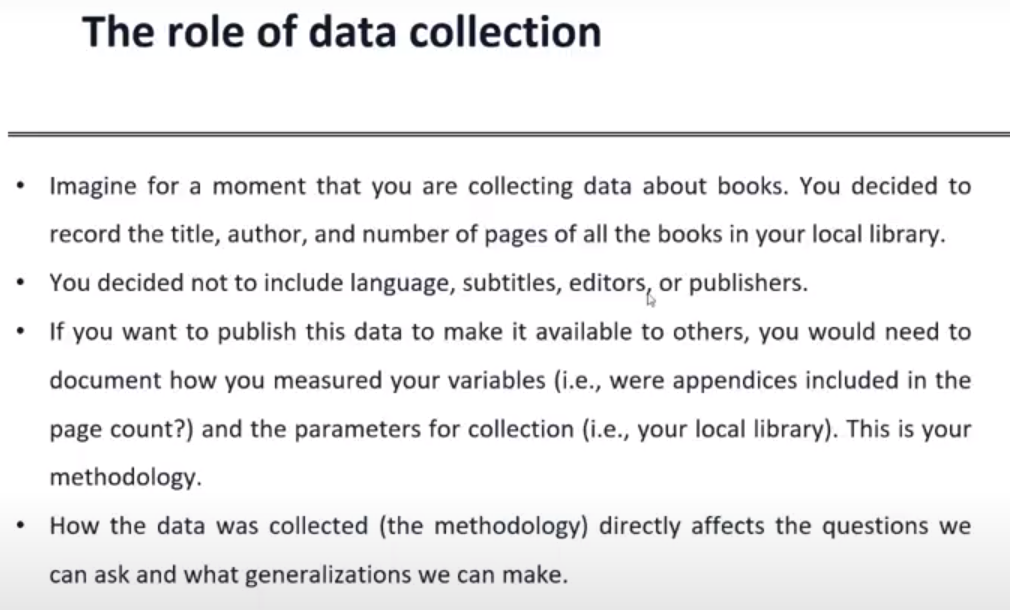
a data scientist is a person who solves business tasks using machine learning and data mining techniques.The role of a data scientist is data preparation and cleaning with further model training and evaluation.

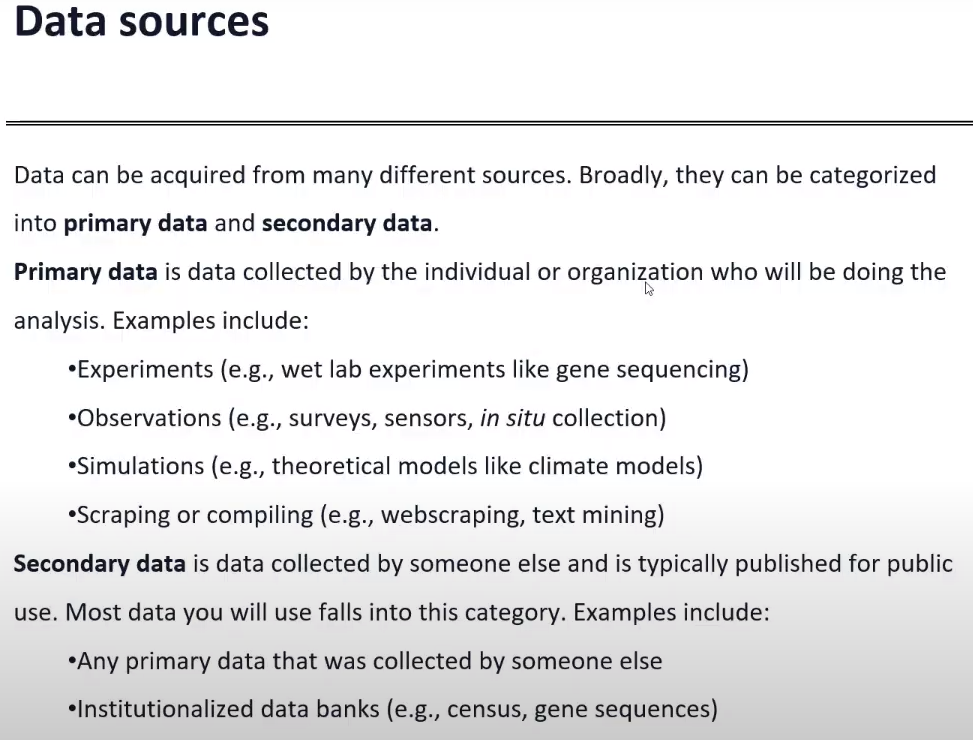
data scientists are the core members of a team. They use statistical methods, machine learning algorithms and other tools to analyse data and create predictive models; some also build data products, recommendation engines, chatbots and other technologies for various use cases.

A machine learning engineer combines software engineering and modelling skills by determining which model to use and what data should be used for each model. Probability and statistics are also their forte. Everything that goes into training, monitoring, and maintaining a model is ML engineer’s job.

<https://www.kdnuggets.com/2019/03/building-effective-data-science-teams.html>

**Q) EXPLAIN ABOUT DATA COLLECTION**





**Q) Introduction to data**

When analysing your data, it is important to recognize and understand the importance of each data type. Depending on the type of your data, a specific analysis will be appropriate. Similarly, the data type will also drive the choice of data visualisation techniques.

**Q) Types of Data: Numeric – Categorical – Graphical – High Dimensional Data (or) data attributes**

It can be seen as a data field that represents the characteristics or features of a data object. For a customer, object attributes can be customer Id, address, etc. We can say that a set of attributes used to describe a given object are known as attribute vector or feature vector.

Type of attributes :

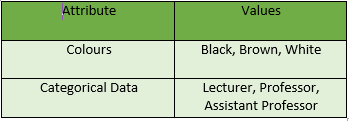
This is the First step of Data Data-preprocessing. We differentiate between different types of attributes and then preprocess the data. So here is the description of attribute types.

* Qualitative (Nominal (N), Ordinal (O), Binary(B)).
* Quantitative (Numeric, Discrete, Continuous)

**Qualitative Attributes:**

1. Nominal Attributes – related to names: The values of a Nominal attribute are names of things, some kind of symbols. Values of Nominal attributes represents some category or state and that’s why nominal attribute also referred as categorical attributes and there is no order (rank, position) among values of the nominal attribute.

Example of nominal data:



2. Binary Attributes: Binary data has only 2 values/states. For Example yes or no, affected or unaffected, true or false. There are 2 types of binary data.

* Symmetric: Both values are equally important (Gender).
* Asymmetric: Both values are not equally important (Result).



Table

Description automatically generated

3. Ordinal Attributes : The Ordinal Attributes contains values that have a meaningful sequence or ranking(order) between them, but the magnitude between values is not actually known, the order of values that shows what is important but don’t indicate how important it is.

Table

Description automatically generated

**Quantitative Attributes:**

1. Numeric: A numeric attribute is quantitative because, it is a measurable quantity, represented in integer or real values. Numerical attributes are of 2 types, interval, and ratio.

An interval-scaled attribute has values, whose differences are interpretable, but the numerical attributes do not have the correct reference point, or we can call zero points. Data can be added and subtracted at an interval scale but can not be multiplied or divided. Consider an example of temperature in degrees Centigrade. If a day’s temperature of one day is twice of the other day we cannot say that one day is twice as hot as another day.

A ratio-scaled attribute is a numeric attribute with a fix zero-point. If a measurement is ratio-scaled, we can say of a value as being a multiple (or ratio) of another value. The values are ordered, and we can also compute the difference between values, and the mean, median, mode, Quantile-range, and Five number summary can be given.

2. Discrete : Discrete data have finite values it can be numerical and can also be in categorical form. These attributes has finite or countably infinite set of values.

Example:

Table

Description automatically generated

3. Continuous: Continuous data have an infinite no of states. Continuous data is of float type. There can be many values between 2 and 3.

Example :

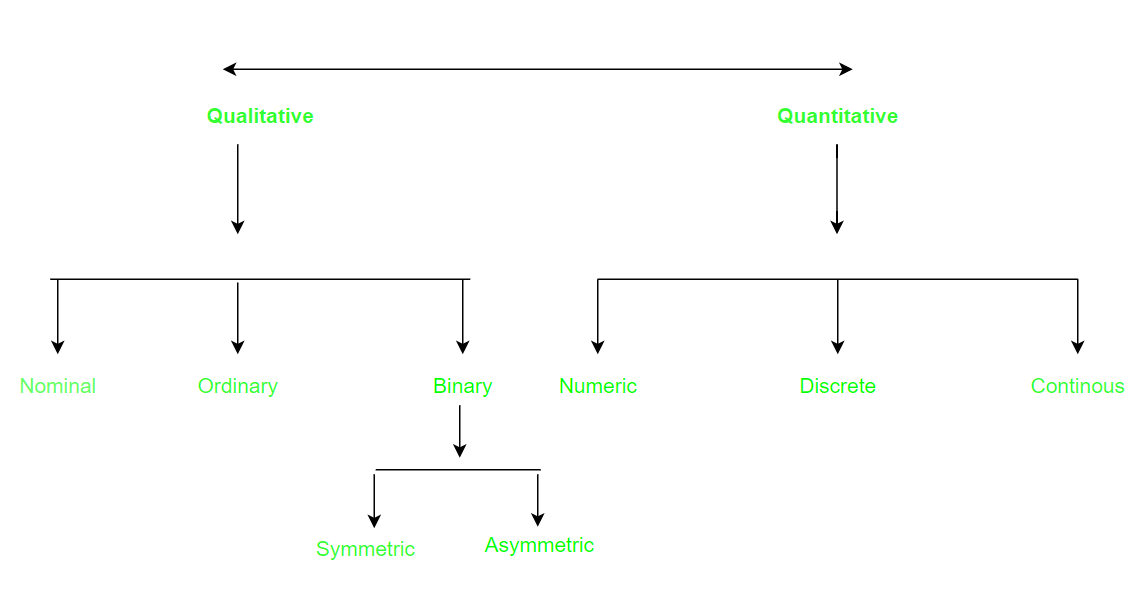
Table

Description automatically generated

Resources:

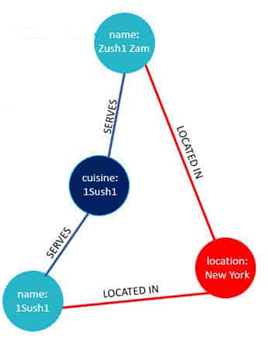
<https://www.geeksforgeeks.org/understanding-data-attribute-types-qualitative-and-quantitative/>

<https://www.geeksforgeeks.org/data-mining-data-attributes-and-quality/>



**High dimensional data** refers to **a dataset in which the number of features/columns p is larger than the number of observations/rows N**, often written as p >> N

**Graphical data**

Graphical data: The data is stored in the shape of a graph, like a network.

The nodes in graph are noun sets, person, place, thing,

And these nodes could also have properties,

And then you could have relationships between those nodes

So just like you could add different types of nodes,you can also have different types of relationships,

you could put properties in those relationships,you could put values, so they could be weighted,you could have geospatial information,you could have date and time.

Graphical data is used

1. whenever you have data that's complexity connected
2. you want to understand how these things are connected to each other
3. you want to find a shortest path
4. you're looking for patterns in your data or combination of patterns.
5. if you're doing fraud detection,
6. any kind of graph algorithmtype of analysis,

Example: Neo4j is a native graph database.

<https://k21academy.com/microsoft-azure/dp-900/structured-data-vs-unstructured-data-vs-semi-structured-data/>

<https://unstop.com/blog/difference-between-structured-and-unstructured-data>

<https://k21academy.com/microsoft-azure/dp-900/structured-data-vs-unstructured-data-vs-semi-structured-data/>

**Q) Classification of digital Data:**

Process of classifying data in relevant categories so that it can be used or applied more efficiently. The classification of data makes it easy for the user to retrieve it.   
**Types of Data Classification :**

Data can be broadly classified into 3 types.  
**1.** **Structured Data :**Structured data is created using a **fixed structure** and is mostly maintained in **table format**. It contains all the data which can be stored in the SQL database in a tabular format. SQL databases provide an easy way to manage information.

**Examples –**

Account data or Relational data, Geo-location, credit card numbers, addresses, etc.

Consider an example for Relational Data like you have to maintain students data for a university like the name of the student, ID of a student, address, and Email of the student. To store the record of students used the following relational schema and table for the same.

| **ID** | **Name** | **Address** | **Email** |
| --- | --- | --- | --- |
| 1001 | A | Delhi | A@gmail.com |
| 1002 | B | Mumbai | B@gmail.com |

**2. Unstructured Data :**

It is defined as the data in which **does not follow any organised format**. Unstructured data is also very important for the big data domain. To manage and store Unstructured data there are many platforms to handle it like No-SQL Database.

**Examples –**

Word, PDF, text, media logs, etc.

**3. Semi-Structured Data :**

Semi-structured data is information that does not reside in a relational database but that has some organisational properties that make it easier to analyse. With some process, you can store them in a relational database but it is very hard for some kind of semi-structured data, but semi-structured exist to ease space.

**Example –**

XML data.

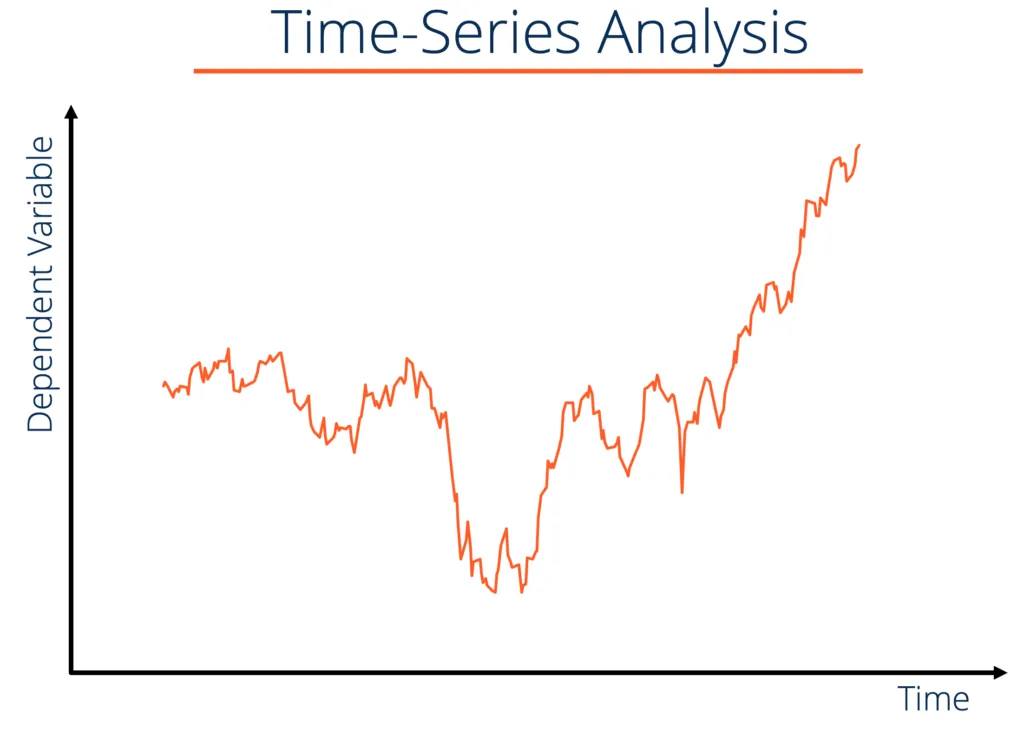
**Features of Data Classification :**

The main goal of the organisation of data is to arrange the data in such a form that it becomes fairly available to the users. So it’s basic features are as follows.

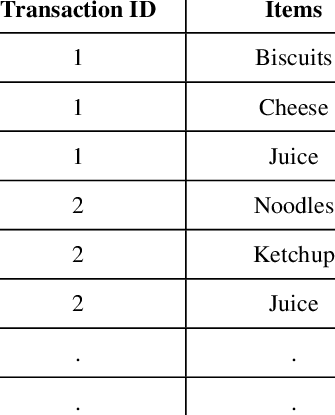
* **Homogeneity –** The data items in a particular group should be similar to each other.
* **Clarity –** There must be no confusion in the positioning of any data item in a particular group.
* **Stability –** The data item set must be stable i.e. any investigation should not affect the same set of classification.
* **Elastic –** One should be able to change the basis of classification as the purpose of classification changes.

**Q) Sources of Data: Time Series – Transactional Data –Biological Data – Spatial Data – Social Network Data (or) Types of databases**

<http://webdocs.cs.ualberta.ca/~zaiane/courses/cmput690/notes/Chapter1/>

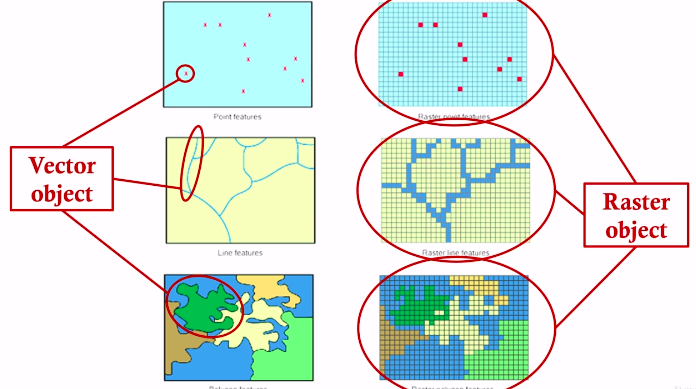
**Time Series Data: It refers to data collected over a period of time, such as stock prices or weather conditions, allowing for analysis of patterns and trends.  
**

**Transactional Data: This type of data records individual transactions, such as purchases or financial transactions, providing insights into customer behaviour and business operations.**

****

**Biological Data: It contains information related to living organisms, such as genetic sequences or physiological measurements.This data aids research in areas like genetics, medicine, and ecology.**

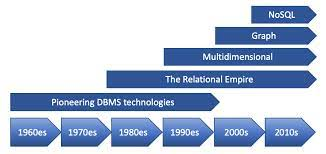
**Spatial Data: Spatial data contains information about physical locations and. This data helps in analysis and visualisation of geographic patterns, such as maps or satellite imagery.**

****

**Social Network Data: It involves data about individuals and their relationships in a social network, offering insights into social interactions, influence, and community structures.**

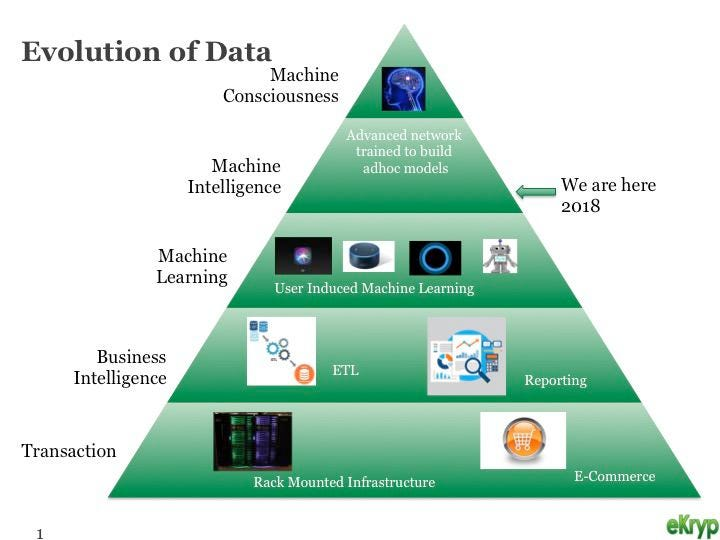
****

**Q) Database Evolution**





**Q) Evolution of data:** picture taken from forbes.com



**Unit-2**

**Q) Introduction to R**

R is an open-source programming language that is widely used as a statistical software and data analysis tool.

R is an interpreted language, and R is a multi-paradigm language, and that means it has multiple paradigms that are: object-oriented, procedural, reflective, etc.

**Q) Features of R**

1. Open-source

R is an open-source software environment. It is free of cost and can be adjusted and adapted according to the user’s and the project’s requirements.

You can make improvements and add packages for additional functionalities.

R is freely available. You can learn how to install R, Download and start practicing it.

2. Strong Graphical Capabilities

R can produce static graphics with production quality visualizations and has extended libraries providing interactive graphic capabilities.

This makes data visualization and data representation very easy.

From concise charts to elaborate and interactive flow diagrams, all are well within R’s repertoire. Look at the attractive graphical visualizations in R.

3. Highly Active Community

R has an open-source library which is supported by its growing number of users.

The R environment is continuously growing. This growth is due to its large user-base.

4. A Wide Selection of Packages

CRAN or Comprehensive R Archive Network houses more than 10,000 different packages and extensions that help solve all sorts of problems in data science.

High-quality interactive graphics, web application development, quantitative analysis or machine learning procedures, there is a package for every scenario available.

R contains a sea of packages for all the forms of disciplines like astronomy, biology, etc. While R was originally used for academic purposes, it is now being used in industries as well.

5. Comprehensive Environment

R has a very comprehensive development environment meaning it helps in statistical computing as well as software development.

R is an object-oriented programming language. It also has a robust package called Rshiny which can be used to produce full-fledged web apps.

Combined with data analysis and data visualization, R can be used for highly interactive online data-driven storytelling.

6. Can Perform Complex Statistical Calculations

R can be used to perform simple and complex mathematical and statistical calculations on data objects of a wide variety. It can also perform such operations on large data sets.

7. Distributed Computing

In distributed computing, tasks are split between multiple processing nodes to reduce processing time and increase efficiency.

R has packages like ddR and multiDplyr that enable it to use distributed computing to process large data sets.

8. Running Code Without a Compiler

R is an interpreted language which means that it does not need a compiler to make a program from the code. R directly interprets provided code into lower-level calls and pre-compiled code.

9. Interfacing with Databases

R contains several packages that enable it to interact with databases like Roracle, Open Database Connectivity Protocol, RmySQL, etc.

10. Data Variety

R can handle a variety of structured and unstructured data. It also provides various data modeling and data operation facilities due to its interaction with databases.

11. Machine Learning

R can be used for machine learning as well. The best use of R when it comes to machine learning is in case of exploration or when building one-off models.

12. Data Wrangling

Data wrangling is the process of cleaning complex and inconsistent data sets to enable convenient computation and further analysis. This is a very time taking process.

R with its extensive library of tools can be used for database manipulation and wrangling.

13. Cross-platform Support

Cross Platform compatible with r

R is machine-independent. It supports the cross-platform operation. Therefore, it can be used on many different operating systems.

14. Compatible with Other Programming Languages

While most of its functions are written in R itself, C, C++ or FORTRAN can be used for computationally heavy tasks. Java, .NET, Python, C, C++, and FORTRAN can also be used to manipulate objects directly.

15. Data Handling and Storage

R is integrated with all the formats of data storage due to which data handling becomes easy.

16. Vector Arithmetic

Vectors are the most basic data structure in R, and most other data structures are derived from vectors.

R uses vectors and vector arithmetic and does not need a lot of looping to process a large set of values. This makes R much more efficient.

17. Compatibility with Other Data Processing Technologies

R can be easily paired with other data processing and distributed computing technologies like Hadoop and Spark. It is possible to remotely use a Spark cluster to process large datasets using R.

R and Hadoop can be paired as well to combine Hadoop’s large scale data processing and distributing computing capabilities with R’s statistical computing power.

18. Generates Report in any Desired Format

R’s markdown package is the only report generation package you will ever need when working with R. The markdown package can help produce web pages.

It can also generate reports in the form of word documents or PowerPoint presentations. All with your R code and results embedded into them

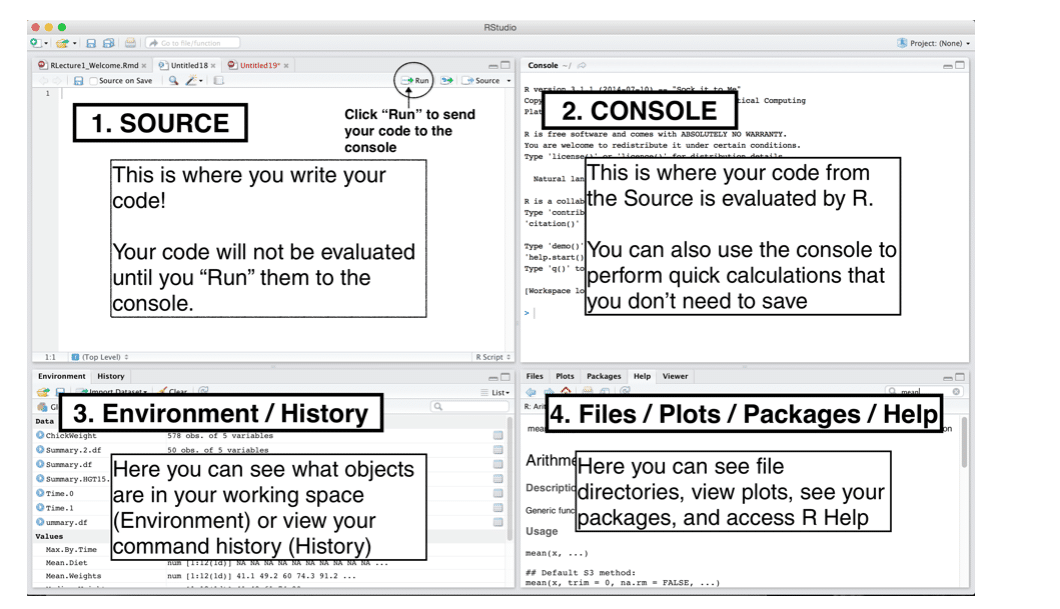
**Q) Environment-RStudio**

R can even be run in a shell environment like Linux:

RStudio is not functional without an installation of R. You therefore need to install R regardless of whether or not you use RStudio.

R studio is a software that provides an interface to R. It’s referred to as an Integrated Development Environment (IDE).

RStudio by default displays four panes: Console, Source Code, Environment/History, and Files. You can rearrange them by going to View -> Panes -> Pane Layout.



**Q) R assignment operator**

Assignment operator in R is left-arrow operator (<-).

On the following example, I want to assign the numeric value 3 to object called x.

x <- 3

You can also use equal to = operator

x = 3

If you want to know the value of an object just simply call the object or use print() function.

x

[1] 3

# or

print(x)

[1] 3

The another way to assign value to an object in R is using the assign() function. This function used when the object is a string.

Let’s say you have object x which have a string value, x <- "a". You want to create object based on x’s value without mention it explicitly.

x <- "a"

So, i want to assign value 7 to a object which is the value of x.

assign(x, 7)

**Q) Basics of R-Modes** In R, data modes and classes define the fundamental attributes and behavior of a data object. For example, different modes and classes are handled differently by core functions like print(), summary(), and plot().

#### ***Data Object Modes***

All data in R is an object and all objects have a “mode.” The **mode determines what type of information can be found within the object** and how that information is stored.  **Atomic “modes” are the basic building blocks for data objects in R.** There are 6 basic atomic modes:

| **Data Mode** | **Storage** | **Example** |
| --- | --- | --- |
| logical | logical | TRUE or FALSE |
| numeric | integer, single or double | Floating point real numbers; 3, o.753, 1.e+200 |
| complex | complex | 3 + 1.23i |
| character | character | strings in quotes (“) or apostrophes (‘) |
| function | special or built-in | do.it <- function(x) {…} |
| name | symbol | any name assigned to an object (e.g. "my.data") |

The atomic mode and the storage mode of any data object can be obtained as follows:

R

| 1  2 | > mode(x)  > storage.mode(x) |
| --- | --- |

#### ***Data Object Classes***

R has an elaborate class system. If the number of modes is limited, the number of object classes is numerous and custom classes can be created. The class of an object is used to support object-oriented programming in R. Specifically, the class defines how an object will be treated by generic functions like print(), plot(), summary(), etc.

Important classes, including custom classes from popular packages, are listed below:

* Vector data is classed in R using its mode as either *numeric*, *character*, etc.
* Rectangular data is classed in R as either a *matrix*, *data frame* or *list* object;
* Sourced scripts assume the class *function;*
* Categorical information are classed as *factor* objects;
* Time/date information in R is classed as a *timeDate* object; and
* Spatial GIS data can assume a wide range of classes, including *SpatialPoints*, *SpatialLines*, *SpatialPolygons*, and more.

The class of any object can be interrogated and quickly determined using the class() function. The example below creates a simple matrix and obtains its class:

R

| 1  2  3  4  5  6  7  8 | > x <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3)  > x  [,1] [,2] [,3]  [1,] 1 3 12  [2,] 2 11 13    > class(x)  [1] "matrix" |
| --- | --- |

At the same time, the attributes() functions can be used to identify other attributes of an object. The attr() function does the same, but supports additional arguments for more precise diagnostics.

# **Q) R Operators**

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

### Example

10 + 5

R divides the operators in the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Miscellaneous operators

## **R Arithmetic Operators**

Arithmetic operators are used with numeric values to perform common mathematical operations:

| **Operator** | **Name** | **Example** |
| --- | --- | --- |
| + | Addition | x + y |
| - | Subtraction | x - y |
| \* | Multiplication | x \* y |
| / | Division | x / y |
| ^ (or) \*\* | Exponent | x ^ y |
| %% | Modulus (Remainder from division) | x %% y |
| %/% | Integer Division | x%/%y |

**Comparison operators** are used to compare two values:

| **Operator** | **Name** | **Example** |
| --- | --- | --- |
| == | Equal | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

## **R Logical Operators**

Logical operators are used to combine conditional statements:

| **Operator** | **Description** |
| --- | --- |
| & | Element-wise Logical AND operator. It returns TRUE if both elements are TRUE |
| && | Logical AND operator - Returns TRUE if both statements are TRUE |
| | | Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE |
| || | Logical OR operator. It returns TRUE if one of the statement is TRUE. |
| ! | Logical NOT - returns FALSE if statement is TRUE |

## **R Miscellaneous Operators**

Miscellaneous operators are used to manipulate data:

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| : | Creates a series of numbers in a sequence | x <- 1:10 |
| %in% | Find out if an element belongs to a vector | x %in% y |
| %\*% | Matrix Multiplication | x <- Matrix1 %\*% Matrix2 |

**Q) Special numbers in R**

Like most programming languages, R has a number of Special values that are exceptions to the normal values of a type. These are NA, NULL, ±Inf and NaN.

Inf is infinity. You can have positive or negative infinity.

1/0

# [1] Inf

1/Inf

# [1] 0

NaN means Not a number. it's an undefined value.

0/0

NaN.

**Q) Logical values in R**

A logical value is often created by doing comparison between variables.

> x = 1; y = 2 # sample values

> z = x > y # is x larger than y?

> z # print the logical value

[1] FALSE

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

[1] "logical"

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

> u = TRUE; v = FALSE

> u & v # u AND v

[1] FALSE

> u | v # u OR v

[1] TRUE

> !u # negation of u

[1] FALSE

Further details and related logical operations can be found in the R documentation.

> help("&")

**Q) help function in R**

The help() function and ? help operators in R provide access to the documentation pages for R functions, data sets, and other objects.

To read more about a given function, for example mean, the R function help() can be used as follow:

**help(mean)**

(or **)**

**?mean**

To access help for a function in a package that’s not currently loaded, specify in addition the name of the package: For example, to obtain documentation for the rlm() (robust linear model) function in the MASS package, help(rlm, package="MASS").

Examples

* help(): this provides help on using the help() function itself
* help(mean): this returns help about the arithmetic mean() function.
* help(package=base): this returns information about the base package.
* ?mean: this is the same as help(mean)
* ?'while' or ?'\*': these returns information on control structures or unary operators (enclose these in single quotes)

The Example() Function

R provides the example() function too. If you want to see an example of a command in use, this will help you.

Example

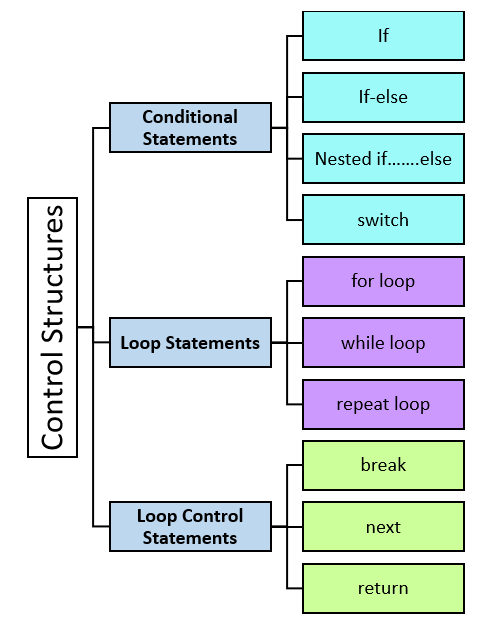
* example(mean): this provides example code so you can see how to use the arithmetic mean function
* example('for'): this provides examples so you can see how to use ''for'' loops

**Q) R Data Structures**

<https://www.geeksforgeeks.org/data-structures-in-r-programming/>

Refer notes

**Q) Control Structures in R**

****

**2. ifelse() Function**

The ifelse() function acts like the if-else structure. The following is the syntax of the ifelse() function in R:

ifelse(**condition**, *exp\_if\_true*, *exp\_if\_false*)

Where condition is the condition that evaluates to either TRUE or FALSE,

exp\_if\_true is the expression that is returned if the condition results in TRUE,

exp\_if\_false is the expression that is returned if the condition results in FALSE.

Example: ifelse(a<7,"a is less than 7","a is greater than 7")

3. Switch Statement in R

The R programming language using switch statements has m number of options to choose and can execute any one of the options (cases)at a given time.

When the condition matches with the corresponding case then that case will only execute.

Syntax

**switch(condition,case1 = “”,case2= “”,………., Default statement)**

Where

* condition is the expression to be evaluated.
* ……… is the alternative list such as case1,case2, etc.
* Default statement is executed when the match case to the condition is missing
* The first argument is always the condition or expression to evaluate.

Consider an example

a <- switch(

4, #condition

#(...... or case1,case2..)

"red", #case1

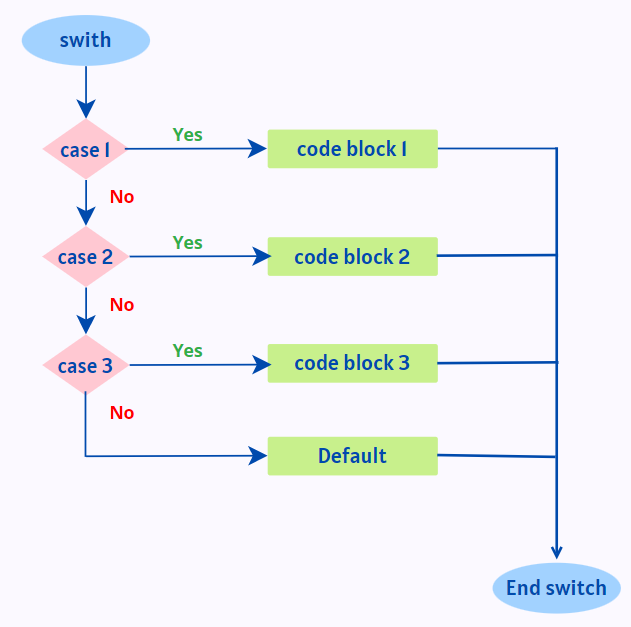
"blue", #case2

"pink", #case3

"orange" #case4

)

print(a)



<https://learnetutorials.com/r-programming/conditional-statements>

There are three types of loop in R programming:

1. for
2. while
3. repeat

**1. for loop**

Syntax:

for (value in sequence)

{

statement

}

Flow Chart:

Diagram

Description automatically generated

Program to print first five natural numbers:

for (i in 1:5)

{

    # statement

    print(i)

}

Output:

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

**Program to display days of week using for loop**

week = c('Sunday', 'Monday','Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')

for (day in week)

{

print(day)

}

Output:

[1] "Sunday"

[1] "Monday"

[1] "Tuesday"

[1] "Wednesday"

[1] "Thursday"

[1] "Friday"

[1] "Saturday"

**While loop**

Syntax:

while( condition)

{

statement

}

Flow chart:

Diagram

Description automatically generated

Program to calculate factorial of 5.

n=5

factorial = 1

i = 1

while (i <= n)

{

factorial = factorial \* i

i = i + 1

}

print(factorial)

Output:

[1] 120

**Repeat Loop : (just like infinite loop in C programming)**

Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop’s body and use the declaration of a break statement to terminate this loop. If no condition is present in the body of the repeat loop then it will iterate infinitely.

Syntax:

repeat

{

statement

if( condition )

{

break

}

}

Flow chart:

Diagram

Description automatically generated

Program:

Program to display numbers from 1 to 5 using repeat loop in R.

val = 1

repeat

{

print(val)

val = val + 1

if(val > 5)

{

break

}

}

Output:

[1] 1

[1] 2

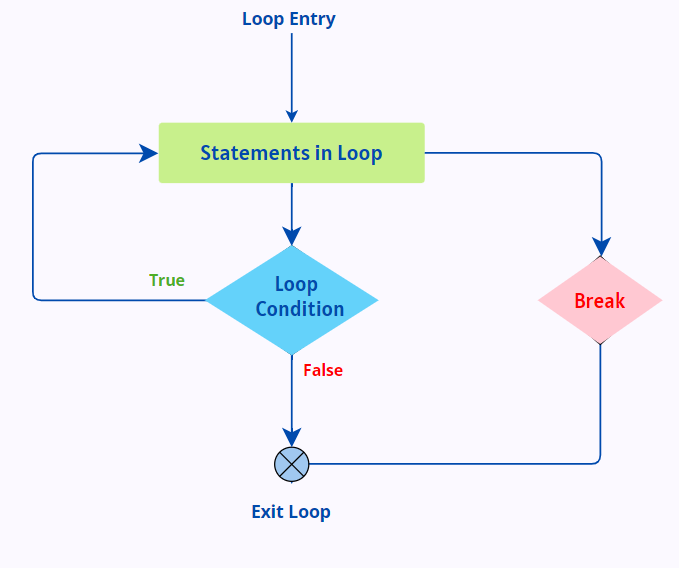
[1] 3

[1] 4

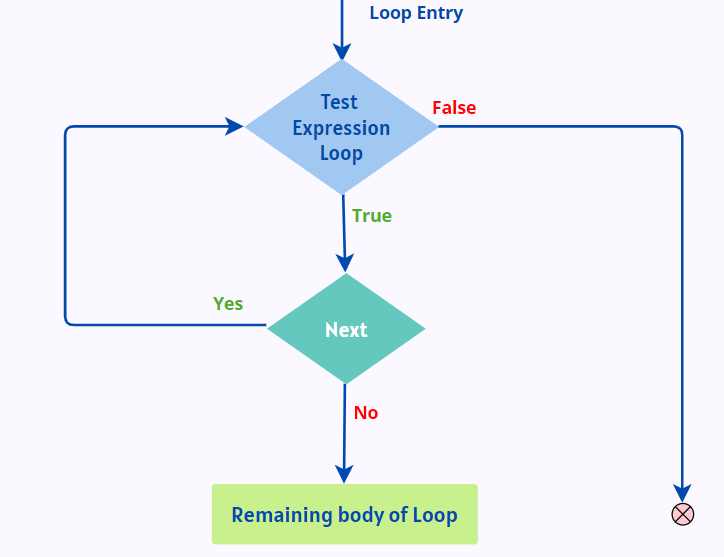
[1] 5

**Break statement**

The break statement in a loop allows the sudden termination from the current iteration which interrupts the further execution of the iteration. The control gets transferred to immediate instruction after the loop



**Next statement in R** is used to omit the current iteration and jump to the next iteration cycle without terminating a loop. (just like continue in C programming)



Example:

v < -c("Learn", "eTutorials", "for", "R", "Program")

for (i in 1: 5)

{

if (i == 3)

{

next

}

print(v[i])

}

The output generated is

[1] "Learn"

[1] "eTutorials"

[1] "R"

[1] "Program"

**return statement in R**

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

Syntax:return(expression)

Example:

check <- function(x)

{

if (x > 0)

result <- "Positive"

else if (x < 0)

result <- "Negative"

else

result <- "Zero"

return(result)

}

In the console window, we type:

> check(1)

> check(-10)

> check(0)

Output:

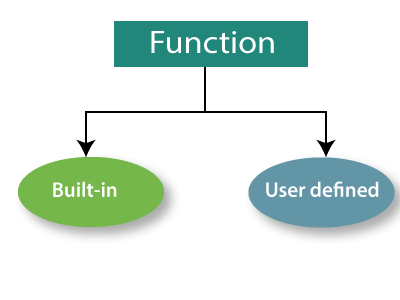
Positive

Negative

Zero

**Q)Basic Functions in R**

Functions are set of statements organized to perform a particular task. Functions are of 2 types



Some statistical inbuilt functions already written and available to us are:

mean, median, quantiles, minimum, maximum, variance, and standard deviation.

1. mean(data)
2. median(data)
3. sd(data)
4. var(data)
5. max(data)
6. min(data)
7. quantile(data)
8. summary(data)

the summary function will provide the output based on the type of the dataset.

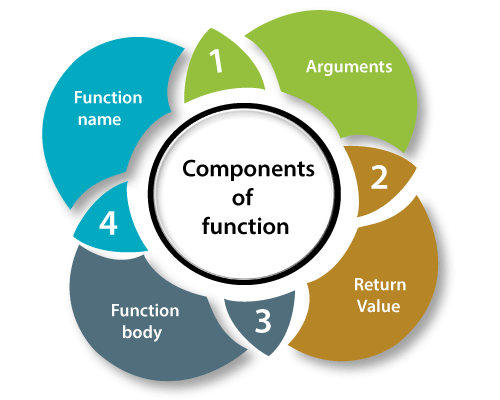
User defined functions: are the functions written by the programmer.

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



Components of Functions

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



Example:

check <- function(x)

{

if (x > 0)

result <- "Positive"

else if (x < 0)

result <- "Negative"

else

result <- "Zero"

return(result)

}

Here check is the name of the function.

x is the argument

return value is the value stored in result variable

All the statements inside the { } are the function body.

To call the function, check

check(10);

We get an output

Positive

**Q) Vectors: Definition**

Vector is a basic data structure in R. It contains element of the same type. The data types can be logical, integer, double, character, complex or raw.

A vector’s type can be checked with the typeof() function.

Another important property of a vector is its length. This is the number of elements in the vector and can be checked with the function length().

**Q) Declaration and Generating of vectors**

Vectors are generally created using the c() function.

Since, a vector must have elements of the same type, this function will try and coerce elements to the same type, if they are different.

Coercion is from lower to higher types from logical to integer to double to character.

Example1:

> x <- c(1, 5, 4, 9, 0)

> typeof(x)

Output:

[1] "double"

Example2:

> x <- c(1, 5.4, TRUE, "hello")

> x

Output:

[1] "1" "5.4" "TRUE" "hello"

> typeof(x)

Output:

[1] "character"

Creating a vector using : operator

If we want to create a vector of consecutive numbers, the : operator is very helpful.

Example 1:

> x <- 1:7;

print(x)

Output:

[1] 1 2 3 4 5 6 7

> y <- 2:-2; y

[1] 2 1 0 -1 -2

Creating vectors using seq() function: More complex sequences can be created using the seq() function, like defining number of points in an interval, or the step size.

Example 1: Creating a vector using seq() function

> seq(1, 3, by=0.2) # specify step size

[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0

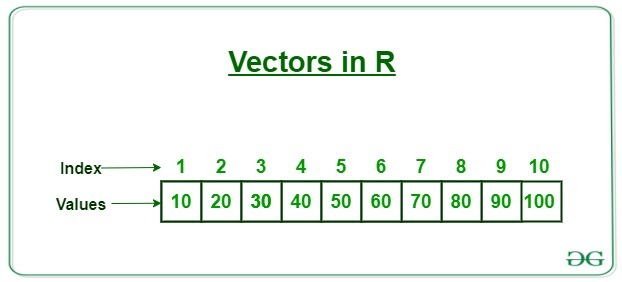
Example2:

> seq(1, 5, length.out=4) # specify length of the vector

[1] 1.000000 2.333333 3.666667 5.000000

**Q) Indexing of vectors -**

Vectors in R are the same as the arrays in C language which are used to hold multiple data values of the same type. One major key point is that in R the indexing of the vector will start from ‘1’ and not from ‘0’.



**Q) How to access Elements of a Vector?**

Elements of a vector can be accessed using vector indexing. The vector used for indexing can be logical, integer or character vector.

We can use a vector of integers as index to access specific elements.

We can also use negative integers to return all elements except that those specified.

But we cannot mix positive and negative integers while indexing and real numbers, if used, are truncated to integers.

> x

[1] 0 2 4 6 8 10

> x[3] # access 3rd element

[1] 4

> x[c(2, 4)] # access 2nd and 4th element

[1] 2 6

> x[-1] # access all but 1st element

[1] 2 4 6 8 10

> x[c(2, -4)] # cannot mix positive and negative integers

Error in x[c(2, -4)] : only 0's may be mixed with negative subscripts

> x[c(2.4, 3.54)] # real numbers are truncated to integers

[1] 2 4

**Naming vectors**

As a data analyst, it is important to have a clear view on the data that you are using. Understanding what each element refers to is therefore essential.

We can assign names to vector members. For example, the following variable v is a character string vector with two members.

> v = c("Mary", "Sue")

> v

[1] "Mary" "Sue"

We now name the first member as First, and the second as Last.

> names(v) = c("First", "Last")

> v

First Last

"Mary" "Sue"

Then we can retrieve the first member by its name.

> v["First"]

[1] "Mary"

**Q) Adding elements to vectors**

The size of a vector is determined at its creation, so if you wish to add or delete elements, you’ll need to reassign the vector.

For example, let’s add an element to the middle of a four-element vector:

> x <- c(64,5,12,45)

> x <- c(x[1:2],75,x[3],x[4]) # insert 75 before the 12

> x

[1] 64 5 75 12 45

Here, we created a four-element vector and assigned it to x.

To insert a new number 75 between the second and third elements, we string together the first two elements of x, then the 75, then the fourth and fifth elements of x with concatenation function c().

This creates a new five-element vector. We then assigned that new vector to x.

append() method in R programming is used to append the different types of integer values into a vector in the last.

Syntax: append(x, value, index(optional))

Return: Returns the new vector after appending given value.

Example:

data <- 4:8

data <- append(data, 11)

print(data)

[1] 4 5 6 7 8 11

**Q) Deleting elements from vectors**

we will just re-assign the vector to delete and update the vector elements.

> x <- c(64,75,12,45)

> x<- c(75,12)

> x

[1] 75 12

> x<- c(x[3:5])

> x

[1] 75 12 45.

>x <- c( 12 5 13 20 16 8 20)

> x <- x[-2:-4] # delete elements 2 through 4

**Q)Operations on Vectors**

#### **The Vector**

**> v <- c(3,2,2)**

**> v**

[1] 3 2 2

#### **Is Something A Vector**

**> is.vector(v)**

[1] TRUE

**> is.matrix(v)**

[1] FALSE

#### **"Transpose" of a Vector**

**> vp <- t(v)**

**> vp**

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

[1,] 3 2 2

#### **Adding Vectors**

**> w <- c(4,1,2)**

**> s <- v + w**

**> s**

[1] 7 3 4

#### **Subtracting Vectors**

**> d <- v - w**

**> d**

[1] -1 1 0

#### **Multiplication by a Scalar**

**> s <- 3\*v**

**> s**

[1] 9 6 6

**> t <- v\*3**

**> t**

[1] 9 6 6

#### **Vector Dot Product**

**> d <- v %\*% w**

**> d**

[,1]

[1,] 18

**> d <- w %\*% v**

**> d**

[,1]

[1,] 18

**> d <- v %\*% v**

**> d**

[,1]

[1,] 17

**> d <- w %\*% w**

**> d**

[,1]

[1,] 21

Finding length of vector

x <- c(2, 4, 6, 8)

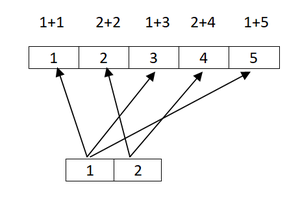
len= length(x)

print(len)

Output: 4

**Q)Vector recycling in R**

We can see vector recycling, when we perform some kind of operations like addition, subtraction. . . .etc on two vectors of unequal length. The vector with a small length will be repeated as long as the operation completes on the longer vector. If we perform an addition operation on a vector of equal length the first value of vector1 is added with the first value of vector 2 like that. The below image demonstrated operation on unequal vectors and operation on equal vector.

****

So, the repetition of small length vector as long as completion of operation on long length vector is known as vector recycling.

# creating vector with 1 to 6 values

vec1=1:6

# creating vector with 1:2# values

vec2=1:2

# adding vector1 and vector2

print(vec1+vec2)

Output:

2 4 4 6 6 8

Here the longer object length (6) is multiple of the shortest object length(2). So, we didn’t get warning message. **If longer object length is not multiple of the shortest object length, we get a warning message.**

**Q) A Vectorized if-then-else: The ifelse() Function**

In addition to the usual if-then-else construct found in most languages, R also includes a vectorized version, the ifelse() function. The form is as follows:

ifelse(**condition**, *exp\_if\_true*, *exp\_if\_false*)

Where condition is the condition that evaluates to either TRUE or FALSE,

exp\_if\_true is the expression that is returned if the condition results in TRUE,

exp\_if\_false is the expression that is returned if the condition results in FALSE.

**The return value is itself a vector;**

> x <- 1:10

> y <- ifelse(x %% 2 == 0,5,12) # %% is the mod operator

> y

[1] 12 5 12 5 12 5 12 5 12 5

Here, we wish to produce a vector in which there is a 5 wherever x is even or a 12 wherever x is odd.

Example: ifelse(b,u,v)

u[i] if b[i] is true, or v[i] if b[i] is false

.

**Q) Vector Equality**

Suppose we wish to test whether two vectors are equal.

using ==, won’t work. == does element wise comparison

> x <- 1:3

> y <- c(1,3,4)

> x == y

[1] TRUE FALSE FALSE

In fact, **== is a vectorized function.**

The expression x == y applies the function ==() to the elements of x and y. yielding a vector of Boolean values.

To test whether two vectors are equal using all

> all(x == y)

[1] FALSE

Applying all() to the result of == asks whether all of the elements of the latter are true, which is the same as asking whether x and y are equal.

To test whether two vectors are equal using identical

> identical(x,y)

[1] FALSE

Be careful, though because the word identical really means what it says.

Consider this little R session:

> x <- 1:2

> y <- c(1,2)

> x

[1] 1 2

> y

[1] 1 2

> identical(x,y)

[1] FALSE

> typeof(x)

[1] "integer"

> typeof(y)

[1] "double"

So, : produces integers while c() produces floating-point numbers.

Who knew?

**Q) Missing values -NA and NULL Values**

**missing data** in R is represented with the value NA.

NULL, on the other hand, represents that the value *doesn’t exist or unknown*.

Using NA

In many of R’s statistical functions, we can instruct the function to skip over

any missing values, or NAs. Here is an example:

> x <- c(88,NA,12,168,13)

> x

[1] 88 NA 12 168 13

> mean(x)

[1] NA

> mean(x,na.rm=T)

[1] 70.25

In the first call, mean() refused to calculate, as one value in x was NA. But

by setting the optional argument na.rm (NA remove) to true (T), we calculated

the mean of the remaining elements.

> x <- c(88,NULL,12,168,13)

> mean(x)

[1] 70.25

But R automatically skipped over the NULL value

There are multiple NA values, one for each mode:

> x <- c(5,NA,12)

> mode(x[1])

[1] "numeric"

> mode(x[2])

[1] "numeric"

> y <- c("abc","def",NA)

> mode(y[2])

[1] "character"

> mode(y[3])

[1] "character"

Using NULL

One use of NULL is to build up vectors in loops, in which each iteration adds another element to the vector. In this simple example, we build up a vector of even numbers:

# build up a vector of the even numbers in 1:10

> z <- NULL

> for (i in 1:10) if (i %%2 == 0) z <- c(z,i)

> z

[1] 2 4 6 8 10

%% is the modulo operator, giving remainders upon division.

For example, 13 %% 4 is 1, as the remainder of dividing 13 by 4 is 1.

Thus the example loop starts with a NULL vector and then adds the element 2 to it,

then 4, and so on.

The difference between NA and NULL.

If we were to use NA instead of NULL in the preceding example, we

would pick up an unwanted NA:

> z <- NA

> for (i in 1:10) if (i %%2 == 0) z <- c(z,i)

> z

[1] NA 2 4 6 8 10

NULL values really are counted as nonexistent, as you can see here:

> u <- NULL

> length(u)

[1] 0

> v <- NA

> length(v)

[1] 1

NULL is a special R object with no mode.

**Q) Filtering**

This allows us to extract a vector’s elements that satisfy certain conditions.

Example 1 : to extract from z all its elements whose squares were greater than 8 and then assign to w.

> z <- c(5,2,-3,8)

> w <- z[z\*z > 8]

> w

[1] 5 -3 8

Explanation

> z <- c(5,2,-3,8)

> z\*z > 8

[1] TRUE FALSE TRUE TRUE

Evaluation of the expression z\*z > 8 gives us a vector of Boolean values!

Then the resulting Boolean values are used to cull out the desired elements of z:

> z[c(TRUE,FALSE,TRUE,TRUE)]

[1] 5 -3 8

Example 2: we will use the results to extract from another vector, y, instead of extracting from z:

> z <- c(5,2,-3,8)

> j <- z\*z > 8

> j

[1] TRUE FALSE TRUE TRUE

> y <- c(1,2,30,5)

> y[j]

[1] 1 30 5

Example 3:To have a vector x in which we wish to replace all elements larger than a 3 with a 0.

> x[x > 3] <- 0

Let’s check:

> x <- c(1,3,8,2,20)

> x[x > 3] <- 0

> x

[1] 1 3 0 2 0

Filtering with the subset() Function

Filtering can also be done with the subset() function. When applied to vectors, the difference between using this function and ordinary filtering lies in the manner in which NA values are handled.

> x <- c(6,1:3,NA,12)

> x

[1] 6 1 2 3 NA 12

> x[x > 5]

[1] 6 NA 12

> subset(x,x > 5)

[1] 6 12

When we did ordinary filtering in the previous section, R basically said,“Well, x[5] is unknown, so it’s also unknown whether its square is greater than 5.” But you may not want NAs in your results. When you wish to exclude NA values, using subset() saves you the trouble of removing the NA values yourself.

The Selection Function which()

As you’ve seen, filtering consists of **extracting elements of a vector z that satisfy a certain condition**. In some cases, though, we may just want **to find the positions within z at which the condition occurs**. We can do this using which(), as follows:

> z <- c(5,2,-3,8)

> which(z\*z > 8)

[1] 1 3 4

The result says that elements 1, 3, and 4 of z have squares greater than 8.

which() is also used for determining the location within a vector at which the first occurrence of some condition holds.

For example, to find the index of first 1 in x

first1 <- function(x)

{

for (i in 1:length(x))

{

if (x[i] == 1) break # break out of loop

}

return(i)

}

Here is an alternative way of coding this task:

first1a <- function(x) return(which(x == 1)[1])

The call to which() yields the indices of the 1s in x. These indices will be given in the form of a vector, and we ask for element index 1 in that vector, which is the index of the first 1.

it’s wasteful, as it actually finds all instances of 1s in x, when we need only the first.

So, although it is a vectorized approach and thus possibly faster, if the first 1 comes early in

x, this approach may actually be slower.

Recycling The automatic lengthening of vectors in certain settings

Filtering The extraction of subsets of vectors

Vectorization Where functions are applied element-wise to vectors

Examine your vector

1. **typeof(z)**
2. **length(z)**
3. **class(z)**
4. **str(z)**

<https://data-flair.training/blogs/r-vector-functions/>

**Unit-3: syllabus**

Matrices - Creating Matrices - Adding or Removing rows/columns - Reshaping - Operations -

Special functions on Matrices. Lists - Creating List – General List Operations - Special

Functions - Recursive Lists. Data Frames - Creating Data Frames - Naming - Accessing - Adding- Removing - Applying Special functions to Data Frames - Merging Data Frames- Factors and Tables.

**Creating matrices**

Matrix row and column subscripts begin with 1. For example, the upper-left corner of the matrix a is denoted a[1,1]. The internal storage of a matrix is in column-major order, meaning that first all of column 1 is stored, then all of column 2, and so on,

One way to create a matrix is by using the matrix() function:

> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

As there were four elements,we can specify only nrows;

As shown below

> y <- matrix(c(1,2,3,4),nrow=2)

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

[,2] means the entirety of column 2, as can be seenin this check:

> y[,2]

[1] 3 4

Another way to build y is to specify elements individually:

> y <- matrix(nrow=2,ncol=2)

> y[1,1] <- 1

> y[2,1] <- 2

> y[1,2] <- 3

> y[2,2] <- 4

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

you can set the byrow argument in matrix() to true to indicate that the data is coming

in row-major order. Here’s an example of using byrow:

> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)

> m

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

[1,] 1 2 3

[2,] 4 5 6

**Adding and Deleting Matrix Rows and Columns**

Matrices are of fixed length and dimensions, so we cannot add or delete rows or columns. However, matrices can be reassigned. Memory allocation each time.

Changing the Size of a Matrix

Using the rbind() (row bind) and cbind() (column bind) functions

Consider two 2x2 matrices one and z as follows:

one = matrix(c(1,1,1,1),ncol=2,nrow=2)

one

[,1] [,2]

[1,] 1 1

[2,] 1 1

z = matrix(ncol =2,nrow=2)

z[,1]=c(1,2)

z[,2]=c(3,4)

z

[,1] [,2]

[1,] 1 3

[2,] 2 4

cbind(one,z)

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

[1,] 1 1 1 3

[2,] 1 1 2 4

Here, cbind() creates a new matrix by combining columns of 1s with the columns of z.

Note, too, that we could have relied on recycling:

> cbind(1,z)

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

[1,] 1 1 3

[2,] 1 2 4

Here, the 1 value was recycled into a vector of two 1 values.

You can also use the rbind() and cbind() functions as a quick way to create small matrices. Here’s an example:

> q <- cbind(c(1,2),c(3,4))

> q

[,1] [,2]

[1,] 1 3

[2,] 2 4

You can delete rows or columns by reassignment:

> m <- matrix(1:6,nrow=3)

> m

[,1] [,2]

[1,] 1 4

[2,] 2 5

[3,] 3 6

To delete 2nd and 3rd row

> m <- m[1,]

> m

[1] 1 4

**Naming Matrix Rows and Columns**

The natural way to refer to rows and columns in a matrix is via the row and column numbers. However, you can also give names to these entities. Here’san example:

z = matrix(c(1,2,3,4),ncol=2,nrow=2)

z

[,1] [,2]

[1,] 1 3

[2,] 2 4

> colnames(z)

NULL

> colnames(z) <- c("a","b")

>z

a b

[1,] 1 3

[2,] 2 4

**Applying Functions to Matrix Rows and Columns**

One of the most famous and most used features of R is the \*apply() family of functions, such as apply(), tapply(), and lapply().

Using the apply() Function

This is the general form of apply for matrices:

apply(m,dimcode,f,fargs)

where the arguments are as follows:

• m is the matrix.

• dimcode is the dimension, equal to 1 if the function applies to rows or 2 for columns.

• f is the function to be applied.

• fargs is an optional set of arguments to be supplied to f.

For example, here we apply the R function mean() to each column of a matrix z:

>z = matrix(c(1,2,3,4),ncol=2,nrow=2)

>z

[,1] [,2]

[1,] 1 3

[2,] 2 4

> apply(z,2,mean)

[1] 1.5 3.5

Here’s an example using our own function f:

> f <- function(x) x/c(2,8)

> y <- apply(z,1,f)

> y

[,1] [,2]

[1,] 0.500 1.0

[2,] 0.375 0.5

Our f() function divides a two-element vector by the vector (2,8). (1,3) is first row and is passed to f() first,. So, x is (1,3). The value of (1,3)/(2,8), which in R’s element-wise vector arithmetic is (0.5,0.375).

That first computation, (0.5,0.375), ends up at the first column in the output of apply(), not the first row. But this is the behavior of apply().

If the function to be applied returns a vector of k components, then the result of apply() will have k rows.

You can use the matrix transpose function t() to change it if necessary, as follows:

> t(apply(z,1,f))

[,1] [,2]

[1,] 0.5 0.375

[2,] 1.0 0.500

**General Matrix Operations**

These include performing linear algebra operations, matrix indexing, and matrix filtering.

Performing Linear Algebra Operations on Matrices

You can perform various linear algebra operations on matrices, such as

* matrix multiplication,
* matrix scalar multiplication,
* matrix addition.

>y

[,1] [,2]

[1,] 1 3

[2,] 2 4

>y%\*% y # mathematical matrix multiplication

[,1] [,2]

[1,] 7 15

[2,] 10 22

> 3\*y # mathematical multiplication of matrix by scalar

[,1] [,2]

[1,] 3 9

[2,] 6 12

> y+y # mathematical matrix addition

[,1] [,2]

[1,] 2 6

[2,] 4 8

Matrix Indexing

>z=matrix(c(1,2,3,4,1,1,0,0,1,0,1,0),ncol=3)

>z

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

[1,] 1 1 1

[2,] 2 1 0

[3,] 3 0 1

[4,] 4 0 0

> z[,2:3]

[,1] [,2]

[1,] 1 1

[2,] 1 0

[3,] 0 1

[4,] 0 0

Here, we requested the submatrix of z consisting of all elements with column numbers 2 and 3 and any row number. This extracts the second and third columns.

Here’s an example of extracting rows instead of columns:

> z[2:3,]

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

[1,] 2 1 0

[2,] 3 0 1

> z[2:3,1]

[1] 2 3

You can also assign values to submatrices:

> x <- matrix(nrow=3,ncol=3)

> y <- matrix(c(4,5,2,3),nrow=2)

> y

[,1] [,2]

[1,] 4 2

[2,] 5 3

> x[2:3,2:3] <- y

> x

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

[1,] NA NA NA

[2,] NA 4 2

[3,] NA 5 3

To delete second row

> y

[,1] [,2]

[1,] 1 4

[2,] 2 5

[3,] 3 6

> y[-2,]

[,1] [,2]

[1,] 1 4

[2,] 3 6

Filtering on Matrices

> x

x

[,1] [,2]

[1,] 1 2

[2,] 2 3

[3,] 3 4

> x[x[,2] >= 3,]

x

[,1] [,2]

[1,] 2 3

[2,] 3 4

• The object x[,2] is a vector.

• The operator >= compares two vectors.

• The number 3 was recycled to a vector of 3s.

*Explanation:*

> j <- x[,2] >= 3

> j

[1] FALSE TRUE TRUE

Here, we look at the vector x[,2], which is the second column of x, and determine which of its elements are greater than or equal to 3. The result,assigned to j, is a Boolean vector.

Now, use j in x:

> x[j,]

>x

[,1] [,2]

[1,] 2 3

[2,] 3 4

Here, we compute x[j,]—that is, the rows of x specified by the true elements of j—getting the rows corresponding to the elements in column 2 that were at least equal to 3.

Example 2:

> z <- c(5,12,13)

> x[z %% 2 == 1,]

[,1] [,2]

[1,] 1 4

[2,] 3 6

Here, the expression z %% 2 == 1 tests each element of z for being an odd number, thus yielding (TRUE,FALSE,TRUE). As a result, we extracted the first and third rows of x.

*which()*

Here’s an example:

>m=matrix(c(5,2,9,-1,10,11),ncol=2)

>print(m)

[,1] [,2]

[1,] 5 -1

[2,] 2 10

[3,] 9 11

> which(m > 2)

[1] 1 3 5 6

elements 1, 3, 5, and 6 of m are larger than 2. For example, element 5 is the element in

row 2, column 2 of m.

**Creating Lists**

Lists are referred to as recursive vectors.

Let’s consider an employee database. For each employee, we wish to store the name, salary, and a Boolean indicating union membership. Since we have three different modes here—character, numeric, and logical—it’s a perfect place for using lists. Our entire database might then be a list of lists.

For example: To Create a list to represent our employee, Joe

j <- list(name="Joe", salary=55000, union=T)

We could print out j, either in full or by component:

> j

$name

[1] "Joe"

$salary

[1] 55000

$union

[1] TRUE

Actually, the component names—called tags in the R literature—such as name,salary,union are optional. We could also create lists without tags. Numeric indices are created as shown below:

> jalt <- list("Joe", 55000, T)

> jalt

[[1]]

[1] "Joe"

[[2]]

[1] 55000

[[3]]

[1] TRUE

Since lists are vectors, they can be created via vector():

> z <- vector(mode="list")

> z[["abc"]] <- 3

> z

$abc

[1] 3

**General List Operations**

1. List Indexing
2. Adding and Deleting List Elements
3. Getting the Size of a List

1. List Indexing

List components can be accessed in several different ways:

j <- list(name="Joe", salary=55000, union=T)

there are three ways to access an individual component ***salary*** of a list ***j*** and return it in the data type of ***salary***

> j$salary

[1] 55000

> j[["salary"]]

[1] 55000

> j[[2]]

[1] 55000

If single brackets [ ] are used, the result is another list—a sublist of the original.

Double brackets [[ ]] are used to access a single component, with the result having the type of that component.

For example, we have this:

>j <- list(name="Joe", salary=55000, union=T)

>j1=j[[2]]

>typeof(j1)

[1] "double"

j2=j[2]

typeof(j2)

[1] "list"

2. Adding and Deleting List Elements

List uses linked list data structure.

New components can be added after a list is created.

> z <- list(a="abc",b=12)

Adding new component c

> z$c <- "sailing" # add a c component

Adding components using vector index:

> z[[4]] <- 28

> z[5:7] <- c(FALSE,TRUE,TRUE)

> print(z)

$a

[1] "abc"

$b

[1] 12

$c

[1] "sailing"

[[4]]

[1] 28

[[5]]

[1] FALSE

[[6]]

[1] TRUE

[[7]]

[1] TRUE

Delete a list component by setting it to NULL.

>z[1:2]<- NULL

>z[5:7]<- NULL

>print(z)

$c

[1] "sailing"

[[2]]

[1] 28

[[3]]

[1] FALSE

[[4]]

[1] TRUE

Note that upon deleting z$a, the indices of the elements after it moved

up by 1.

Concatenate lists.

> c(list("Joe", 55000, T),list(5))

[[1]]

[1] "Joe"

[[2]]

[1] 55000

[[3]]

[1] TRUE

[[4]]

[1] 5

3. Getting the Size of a List

Since a list is a vector, you can obtain the number of components in a list via

length().

> length(j)

[1] 3

**Q) Accessing List Components and Values**

To obtain the tags using names():

> names(j)

[1] "name" "salary" "union"

Converting a list to vector using unlist() function In order to perform arithmetic operations, lists should be converted to vectors using unlist() function.

Example1:

> z <- list(a=5,b=12,c=13)

> y <- unlist(z)

> class(y)

[1] "numeric"

> y

a b c

5 12 13

So the output of unlist() in this case was a numeric vector.

Example2:

> w <- list(a=5,b="xyz")

> wu <- unlist(w)

> class(wu)

[1] "character"

> wu

a b

"5" "xyz"

Where possible the list components are coerced to a common mode during the unlisting, and so the result often ends up as a character vector. Vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer< real < complex < character < list < expression: pairlists are treated as lists.

But there is something else to deal with here. Though wu is a vector and not a list, R did give each of the elements a name. We can remove them by setting their names to NULL

> names(wu) <- NULL

> wu

[1] "5" "xyz"

We can also remove the elements’ names directly with unname(), as follows:

> wun <- unname(wu)

> wun

[1] "5" "xyz"

**Q) Applying Functions to Lists**

Two functions are handy for applying functions to lists: lapply and sapply.

The function lapply() (for list apply) works like the matrix apply() function,calling the specified function on each component of a list (or vector coercedto a list) and returning another list. Here’s an example:

> lapply(list(1:3,25:29),median)

[[1]]

[1] 2

[[2]]

[1] 27

R applied median() to 1:3 and to 25:29, returning a list consisting of 2 and 27.

In some cases, such as the example here, the list returned by lapply() could be simplified to a vector or matrix. This is exactly what sapply()

> sapply(list(1:3,25:29),median)

[1] 2 27

Sorting list by names

lis = list(be=67,and=25,but=32)

sortlist <- function(lis)

{

nms <- names(lis) # the words

sn <- sort(nms) # same words in alpha order

return(lis[sn]) # return rearranged version

}

lis= sortlist(lis)

print(lis)

$and

[1] 25

$be

[1] 67

$but

[1] 32

Since our words are the names of the list components, we can extract the words by simply calling names(). We sort these alphabetically, we use the sorted version as input to list indexing, giving us a sorted version of the list.

We can sort by word frequency in a similar manner.

lis = list(be=c(67,68,89),and=c(25,67),but=32)

The output of sapply(lis,length)

be and but

3 2 1

sortlist <- function(lis)

{

nms <- names(lis) # the words

sn <- sapply(lis,length) # get word frequencies

return(lis[order(sn)]) # return rearranged version

}

lis=sortlist(lis)

lis

$but

[1] 32

$and

[1] 25 67

$be

[1] 67 68 89

**Recursive Lists**

Lists can be recursive, meaning that you can have lists within lists. Here’s an

example:

> b <- list(u = 5, v = 12)

> c <- list(w = 13)

> a <- list(b,c)

> a

[[1]]

[[1]]$u

[1] 5

[[1]]$v

[1] 12

[[2]]

[[2]]$w

[1] 13

> length(a)

[1] 2

This code makes a into a two-component list, with each component itself

also being a list.

The concatenate function c() has an optional argument recursive, which

controls whether flattening occurs when recursive lists are combined.

> c(list(a=1,b=2,c=list(d=5,e=9)))

$a

[1] 1

$b

[1] 2

$c

$c$d

[1] 5

$c$e

[1] 9

> c(list(a=1,b=2,c=list(d=5,e=9)),recursive=T)

a b c.d c.e

1 2 5 9

In the first case, we accepted the default value of recursive, which is

FALSE, and obtained a recursive list, with the c component of the main list

itself being another list. In the second call, with recursive set to TRUE, we got

a single list as a result; only the names look recursive.

**Q) Creating Data Frames**

A data frame is a special type of list where every element of a list has the same length.

> lang <- c("Java","C")

> inventor <- c(“james”,”dennis”)

> d <- data.frame(lang,inventor,stringsAsFactors=FALSE)

> d # matrix-like viewpoint

lang inventor

1 Java james

2 C dennis

If the named argument stringsAsFactors is not specified, then by default,stringsAsFactors will be TRUE. This means that if we create a data frame from a character vector—in this case, kids—R will convert that vector to a factor.

Accessing Data Frames

Since d is a list, we can access it via component index values or component names:

**First column data can be accessed using d[[1]] (or) d$lang (or) d[,1]**

> d[[1]]

[1] "Java" "C"

> d$lang

[1] "Java" "C"

But we can treat it in a matrix-like fashion as well. For example, we can view column 1:

> d[,1]

[1] "Java" "C"

This matrix-like quality is also seen when we take d apart using str():

> str(d)

'data.frame': 2 obs. of 2 variables:

$ lang : chr "Java" "C"

$ inventor: chr "james" "dennis"

R tells us here that d consists of two observations—our two rows—that store data on two variables—our two columns.

**Q) Other Matrix-Like Operations**

Various matrix operations also apply to data frames.

*Extracting Subdata Frames* we can extract sub data frames by rows or columns. Here’s an

example:

> examsquiz = data.frame(id=c(1,2,3,4,5), name = c("sita","ram","laxman","hanuman","bharat"), per=c(89,90,84,95,99))

>examsquiz

id name per

1 1 sita 89

2 2 ram 90

3 3 laxman 84

4 4 hanuman 95

5 5 bharat 99

To display 2nd row to 5th row

> examsquiz[2:5,]

To display data type of 3rd column

> class(examsquiz[,3])

[1] "numeric"

> class(examsquiz[2:5,3,drop=FALSE])

[1] "data.frame"

Since examsquiz[2:5,3] is a vector, R created a vector instead of another data frame. By specifying drop=FALSE, we can keep it as a (one-column) data frame.

Filtering

To display the data of the students whose percentage is >=90

> examsquiz[examsquiz$per >= 90,]

id name per

2 2 ram 90

4 4 hanuman 95

5 5 bharat 99

More on Treatment of NA Values

To calculate mean without considering NA values, we would write

>mean(examsquiz$per, na.rm=TRUE)

If you use the subset() function, you need not write na.rm=TRUE.

In our example, instead of typing this:

> examsquiz[examsquiz$per >= 90,]

we could run this:

> subset(examsquiz,per >= 90)

In some cases, we may wish to rid our data frame of any observation that has at least one NA value. A handy function for this purpose is

complete.cases().

> d4

kids states

1 Jack CA

2 <NA> MA

3 Jillian MA

4 John <NA>

> complete.cases(d4)

[1] TRUE FALSE TRUE FALSE

***To take the rows that has no NA values***

> d5 <- d4[complete.cases(d4),]

> d5

kids states

1 Jack CA

3 Jillian MA

Using the rbind() and cbind() Functions and Alternatives

cbind() can be used to add a new column that has the same length as the existing columns.

rbind() can be used to add a row, the added row is in the form of another data frame or list.

> d

kids ages

1 Jack 12

2 Jill 10

Example: to add new row

> rbind(d,list("Laura",19))

Example to add new column, college

>cbind(examsquiz,college="srr")

id name per college

1 1 sita 89 srr

2 2 ram 90 srr

3 3 laxman 84 srr

4 4 hanuman 95 srr

5 5 bharat 99 srr

Here we used **recycling** to add a column that is of a different length than those in the data frame:

To delete first column

>examsquiz[,-1]

Applying apply()

You can use apply() on data frames, if the columns are all of the same type.

For Example : *to find the student details who got highest percentage*

>apply(examsquiz, 2, max)

id name per

"5" "sita" "99"

**Q) Merging Data Frames**

In relational databases, two tables can be combined according to the values of a common variable. In R, two data frames can be similarly combined using the merge() function.

The simplest form is as follows:

merge(x,y)

This merges data frames x and y. It assumes that the two data frames have one or more columns with names in common. Here’s an example:

> d1=data.frame(id=c(101:103), group = c("MPCs","MECs","DSCs"))

> d1

id group

1 101 MPCs

2 102 MECs

3 103 DSCs

> d2=data.frame(id=c(101,103,105),name=c("Brahmi","Ram","sita"))

> d2

id name

1 101 Brahmi

2 103 Ram

3 105 sita

> d <- merge(d1,d2)

> d

id group name

1 101 MPCs Brahmi

2 103 DSCs Ram

Here, the two data frames have the variable id in common. R found the rows in which this variable had the same value of id in both data frames. It then created a data frame with corresponding rows and with columns taken from data frames (id,group,name).

The merge() function has named arguments by.x and by.y, which handle cases in which variables have similar information but different names in the two data frames. Here’s an example:

> d3=data.frame(roll=c(103,104,105),name=c("krishna","arjun","bheem"))

>d3

id name

1 103 krishna

2 104 arjun

3 105 bheem

>merge(d1,d3,by.x="id",by.y="roll")

id group name

1 103 DSCs krishna

Even though our variable was called id in one data frame and roll in the other, it was meant to store the same information, and thus the merge made sense.

Duplicate matches will appear in full in the result, possibly in undesirable ways.

> d4

id name

1 101 abc

2 101 xyz

3 101 def

> merge(d1,d4)

id group name

1 101 MPCs abc

2 101 MPCs xyz

3 101 MPCs def

There are three 101 in d4. There is only one 101 in d1 who belongs to MPCs. But here, in the call merge(d1,d4), it may have been the case that only one of the 101 belongs to MPCs.. It is clear from this little example that you must choose matching variables with great care.

<https://www.geeksforgeeks.org/joining-of-dataframes-in-r-programming/#:~:text=In%20R%20we%20use%20merge,on%20which%20the%20merging%20happens>.

[**https://www.stat.cmu.edu/~ryantibs/statcomp/lectures/apply.html**](https://www.stat.cmu.edu/~ryantibs/statcomp/lectures/apply.html)

**Q) Factors**

An R factor might be viewed simply as a vector with a bit more information. Added That extra information consists of a record of the distinct values in that vector, called

levels. Here’s an example:

> x <- c(5,12,13,12) # x is a vector

> xf <- factor(x) #we used factor() to convert x to factor xf

> xf

[1] 5 12 13 12

Levels: 5 12 13

The distinct values in xf—5, 12, and 13—are the levels here.

Let’s take a look inside:

> str(xf)

Factor w/ 3 levels "5","12","13":1232

> unclass(xf)

[1] 1 2 3 2

attr(,"levels")

[1] "5" "12" "13"

This is revealing. The core of xf here is not (5,12,13,12) but rather (1,2,3,2). The latter means that our data consists first of a level-1 value, then level-2 and level-3 values, and finally another level-2 value. So the data has been recoded by level.

The length of a factor is still defined in terms of the length of the data

> length(xf)

[1] 4

Common Functions Used with Factors

The tapply() Function: Suppose we have a vector x of ages of voters and a factor f showing party affiliation (YSRCP, TDP, BJP). To find the mean ages in x within each of the party groups.

> ages <- c(25,26,55,37,21,42)

> affils <- c("T","Y","Y","T","B","Y")

> tapply(ages,affils,mean)

Y T B

41 31 21

> d <- data.frame(gender=c("M","M","F","M","F","F"),

age=c(47,59,21,32,33,24),income=c(55000,88000,32450,76500,123000,45650))

> d

gender age income

1 M 47 55000

2 M 59 88000

3 F 21 32450

4 M 32 76500

5 F 33 123000

6 F 24 45650

> d$over25 <- ifelse(d$age > 25,1,0)

> d

gender age income over25

1 M 47 55000 1

2 M 59 88000 1

3 F 21 32450 0

4 M 32 76500 1

5 F 33 123000 1

6 F 24 45650 0

> tapply(d$income,list(d$gender,d$over25),mean)

0 1

F 39050 123000.00

M NA 73166.67

We specified two factors, gender and indicator variable for age over or under 25. Since each of these factors has two levels, tapply() partitioned the income data into four groups, one for each combination of gender and age, and then applied to mean() function to each group.

The split() Function

In contrast to tapply(), which splits a vector into groups and then applies a specified function on each group, split() stops at that first stage, just forming the groups.

The basic form, is split(x,f), x being a vector or data frame and f being a factor or a list of factors.

> split(d$income,list(d$gender,d$over25))

$F.0

[1] 32450 45650

$M.0

numeric(0)

$F.1

[1] 123000

$M.1

[1] 55000 88000 76500

The output of split() is a list. So the last vector, for example, was named "M.1"to indicate that it was the result of combining "M" in the first factor and 1in the second.

We wanted to determine the indices of the vector elements corresponding to male, female, and infant. We can do this in a flash with split().

> g <- c("M","F","F","I","M","M","F")

> split(1:7,g)

$F

[1] 2 3 7

$I

[1] 4

$M

[1] 1 5 6

The results show the female cases are in records 2, 3, and 7; the infant case is in record 4; and the male cases are in records 1, 5, and 6.

The by() function can be used here. It works like tapply() (which it calls internally, in fact), but it is applied to objects rather than vectors.

**Q) Working with Tables**

table() function in R Language is used to create a categorical representation of data with variable name and the frequency in the form of a table.

Syntax: table(x)

Parameters:

x: Object to be converted

Example:

df = data.frame( "Name" = c("abc", "cde", "def"), "Gender" = c("Male", "Female", "Male"))

# Calling table() function

>table(df)

Gender

Name Female Male

abc 0 1

cde 1 0

def 0 1

The following code shows how to create a frequency table for the Gender variable in our data frame:

>table(df$Gender)

Female Male

1 2

<https://www.atnyla.com/tutorial/reading-and-writing-data/7/516>

How do you find all the datasets built into R and the packages on your machine?

<http://stmarysguntur.com/cse%20materials/FULL%20R%20PROGRAMMING%20METERIAL_2.pdf>

**Unit IV**

**Q) Input / Output – Reading and Writing datasets in various formats**

The function read.table() can be used to read the .txt file directly. For example to read airquality.txt:

> airqual <- read.table("C:/Desktop/airquality.txt")

To read in data that does not already have column name information.

> airqual <- read.table("C:/Desktop/airquality.txt",header=F)

We can use the colnames() command to assign column names to the dataset.

>colnames(airqual) =c(Ozone, Solar, Wind,Temp, Month, Day)

Similarly, to read .csv files the read.csv() function can be used to read in the data frame directly

> airqual <- read.csv("C:/Desktop/airquality.csv")

In addition, you can read in files using the file.choose() function in R. After typing in this command in R, you can manually select the directory and file where your dataset is located.

Reading Excel Data Files (XLSX or XLS)

Function: read\_excel()

Common Parameters:

Sheet: The name of the sheet or its location number.

> install.packages(“readxl”)

> library(readxl)

> dataEXCEL <- read\_excel(“survey.xlsx”, sheet = 1)

> dataEXCEL <- read\_excel(“survey.xlsx”, sheet = “sheetname”)

This creates an R tibble (the newer version of an R dataframe). If you are more comfortable with R dataframes, please use:

> dfEXCEL <- as.data.frame(dataEXCEL)

To load iris dataset that is imported by default in R:

dat <- iris # load the iris dataset and renamed it dat

Writing Data to a File

Before reading any data, you must set the R working directory to the location of the data.

setwd(“…”) will set the current working directory to a specific location

getwd() will print out the current directory.

> setwd("C:/mydata")

When specifying the pathname, R reads forward slashes, whereas Windows reads backward slashes. Setting the working directory can eliminate path confusion.

In R, we can write data frames easily to a file, using the write.table() command.

> write.table(cars1, file="cars1.txt", quote=F)

The first argument refers to the data frame to be written to the output file.

The second is the name of the output file. By default R will surround each entry in the output file by quotes, so we use quote=F.

You should see a file with first column giving the index (or row number). If we wanted to create a file without the row indices, we would use the command:

> write.table(cars1, file="cars1.txt", quote=F, row.names=F)

Datasets in R

Around 100 datasets are supplied with R (in the package datasets), and others are available.

To see the list of datasets currently available use the command:

data()

We will first look at a data set on CO2 (carbon dioxide) uptake in grass plants available in R.

> CO2

To get more information on the variables in the dataset, type in

> help(CO2)

**Q) Functions:** A function is a set of statements organised together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword function. The basic syntax of an R function definition is as follows −

function\_name <- function(arg\_1, arg\_2, ...) {

Function body

}

Function Components

The different parts of a function are −

Function Name − This is the actual name of the function. It is stored in the R environment as an object with this name.

Arguments − When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

Function Body − The function body contains a collection of statements that defines what the function does.

Return Value − The return value of a function is the last expression in the function body to be evaluated.

R has many in-built functions which can be directly called in the program without defining them first. We can also create and use our own functions referred to as user defined functions.

Built-in Function

Simple examples of in-built functions are seq(), mean(), max(), sum(x) and paste(...) etc. They are directly called by user written programs.

# Create a sequence of numbers from 32 to 44.

print(seq(32,44))

# Find mean of numbers from 25 to 82.

print(mean(25:82))

# Find sum of numbers frm 41 to 68.

print(sum(41:68))

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

[1] 32 33 34 35 36 37 38 39 40 41 42 43 44

[1] 53.5

[1] 1526

**User-defined Function**

Below is an example of how a function is created and used.

# Create a function to print squares of numbers in sequence.

new.function <- function(a)

{

for(i in 1:a) {

b <- i^2

print(b)

}

}

Calling a Function

# Call the function new.function supplying 6 as an argument.

> new.function(6)

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

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

[1] 36

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get a non default result.

# Create a function with arguments.

new.function <- function(a = 3, b = 6)

{

result <- a \* b

print(result)

}

# Call the function without giving any argument.

new.function()

# Call the function with giving new values of the argument.

new.function(9,5)

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

[1] 18

[1] 45

**Variable Scope**

A variable that is visible only within a function body is said to be local to that function.

The formal parameters in an R function are local variables.

Call by value: To evaluate a function call, R copies each actual argument to the corresponding local parameter variable, and changes to that variable are not visible outside the function.

Variables created outside functions are global and are available within

functions as well.

**Q) The Top-Level Environment**

Consider this example:

> w <- 12

> f <- function(y)

{

d <- 8

h <- function()

{

return(d\*(w+y))

}

return(h())

}

> environment(f)

<environment: R\_GlobalEnv>

Here, the function f() is created at the top level—that is, at the interpreter command prompt—and thus has the top-level environment, which in R output is referred to as R\_GlobalEnv.

The function ls() lists the objects of an environment. If you call it at the top level, you get the top-level environment. Let’s try it with our example code:

> ls()

[1] "f" "w"

As you can see, the top-level environment here includes the variable w,which is actually used within f().

You get a bit more information from ls.str():

> ls.str()

f : function (y)

w : num 12

we have h() being local to f(), just like d. Scope is hierarchical. Thus, R is set up so that d, which is local to f(), is in turn global to h().

Similarly, the hierarchical nature of scope implies that since w is global to f(), it is global to h() as well. Indeed, we do use w within h().

In terms of environments then, h()’s environment consists of whatever objects are defined at the time h() comes into existence; that is, at the time that this assignment is executed:

**Q) Closures**

an R closure consists of a function’s arguments and body together with its environment at the time of the call.

*Closures are used for creating functions within a function in the R environment.*

Example:

counter <- function ()

{

ctr <- 0

f <- function()

{

ctr <<- ctr + 1

cat("this count currently has value",ctr,"\n")

}

return(f)

}

Now we will call counter() twice, assigning the results to c1 and c2.

c1 <- counter()

c2 <- counter()

The two variables (c1 and c2) will consist of functions, specifically copies of f().

Each time counter() is called, the variable ctr will be in a different environment.

In other words, different calls to counter() will produce physically different ctrs.

*The result, then, is that our functions c1() and c2() serve as completely independent counters,*

The below calls produce the output as follows

c1()

c1()

c2()

c2()

Output:

this count currently has value 1

this count currently has value 2

this count currently has value 1

this count currently has value 2

**Q) Recursion**

Recursive function is a function that calls itself.

Below is the example program to find factorial of a given number using recursion

fact <- function(x)

{

if(x==0 || x==1)

{

return(1)

}

else

{

return(x\*fact(x-1))

}

}

>fact(3)

Output:

6

**Q) Exploratory Data analysis:**

Exploratory Data Analysis or EDA is a statistical approach or technique for analysing data sets in order to **summarise their important and main characteristics** generally by using some visual aids. The EDA approach can be used to gather knowledge about the following aspects of data:

1. Main characteristics or features of the data.
2. The variables and their relationships.
3. Finding out the important variables that can be used in our problem.

EDA is an iterative approach that includes:

1. Generating questions about our data.
2. Searching for the answers by using visualisation, transformation, and modelling of our data.
3. Using the lessons that we learn in order to refine our set of questions or to generate a new set of questions.

Exploratory Data Analysis in R

In R Language, we are going to perform EDA under two broad classifications:

Descriptive Statistics, which includes mean, median, mode, inter-quartile range, and so on.

Graphical Methods, which includes histogram, density estimation, box plots, and so on.

Before we start working with EDA, we must perform the data inspection properly.

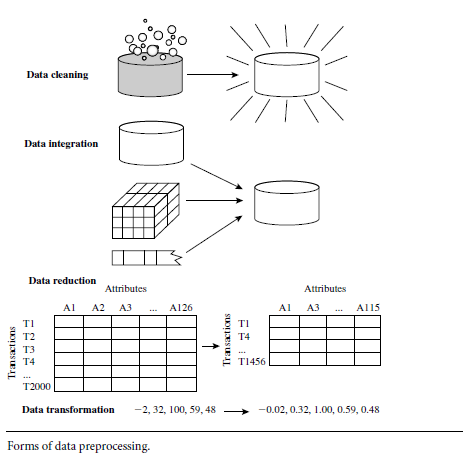
<https://www.rpubs.com/Shiva-R/204073>

<https://www.geeksforgeeks.org/exploratory-data-analysis-in-r-programming/>

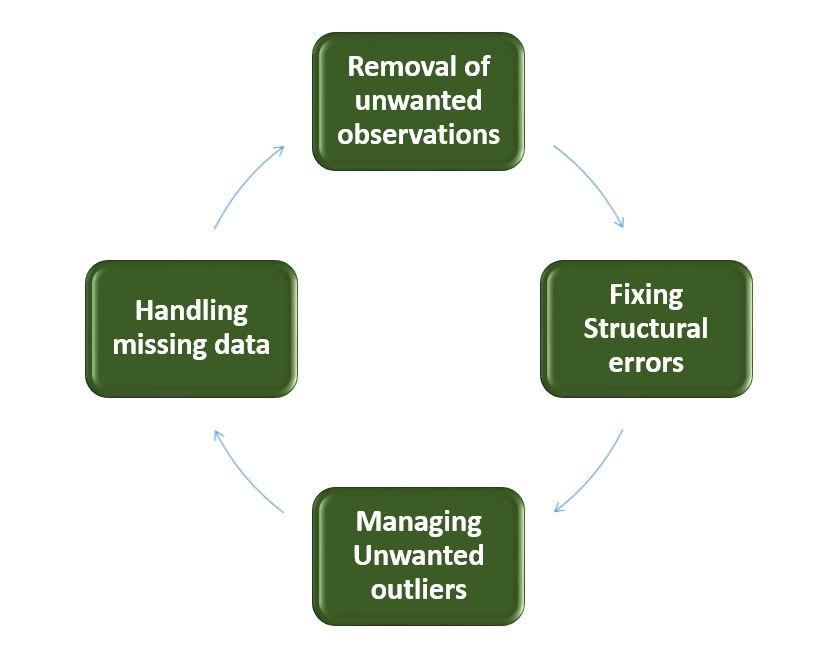
<https://r4ds.had.co.nz/exploratory-data-analysis.html>

**Q) What Is Data Preprocessing & What Are The Steps Involved?**

Data preprocessing is a data mining technique which is used to transform the raw data in a useful and efficient format.



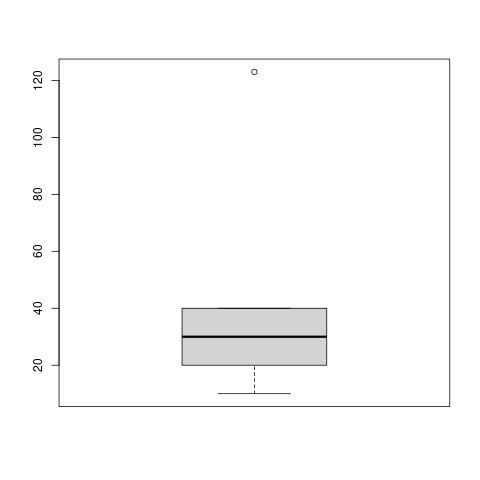
1. Data cleaning: It handles missing values, NULL or unwanted values, duplicate values, misspelt attributes, inconsistent data types and handling outliers.



*Detect Outliers With Boxplot Function*

df = data.frame(roll=c(10,20,30,40,123))

boxplot(df)



2. Data transformation: turns raw data formats into desired outputs. It also normalises the data. Normalisation is done in order to scale the data values in a specified range (-1.0 to 1.0 or 0.0 to 1.0)

For example : Consider the data below:

2001 pens 300

2001 pencils 400

2002 pens 800

2002 pencils 200

The above data can be transformed as shown below. This table format helps us to summarise quickly.

|  | 2001 | 2002 |
| --- | --- | --- |
| pens | 300 | 800 |
| pencils | 400 | 200 |

**Example of normalization:**

#define Min-Max normalization function

min\_max\_norm <- function(x)

{

(x - min(x)) / (max(x) - min(x))

}

df = data.frame(roll=c(1,2,3))

lapply(df, min\_max\_norm)

**Output**

$roll

[1] 0.0 0.5 1.0

3. Data integration: When data from multiple sources are integrated, the data after integration must be accurate and reliable. Primary keys and foreign keys are handled while integrating data.

4. Data reduction: Data reduction is a process that reduces the volume of original data and represents it in a much smaller volume. Data reduction techniques are used to obtain a reduced representation of the dataset that is much smaller in volume by maintaining the integrity of the original data.

**Q) What are the steps involved in Data Cleaning. Explain in Detail**

The success or failure of a project relies on proper data cleaning. **Steps involved in Data Cleaning**

Data cleaning tasks attempts to

* 1.Removal of unwanted data
* 2. Fixing Structural Errors
* 3. Managing unwanted outliers
* 4. Handling missing data

1. **Removal of unwanted observations**  
   This includes deleting duplicate/ redundant or irrelevant values from your dataset.

Irrelevant observations are any type of data that is of no use to us and can be removed directly.

**2. Fixing Structural errors**   
The errors that arise during measurement, transfer of data, or other similar situations are called structural errors.

For example, the model will treat red, yellow, and red-yellow as different classes or attributes, though one class can be included in the other two classes. So, these are some structural errors that make our model inefficient and give poor quality results.

3. **Managing Unwanted outliers**   
Outliers are observations which are distant from the rest of the data. Outliers can be good or bad for the data. Outliers can be used for fraud detection. Scatter plots and box plots help to identify the outliers in the data. Outliers can cause problems with certain types of models. For example, linear regression models are less robust to outliers than decision tree models. Generally, we should not remove outliers until we have a good reason to remove them. Sometimes, removing them improves performance, sometimes not.

**4. Handling missing data** The two most common ways to deal with missing data are:

1. Remove missing values.

df = data.frame(roll=c(1,NA,3))

df=df[!is.na(df)]

print(df)

[1] 1 3

1. **Fill in the missing value manually**
2. **Use a global constant to fill in the missing value:** If all missing values are replaced by “unknown”, then the mining program may mistakenly think that they form an interesting concept.

So this method is simple and not foolproof.

4**. Use the attribute mean to fill in the missing value:** For example, Use average income value to replace the missing value for income.

df = data.frame(income=c(1,NA,3))

df$income[is.na(df$income)]<-mean(df$income,na.rm=TRUE)

print(df)

income

1

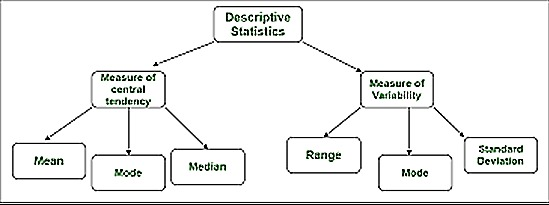
2

3

**Some data cleansing tools**

* Openrefine
* Trifacta Wrangler
* TIBCO Clarity
* Cloudingo
* IBM Infox<-c(1,1,2,2,2,2,3,3)
* hist(x)sphere Quality Stage

**Q) Descriptive Statistics in R**

****

**Q) Central Tendency**

The three commonly reported measures of central tendency are the mean, the median, and the mode. R provides built-in functions for the mean and the median. *Mean*

The mean for a population or a sample of data is defined as the sum of the data values divided by the number of values. Mean uses information from every value in the dataset. It is used in the calculation of additional measures such as standard scores, the variance, and the standard deviation.

The mean can be computed with the mean() function in R:

Syntax

The basic syntax for calculating mean in R is −

mean(x, trim = 0, na.rm = FALSE, ...)

Following is the description of the parameters used −

* x is the input vector.
* trim is used to drop some observations from both end of the sorted vector.
* na.rm is used to remove the missing values from the input vector.

When the trim parameter is supplied, the values in the vector get sorted and then the required numbers of observations are dropped from calculating the mean.

When trim = 0.3, 3 values from each end will be dropped from the calculations to find the mean.

Example:

data <- c(12,7,3,4.2,18,2,54,-21,8,-5)

result<- mean(data ,trim = 0.3)

print(result)

If there are missing values, then the mean function returns NA.

To drop the missing values from the calculation use na.rm = TRUE. which means remove the NA values.

Example: result <- mean(data,,na.rm = TRUE)

*Median*

It is the value separating the dataset into halves.

The upper half of the data contains values greater than the median, and the lower half contains values lower than the median.

The median is also called the second quartile and the 50th percentile. We can locate the median in any set of data by **sorting the data from lowest to highest**. If there are an odd number of values in the data, the median will be the observed middle value. If there are an even number of data values, the median is computed as the **mean of the two middle values.**

The median is insensitive to extreme values. We base its value only on the middle one or two data points. When the data distribution is skewed, the median is often more appropriate than the mean in describing the centre of the data.

The median can be computed thanks to the median() function in R:

Example:

data <- c(12,7,3,4.2,18,2,54,-21,8,-5)

result <- median(data)

print(result)

*Mode*

The mode can be found simply by identifying the most frequently occurring value or values in a dataset. Some datasets have no value that repeats, while other data sets may have multiple modes.

R’s built-in mode function returns the storage class of the R object.

There is no in-built function for finding mode in R. So let’s create a user-defined function that will return the mode of the data passed. We will be using the table() method for this as it creates a categorical representation of data with the variable names and the frequency in the form of a table.

Example:

x<- c(12,7,12,7,18)

names(table(x))[table(x)==max(table(x))]

[1] "7" "12"

**Q) Variability**

Three common measures of variability are the

* range,
* the variance, and
* the standard deviation.

Each has a built-in function in R. The range function returns the minimum and maximum values, so if you want the actual range, you must subtract the minimum from the maximum. The variance function is var, and the standard deviation function is sd.

*The Range*

It is a measure of how closely together or widely apart the data values are.

Example:

data<- c(12,7,12,7,18)

range=max(data)-min(data)

variance=var(data)

stddev=sd(data)

cat("range=",range, " variance =", variance, " standard deviation =" , stddev)

Output:

range= 11 variance = 20.7 standard deviation = 4.549725

**Q) summary command in R**

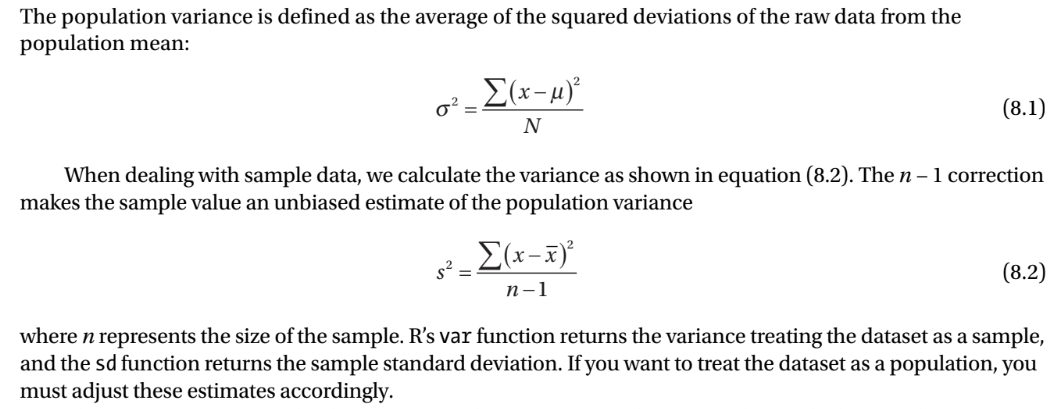
The str() command shows the structure of data. It will inform you about the number of rows and columns in the data and values in the columns with their respective heads.

The summary() command will provide you with a statistical summary of your data. It gives the output as the largest value in data, the least value or mean and median and another similar type of information.

**Example**

data = c(12,7,5)

summary(data)



**output**

Min. 1st Qu. Median Mean 3rd Qu. Max.

5.0 6.0 7.0 8.0 9.5 12.0

**Q) How is R used in data visualisation?**

*1. creating graphs*

i) The plot() Function: plot() is a generic function or a placeholder for a family of functions.

The function that is called depends on the class of objects on which it is called.

**Example:**

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

> y <- c(1,3,8)

> plot(x,y)

The above command will plot the points (1,1), (2,3), and (3,8)

Here data is represented as open circles (default pch) If you want data to be represented as \*, use pch = 8.

plot(x, y,pch=8)

<http://www.sthda.com/english/wiki/r-plot-pch-symbols-the-different-point-shapes-available-in-r>

plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")

This draws axes labelled x and y. The horizontal (x) axis ranges from −3 to 3. The vertical (y) axis ranges from −1 to 5. The argument type="n" means that there is nothing in the graph itself.

ii) Adding Lines: The abline() Function

abline(c(2,1)) draws this line : y =1 · x+ 2 to the existing plot

**Example:**

x <- c(1,2,3)

y <- c(1,3,8)

plot(x, y)

abline(c(2,1))

More lines can be added to the present graph using the lines() function.

to add a line from (1.5,3) to (2.5,3) to the present graph:

> lines(c(1.5,2.5),c(3,3))

If you want the lines to “connect the dots,” but don’t want the dots themselves, include type="l" in your call to lines() or to plot(). type=”b” will draw lines and dots.

**Example:**

x <- c(1,2,3)

y <- c(1,3,8)

plot(x, y,type="l")

abline(c(2,1))

lines(c(1.5,2.5),c(3,3),type="b")

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

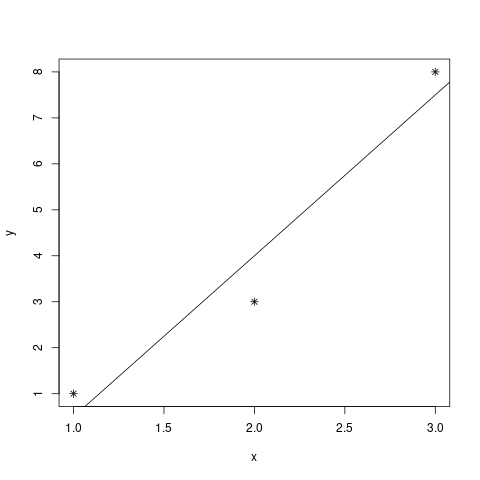
> y <- c(1,3,8)

> plot(x,y,pch=8)

> lmout <- lm(y ~ x)

> abline(lmout)

After the call to plot(), the graph will simply show the three points, along with the x- and y- axes with \* marks. The call to abline() then adds a line to the current graph. The result of the call to the linear-regression function lm() is a class instance containing the slope and intercept of the fitted line. abline() will pick up the slope and intercept it needs from lmout$coefficients and plots the line. It superimposes this line onto the current graph, the one that graphs the three points. In other words, the new graph will show both the points and the line as shown below



You can use the lty parameter in plot() to specify the type of line, such as solid or dashed. To see the types available and their codes, enter this command:

> help(par)

iii) Starting a New Graph While Keeping the Old Ones

Each time you call plot(), directly or indirectly, the current graph window will be replaced by the new one.

To plot two histograms of vectors X and Y and view them side by side in the windows system.

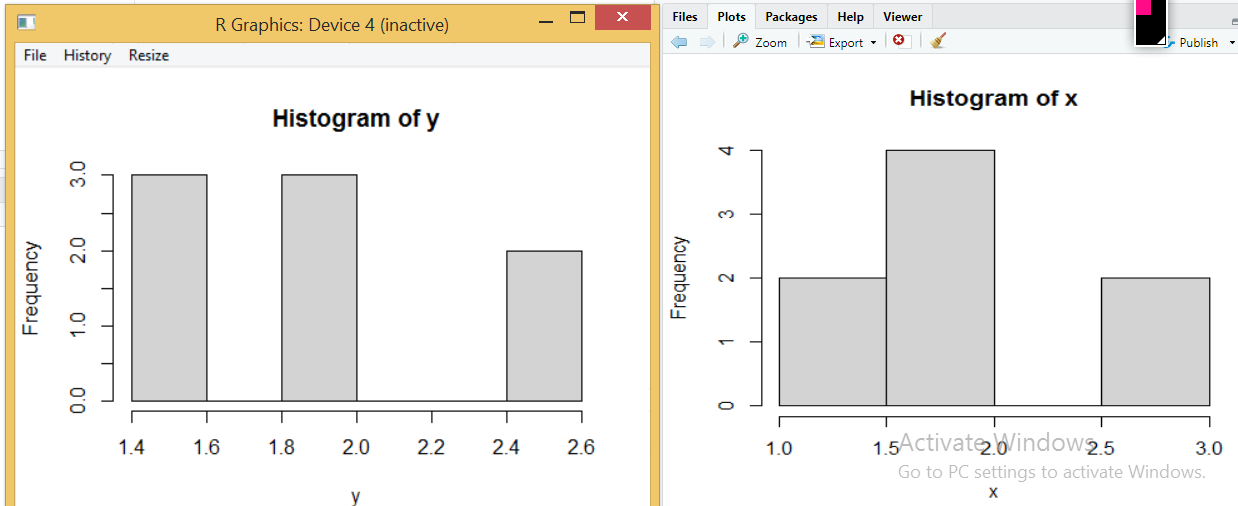
x<-c(1,1,2,2,2,2,3,3)

y<-c(1.5,1.5,1.5,2,2,2,2.5,2.5)

hist(x)

windows()

hist(y)



iv) Adding Points: The points() Function

The points() function adds a set of (x,y) points, with labels for each, to the present graph.

**Example:**

x1 <- c(1,2,3)

y1 <- c(1,3,8)

plot(x1,y1, pch ="+", type="b")

x2 <- c(1.5,2.5)

y2 <- c(3,3)

points(x2,y2,pch=25,col="blue")

v) Adding a Legend: The legend() Function

x<-1:10; y1=x\*x; y2=2\*y1

plot(x, y1,col="red", type="b")

# Add a line

lines(x, y2,col="blue")

# Add a legend

legend("topleft", legend=c("Line 1", "Line 2"),col=c("red", "blue"),lty =c(1,1))

vi) Adding Text: The text() Function

Use the text() function to place some text anywhere in the current graph.

Here’s an example:

text(2.5,4,"abc")

This writes the text “abc” at the point (2.5,4) in the graph.

2. Customizing Graphs

i) Changing Character Sizes: The cex Option

The cex (for character expand) function allows you to expand or shrink characters within a graph

To draw the text “abc” at some point, say (2.5,4), in your graph but with a larger font:

text(2.5,4,"abc",cex = 1.5)

ii) Changing the Range of Axes: The xlim and ylim Options

To have the ranges on the x- and y-axes of your plot be broader or narrower than the default.

You can adjust the axes by specifying the xlim and/or ylim parameters in your call to plot() or points(). For example, ylim=c(0,90000) specifies a range on the y-axis of 0 to 90,000.

If you have several curves and do not specify xlim and/or ylim, you should draw the tallest curve first so there is room for all of them. Otherwise, R will fit the plot to the first one your draw and then cut off taller ones at the top!

iii) Adding a Polygon: The polygon() Function

You can use polygon() to draw arbitrary polygonal objects.

# Draw an empty plot

plot(3, 3,type="n")

# Draw a polygon

polygon(x = c(2.7, 2.3, 2.2), y = c(2.6, 2.8, 2.4), col = "darkgreen")

iv) Smoothing Points: The lowess() and loess() Functions

lowess is for adding a smooth curve to a scatterplot, i.e., for univariate smoothing. loess is for fitting a smooth surface to multivariate data.

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

y <- c(1,3,4,1,3)

df=data.frame(x,y)

plot(df,pch=8)

lines(lowess(df),col="yellow")

v) Graphing Explicit Functions

Say you want to plot the function for between 0 and 5. We can use curve() function

>curve((x^2+1)^0.5,0,5)

If you are adding this curve to an existing plot, use the add argument:

> curve((x^2+1)^0.5,0,5,add=T)

You can also use plot(), as follows:

> f <- function(x) return((x^2+1)^0.5)

> plot(f,0,5) # the argument must be a function name

**Q) Creating Three-Dimensional Plots**

R offers a number of functions to plot data in three dimensions such as persp() and wireframe(), which draw surfaces, and cloud(), which draws three-dimensional scatter plots. Here, we’ll look at a simple example that uses wireframe().

> library(lattice)

> a <- 1:10

> b <- 1:15

> eg <- expand.grid(x=a,y=b)

> eg$z <- eg$x^2 + eg$x \* eg$y

> wireframe(z ~ x+y, eg)

First, we load the lattice library. The **lattice** graphics package has a function **wireframe.** Then the call to expand.grid() creates a data frame, consisting of two columns named x and y, in all possible combinations of the values of the two inputs. Here, a and b had 10 and 15 values, respectively, so the resulting data frame will have 150 rows. (Note that the data frame that is input to wireframe() does not need to be created by expand.grid().)

We then added a third column, named z, as a function of the first two columns. Our call to wireframe() creates the graph. The arguments, given in regression model form, specify that z is to be graphed against x and y. Of course, z, x, and y refer to names of columns in eg.

**Pie chart in R**

In R the pie chart is created using the pie() function

### Syntax

The basic syntax for creating a pie-chart using the R is −

pie(x, labels, radius, main, col, clockwise)

Following is the description of the parameters used −

* x is a vector containing the numeric values used in the pie chart.
* labels is used to give description to the slices.
* radius indicates the radius of the circle of the pie chart.(value between −1 and +1).
* main indicates the title of the chart.
* col indicates the color palette.
* clockwise is a logical value indicating if the slices are drawn clockwise or anti clockwise.

**Example:**

x <- c(50,40,10)

labels <- c("mpcs","mscs","dscs")

# Plot the chart.

pie(x,labels,col = rainbow(length(x)))

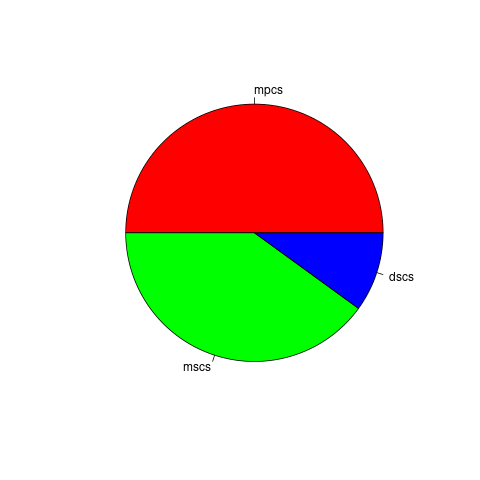
x1 = c(10,25,50,75,100)

y1 = c(1,4,12,23,37)

plot(x1, y1, pch=5, ylim=c(1,50),xlab="Sample Size", ylab="Time(s)" , main = "Time relative to Sample Size", xaxt = "n")

axis(1, at = x1, labels = x1)

lines(x1, y1, col = "gray")



**Barchart in R**

R uses the function barplot() to create bar charts. R can draw both vertical and Horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

The basic syntax to create a bar-chart in R is −

barplot(H,xlab,ylab,main, names.arg,col)

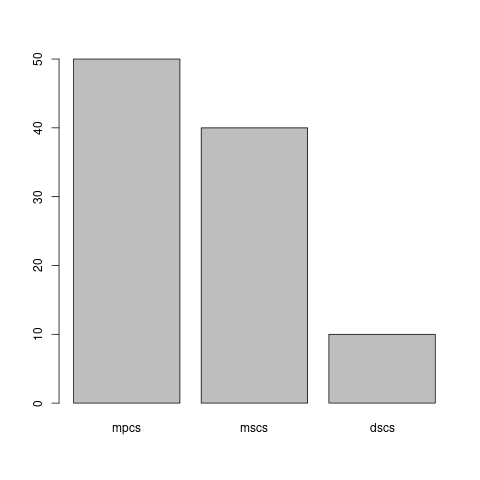
Following is the description of the parameters used −

* H is a vector or matrix containing numeric values used in bar chart.
* xlab is the label for x axis.
* ylab is the label for y axis.
* main is the title of the bar chart.
* names.arg is a vector of names appearing under each bar.
* col is used to give colors to the bars in the graph.

Example:

H <- c(50,40,10)

barplot(H,names.arg=c("mpcs","mscs","dscs"))



**Boxplot in R**

Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.

Boxplots are created in R by using the boxplot() function.

Syntax

The basic syntax to create a boxplot in R is −

boxplot(x, data, notch, varwidth, names, main)

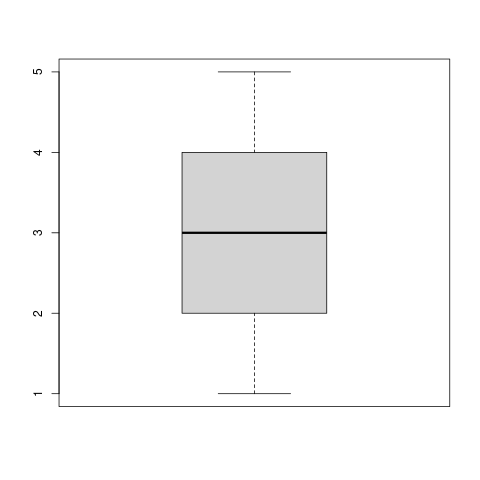
Following is the description of the parameters used −

1. x is a vector or a formula.
2. data is the data frame.
3. notch is a logical value. Set as TRUE to draw a notch.
4. varwidth is a logical value. Set as true to draw width of the box proportionate to the sample size.
5. names are the group labels which will be printed under each boxplot.
6. main is used to give a title to the graph.

Example:

data<- c(1,2,3,4,5)

boxplot(data)



first quartile (Q1), is the value under which 25% of data points are found when they are arranged in increasing order.

First Quartile(Q1) = ((n + 1)/4)th Term. Second Quartile(Q2) = ((n + 1)/2)th Term.

**Histogram in R**

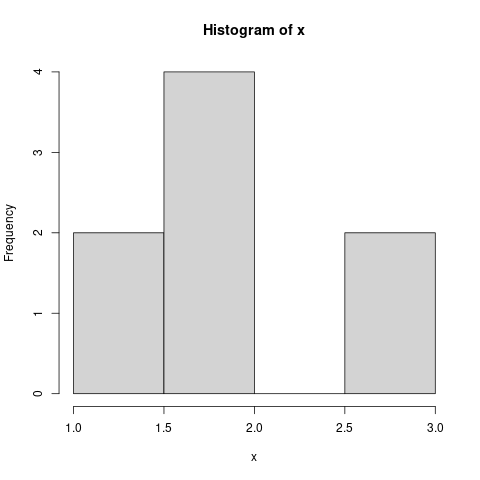
R creates histogram using hist() function. This function takes a vector as an input and uses some more parameters to plot histograms.

### Syntax

The basic syntax for creating a histogram using R is −

hist(v,main,xlab,xlim,ylim,breaks,col,border)

Example:



Scatter plot and line plot:

Data Visualization in R : Types of visualizations - packages for visualizations:

ggplot2

Lattice

highcharter

Leaflet

RColorBrewer

Plotly

sunburstR

RGL

Dygraphs

- Basic Visualizations, Advanced Visualizations and Creating 3D plots.

**Unit -V**

Inferential Statistics with R - Types of Learning - Linear Regression- Simple Linear Regression

- Implementation in R - functions on lm() - predict() - plotting and fitting regression line.

Multiple Linear Regression - Introduction -comparison with simple linear regression -

Correlation Matrix - F-Statistic - Target variables Vs Predictors - Identification of significant

features - Implementation of Multiple Linear Regression in R.

**Q) Types of Learning**

### **Supervised learning**

This machine learning type got its name because the machine is “supervised” while it's learning.

Here we provide training data with class labels. The model learns the relationship between the features and class from training data.

After the model is trained, we can use the model to predict the class of new data.

Examples:

* Predicting real estate prices
* Classifying whether bank transactions are fraudulent or not
* Finding disease risk factors
* Determining whether loan applicants are low-risk or high-risk
* Predicting the failure of industrial equipment's mechanical parts

Common algorithms used during supervised learning include neural networks, decision trees, linear regression, and support vector machines.

### **Unsupervised learning**

This machine learning type is very helpful when you need to identify patterns. Common algorithms used in unsupervised learning include Hidden Markov models, k-means, hierarchical clustering, and Gaussian mixture models.

Common applications also include clustering. Clustering groups data based on specific properties. These groups are called clusters.It identifies the rules existing between the clusters.Example:

* Creating customer groups based on purchase behaviour

### 

### **Reinforcement learning**

Reinforcement learning is the closest to how humans learn. It learns by interacting with its environment. It also gets a positive reward for correct and negative reward for incorrect. Common algorithms include temporal difference, deep adversarial networks, and Q-learning.

Example: If the algorithm classifies them as high-risk and they default, the algorithm gets a positive reward. If they don't default, the algorithm gets a negative reward. In the end, both instances help the machine learn by understanding both the problem and environment better.

* Teaching cars to park themselves and drive
* Dynamically controlling traffic lights to reduce traffic jams
* Training robots to learn policies using raw video images as input that they can use to replicate the actions they see

<https://www.edureka.co/blog/machine-learning-with-r/>

**Q) Explain in detail about linear regression and multiple linear regression.**

Regression analysis is a very widely used statistical tool.

It is used to establish a relationship model between two variables.

One **variable is called a dependent or response variable** whose value must be predicted.

**Other variable is called an independent or predictor variable** whose value is **known.**

In Linear Regression these two variables are related through an equation.

Mathematically a linear relationship represents a straight line.

A non-linear relationship creates a curve.

The general mathematical equation for a linear regression is −

**y = ax + b**

Following is the description of the parameters used −

y is the dependent variable.

x is the independent variable.

a and b are constants which are called the coefficients.

Steps to Establish a Regression

A simple example of regression is to predict the weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

Here y is weight and x is height.

The steps to create the relationship is −

1. **Gather the height and weight** of a few people.
2. **Create a relationship model using the lm() functions in R.**
3. **Find the coefficients from the model**
4. **Know the average error** in prediction. Also called residuals.
5. **Use the predict() function to predict the weight of new persons.**

For example:

heightx <- c(1,2,3)

weighty <- c(1,3,4)

relation <- lm(weighty~heightx) # Apply the lm() function.

print(relation)

When we execute the above code, it give a and b value as coefficients

b= -0.3333 a= 1.5000

Hence the line equation in y=1.5x-0.33

Explanation

| x | y | x\*y |  |
| --- | --- | --- | --- |
| 1 | 1 | 1\*1=1 | 1 |
| 2 | 3 | 2\*3=6 | 4 |
| 3 | 4 | 3\*4=12 | 9 |
| x =1+2+3=6 | y=1+3+4=8 | x\*y=1+6+12=19 | =1+4+9=14 |

n is number of observations: Here n=3

Formula to calculate b = = = = = 1.5

Now a can be calculated from the equation =

print(summary(relation))

When we execute the above code, it produces Residual standard error: 0.4082 on 1 degrees of freedom. The residual standard error is used to measure how well a regression model fits a dataset. From the coefficients,it is understood that the estimated effect of heightx on weighty is 1.5. This means that for every 1% increase in heightz, there is a correlated 1.5% increase in the weighty. The value of y-intercept is -0.33

output:

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -0.3333 0.6236 -0.535 0.687

heightx 1.5000 0.2887 5.196 0.121

Residual standard error: 0.4082 on 1 degrees of freedom

Multiple R-squared: 0.9643, Adjusted R-squared: 0.9286

F-statistic: 27 on 1 and 1 DF, p-value: 0.121

### Predict the weight of new person whose height is 2.5

newdata <- data.frame(heightx = 2.5)

result <- predict(relation,newdata)

print(result)

1

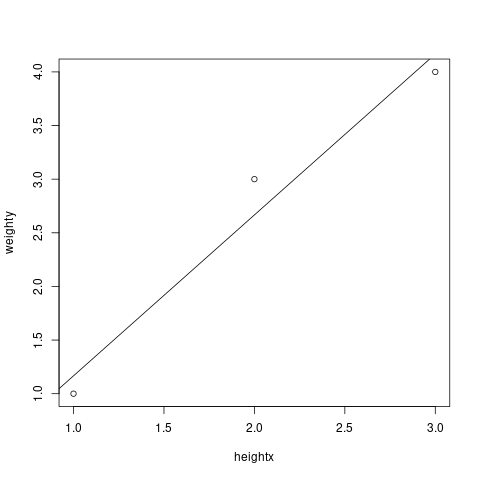
3.416667

The regression model predicted that the weight of the person is 3.14

Visualize the Regression Graphically

plot(heightx,weighty)

abline(lm(weighty~heightx))



Multiple regression is an extension of linear regression.

It finds relationships between more than two variables. In simple linear relation we have one independent and one dependent variable, but in multiple regression we have more than one independent variable and one dependent variable.

The general mathematical equation for multiple regression is −

y = a1x1+a2x2+...+b

Following is the description of the parameters used −

y is the response variable.

b, a1, a2...an are the coefficients.

x1, x2, ...xn are the predictor variables.

We create the regression model using the lm() function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for lm() function in multiple regression is −

lm(weighty ~ heightx+agex)

heightx = c(1,2,3)

weighty=c(1,3,4)

agex=c(0,3,4)

relation=lm(weighty~heightx+agex)

print(relation)

newdata = data.frame(heightx=2.5,agex=3)

predict(relation,newdata)

output:

a1=0.5, a2=0.5, b=0.5

3.25 is the predicted weight of person having height =2.5 and age=3

==========

===========

The standard errors for these regression coefficients are very small, and the t-statistics are very large (-147 and 50.4, respectively). The p-values reflect these small errors and large t-statistics. For both parameters, there is almost zero probability that this effect is due to chance.

<https://www.scribbr.com/statistics/linear-regression-in-r/>

Inferential statistics: Drawing conclusions about a population based on data observed in a sample. Hypothesis testing, T test, Ztest. The most common methodologies in inferential statistics are hypothesis tests, confidence intervals, and regression analysis.

T-test is conducted to test whether there is statistically significant difference in means of two classes or not.

It is conducted especially if there are only 2 classes

Null hypothesis: considers that the difference between means is not statistically significant (means of two classes are same)

If the error is 5% and we get p-value <0.05, then we reject null hypothesis.

We can confirm that there is statistically significant difference between means of two classes with confidence.

Degree of freedom = n-number of constraints

Here n is the size of sample population

Z-test is used to test whether the two datasets are similar or not.

<https://rpubs.com/sujith/IS>

regression: fitting the data into a function

linear regression : best line to fit 2 attributes

multiple linear regression: more than 2 attributes fitted into a multi dimensional surface

Let’s plot nonparametric density estimates (these are basically smoothed histograms). We use the function density() to generate the estimates.

x <- c(1,2,5)

y <- c(1,3,8)

plot(density(x))

lines(density(y))

**Q) F-Statistics:**

F-statistics, or F-tests, are statistical measures used to compare the variances between groups. It assesses the significance of the differences between group means or the overall model's goodness of fit by comparing the variability within groups to the variability between groups.

In R, you can calculate F-statistics using the anova() function or the summary() function applied to linear regression models. Here's a simple example:

# Create a dataset with two groups

group1 <- c(2, 4, 6, 8, 10)

group2 <- c(1, 3, 5, 7, 9)

# Perform an F-test to compare group means

f\_stat <- var.test(group1, group2)

# Print the F-statistic and p-value

print(f\_stat)

In this example, we have two groups (group1 and group2) with five observations each. The var.test() function calculates an F-test to compare the variances of the two groups. The resulting f\_stat object contains the F-statistic and the corresponding p-value.

**Q) Correlation Matrix:**

A correlation matrix is a table that displays the pairwise correlation coefficients between a set of variables. It provides insights into the strength and direction of the linear relationship between variables. The correlation coefficient ranges from -1 to 1, where values close to -1 indicate a strong negative correlation, values close to 1 indicate a strong positive correlation, and values close to 0 indicate a weak or no correlation.

To compute a correlation matrix in R, you can use the cor() function. Here's a simple example:

# Create a dataset with three variables

var1 <- c(1, 2, 3, 4, 5)

var2 <- c(2, 4, 6, 8, 10)

var3 <- c(3, 6, 9, 12, 15)

# Create a data frame with the variables

df <- data.frame(var1, var2, var3)

# Calculate the correlation matrix

corr\_matrix <- cor(df)

# Print the correlation matrix

print(corr\_matrix)

In this example, we have three variables (var1, var2, and var3) with five observations each. The cor() function computes the correlation coefficients between all pairs of variables in the data frame df. The resulting corr\_matrix object is a matrix that displays the correlation coefficients.

**Explain how to input and output data in R. Include information on how to enter data from the keyboard, how to print data with a specific number of digits, and how to handle missing values (NA), infinite values (Inf), and negative infinite values (-Inf).**

To input and output data in R, you can use various functions and options to customize the output.

Entering data from the keyboard: To enter data directly from the keyboard, you can use the scan() function. For example, to enter three numeric values, you can use the following code:

x <- scan()

This will prompt you to enter three numeric values, which will be stored in the x variable.

Printing data with a specific number of digits: To print data with a specific number of digits, you can use the options() function to set the "digits" option. For example, to print data with two digits after the decimal point, you can use the following code:

options(digits=2)

This will set the "digits" option to 2, which will affect all subsequent printing commands.

Handling missing values: In R, missing values are represented by the NA symbol. To handle missing values, you can use the is.na() function to check if a value is missing, or the na.rm option to ignore missing values in calculations. For example:

x <- c(1, 2, NA, 4, 5)

sum(x, na.rm=TRUE) # returns 12

This will sum all the values in x, ignoring the missing value.

Handling infinite values: In R, infinite values are represented by the Inf symbol. Negative infinite values are represented by -Inf. To handle infinite values

**Explain how to handle packages in R. Include the steps for installing packages, checking installed packages, and loading packages using the library() function.**

!

R packages are collections of functions, data, and documentation that extend the functionality of R. Here are the steps for handling packages in R:

Installing packages:

To install a package, you can use the install.packages() function. For example, install.packages("dplyr") will install the "dplyr" package.

You only need to install a package once, unless it is updated to a new version.

Checking installed packages:

To check which packages are installed on your system, you can use the installed.packages() function. For example, installed.packages() will list all the packages installed on your system.

You can also use the find.package() function to check if a specific package is installed. For example, find.package("dplyr") will return the path to the "dplyr" package if it is installed, or an empty string if it is not installed.

Loading packages:

Once a package is installed, you need to load it into your R session to use its functions. You can use the library() function to load a package. For example, library(dplyr) will load the "dplyr" package.

You can also use require() function instead of