CS3352 **FOUNDATIONS OF DATA SCIENCE** **L T P C 3 0 0 3**

COURSE OBJECTIVES:

- To understand the data science fundamentals and process.
- To learn to describe the data for the data science process.
- To learn to describe the relationship between data.
- To utilize the Python libraries for Data Wrangling.
- To present and interpret data using visualization libraries in Python

UNIT I INTRODUCTION 9

Data Science: Benefits and uses – facets of data - Data Science Process: Overview – Defining research goals – Retrieving data – Data preparation - Exploratory Data analysis – build the model– presenting findings and building applications - Data Mining - Data Warehousing – Basic Statistical descriptions of Data

UNIT II DESCRIBING DATA 9

Types of Data - Types of Variables -Describing Data with Tables and Graphs –Describing Data with Averages - Describing Variability - Normal Distributions and Standard (z) Scores

UNIT III DESCRIBING RELATIONSHIPS 9

Correlation –Scatter plots –correlation coefficient for quantitative data –computational formula for correlation coefficient – Regression –regression line –least squares regression line – Standard error of estimate – interpretation of r2 –multiple regression equations –regression towards the mean

UNIT IV PYTHON LIBRARIES FOR DATA WRANGLING 9

Basics of Numpy arrays –aggregations –computations on arrays –comparisons, masks, boolean logic – fancy indexing – structured arrays – Data manipulation with Pandas – data indexing and selection – operating on data – missing data – Hierarchical indexing – combining datasets – aggregation and grouping – pivot tables

UNIT V DATA VISUALIZATION 9

Importing Matplotlib – Line plots – Scatter plots – visualizing errors – density and contour plots –Histograms – legends – colors – subplots – text and annotation – customization – three dimensionalplotting - Geographic Data with Basemap - Visualization with Seaborn.

CS 3352 Foundations of Data Science

UNIT I - Introduction

Data Science: Benefits and uses - facets of data
Data Science Process: Overview - Defining Research Goals - Retrieving data - Data preparation - Exploratory Data Analysis - Build the model - Presenting findings and building applications - Data Mining - Data Warehousing - Basic Statistical descriptions of data.

<u>Big data</u> - It is a blanket term for any collection of data sets so large or complex that it becomes difficult to process them using traditional data management techniques such as RDBMS

<u>Data Science</u> - involves using methods to analyze massive amounts of data and extract the knowledge it contains.

* Relationship between big data and data science is like the relationship between crude oil & oil refinery.
* Data science & Big data evolved from Statistics and traditional data management

The characteristics of big data

– often referred to as three Vs

Volume – How much data?

Variety – How diverse are different types of data?

Velocity – At what speed is new data generated?

Additional characteristics

Veracity (or) Variability – how accurate?

Value – usefulness of Data.

Benefits and uses of Data Science & Big data

* Data Science and Big data are used everywhere in both commercial & non commercial purpose
* Commercial companies use data science and Big data to understand in deep about their customers, processes, staff and products
* To offer customers a better user experience, cross-sell, up-sell products

Eg ① Google Adsense

– which collects data from Internet users and recommends ads based on the history of search

② Maxpoint – Eg of realtime personalized advertising

③ People Analytics – HR professionals use people analytics & data mining to screen candidates, monitor the mood of employees

and study informal networks among coworkers.

* Financial Institutions use data science & big data to predict stock markets, risks of lending money & learn to attract new clients.

* Government organizations rely on data science to discover valuable information
  Eg. Data.gov – It is the home of US govt open data.
  - Detecting fraud and criminal activity
  - optimizing project funding

* Nongovernment organizations (NGOs) use it raise their funds
  - The World Wildlife Fund (WWF) – use it to increase the effectiveness of their fundraising efforts
  Eg. Datakind – Datascientist group that devotes their time to the benefit of mankind

* Universities use datascience in their research and also to enhance the study experience of their students

Online learning platforms –
Learnbay, Coursera, Udacity, Udemy & edx
Moocs – Massive Open Online Courses

Facets of data

Different types of data (or) main categories of data

* Structured
* Unstructured
* Natural language
* Machine generated
* Graph-based
* Audio, video and Images
* Streaming

① Structured data

Structured data is data that depends on a data model & resides in a fixed field within a record. It is easy to store & manage data in tables within databases or Excel files.

Eg. An Excel Table

1	Register No	Student name	Mobile no	email id
2				
3				
4				
5				

② Unstructured data

Unstructured data is not easy to fit into a data model because the content is context specific (or) varying.

Eg: Email

③ Natural language

It is a special type of unstructured data. It is difficult to process because it requires knowledge of specific data science techniques and linguistics.

Eg: Email, word documents

Natural language processing techniques are Topic recognition, Summarization, Text completion & Sentiment Analyses

④ Machine generated Data.

It is information that is automatically created by a computer, process, application or other machine without human intervention.

Eg: Web server logs, call detail records, network event logs

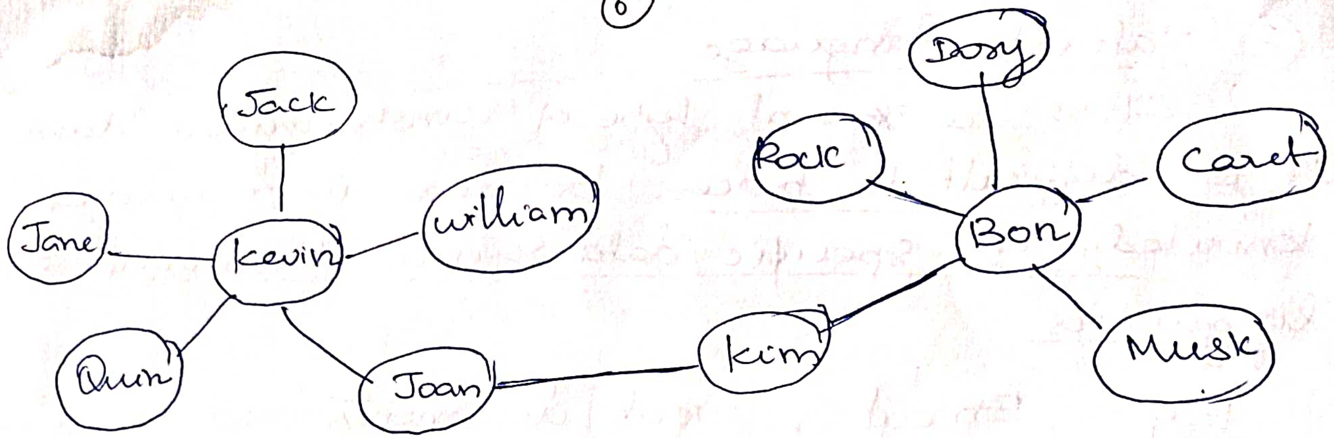
CSIPERF : TXCOMMIT : 313236 2014-11-28 11:36:13
Info CSI 00000153 Creating NT Transaction (seq 69), objectname[6] "null" 2014-11-28 11:36:13

Analysis of Machine data relies on highly scalable tools due to its high volume & speed

⑤ Graph based (or) Network data

Graph or Network data is data that focuses on relationship between objects. It uses graph structure with nodes & edges to represent objects and their relationship

Eg: Friends in Social Networks



Friends in a Social Network

⑥ Audio, Image & Video

Multimedia data in the form of audio, video, images become an integral part of everyday life

Object recognition - challenging for computers

DeepMind - developed an algorithm which is
- capable of learning how to play video games
- able to interpret everything in the video screen via deep learning

⑦ Streaming data

It takes almost any of the previous forms but it means the data flows into the system when an event happens instead of being loaded into a data store in a batch.

Eg: "Whats trending" on Twitter, live sporting or music events

1.3 The Data Science Process

The data science process consists of 6 steps. They are

1. Setting the research goal
2. Retrieving data
3. Data preparation
4. Data exploration
5. Data modeling
6. Presentation and automation

1. Setting the research goal

The purpose of this step is making sure all the stakeholders understand the what, how and why of the project.

2. Retrieving data : This step includes finding suitable data and getting access to the data from the data owner. This results in data in its raw form
3. Data preparation : This includes transforming the data from a raw form into usable data.

This involves detecting & correcting different kinds of error in the data, combine data from different data sources & transform it.

4. Data exploration : This step involves finding patterns, correlations and deviations based on visual and descriptive techniques to gain a deep understanding of the data
5. Model building

Select the variables to build the model & also a modeling technique.

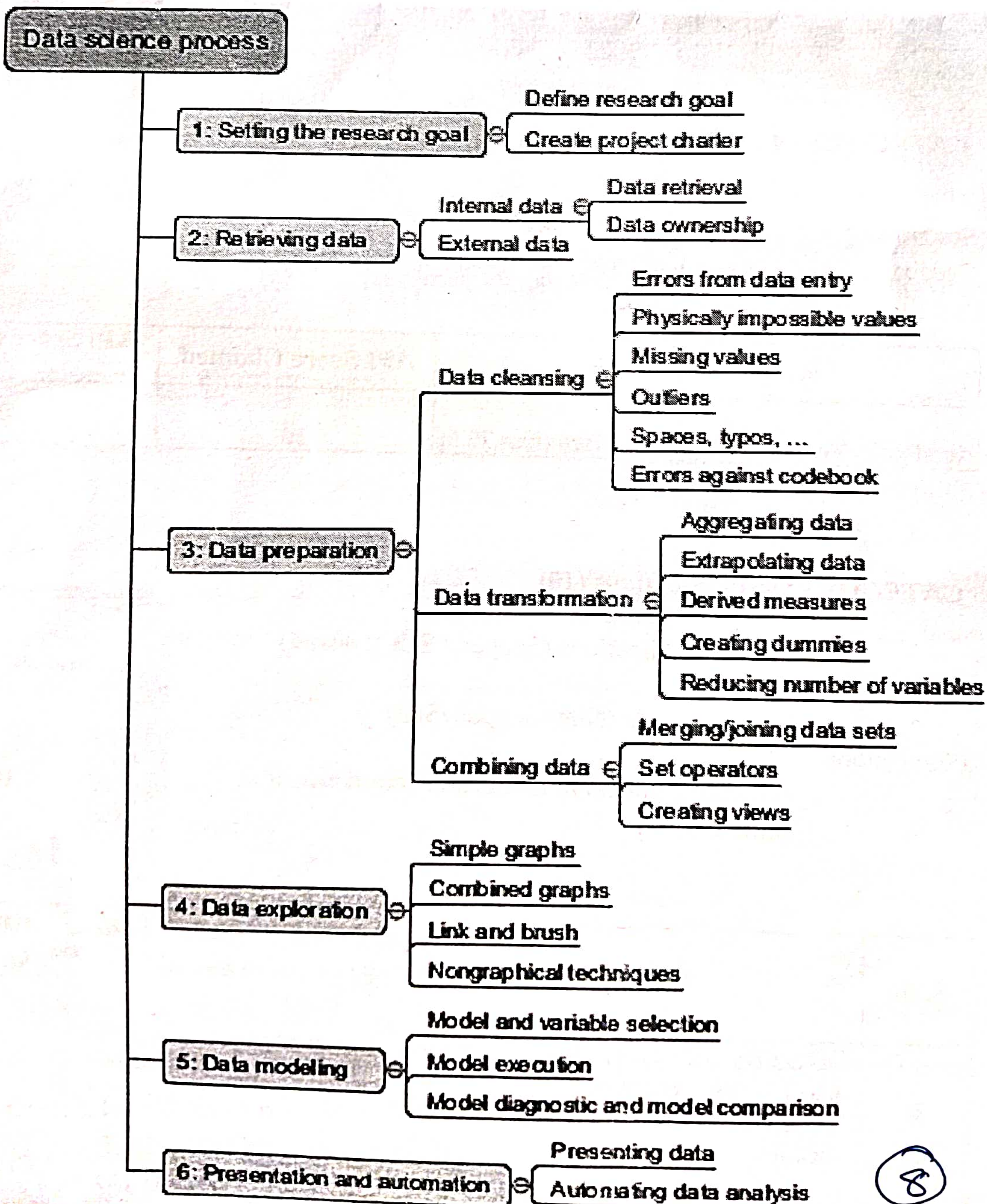
Building models with the goal of making better predictions, classifying objects

⑥ Presentation and automation

Finally present the results to the business. Results can take many forms ranging from presentation to research reports. Automate the execution of the process if needed.

Overview of the data Science Process

The following figure summarizes the data Science Process



Step 1: Defining research goals and creating a project charter

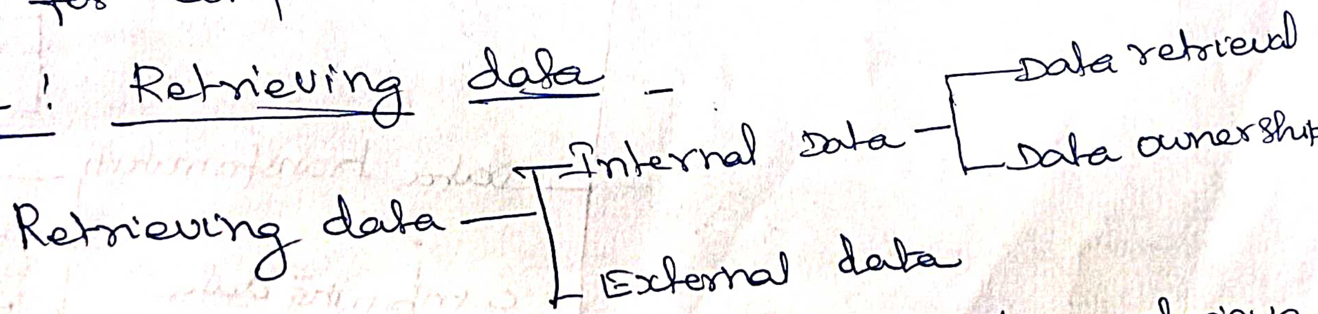
* Understanding the what, why and how of the project & answering questions is the goal of first phase
* Outcome should be a clear research goal
  - good understanding of the context
  - well defined deliverables
  - plan of action with a timetable

Creating a project charter

A project charter ~~requires~~ covers the following
- A clear research goal
- The project mission and context
- How do you perform the analysis
- what resources are expected to use
- Proof of concepts
- Deliverables and a measure of success
- A timetable

* A client can use this information to estimate the project cost, data and people required for completion of project.

Step 2: Retrieving data -



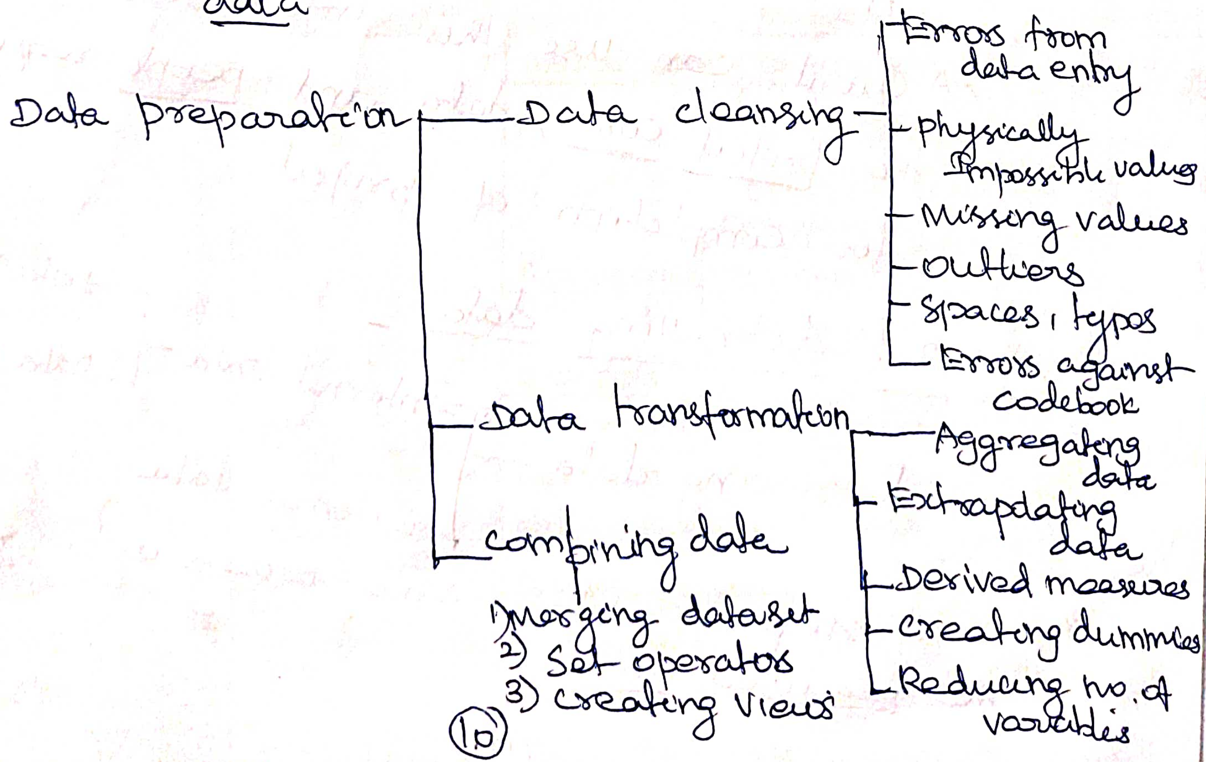
* Second step of data science process is to retrieve the required data.

* Many companies have already collected & stored the data for use internally
* More orgranizations are making high quality data freely available for public & commercial use
* Data can be stored in official data repositories such as databases, data marts, data warehouse & data lakes
* Getting access to data is difficult task. So organizations have ~~polies~~ policies which controls the access of data by everyone in that organization [ie Access rights – who can access what data]

A list of open data providers

1) Data.gov – the home of the US government's open data
2) open-data.europa.eu – the home of European commission's open data
3) Data.worldbank.org – open data initiative from the World Bank

Step 3: cleansing, Integrating and transforming data



* The data received from the data retrieval phase is like "a 'diamond in the rough'". So it must be sanitized & prepared for use in modeling & reporting phase

Cleansing data

Data cleansing is a subprocess of data science process that focuses on removing errors in the data to make the data become true & consistent one

An overview of common errors

Errors Description	possible solution
1) Mistakes during data entry	Manual overrules
2) Redundant white spaces	Use String functions
3) Impossible values	Manual overrules
4) Missing values	Remove observation (or) value
5. outliers	Validate, if erroneous treat as missing
6) Deviations from a code book	Match on keys (or) else use manual overrules
7) Different units of measurements	Recalculate
8) Different levels of aggregation	Bring to same level of measurement

Possible errors with examples:

1) Data entry errors - Entry as "Godo" instead of "Good"
2) Redundant white spaces - Redundant white spaces at the end of a string causes errors, very hard to detect
3) Impossible values - people taller than 3 meters or people with an age of 299 years
4) Outliers - ~~Pixel with value~~ is a data that lies abnormally far away from other values in a data set.

Score values
15, 25, 30, 28, 35
98 - outlier (11)

Deviations from a code Book – Detecting errors in large data sets against a code book

Different units of measurements – Recalculate
Eg Prices/ltr & Prices/gallon

Different levels of aggregation – Bring to same level of measurement

Eg: Salary/week and
Salary / work week

* A good practice is to correct errors as early as possible.

Combining data from different data sources

*Different ways of combining data

(1) <u>Joining</u> – enriching an observation from one table with information from another table

Eg

Client	Item	Month
John	Coca-cola	January
Jane	Pepsi-cola	January

client	Region
John	Tamilnadu
Jane	karnataka

<u>Result of joining two tables</u>

client	Item	Month	Region
John	Coca-cola	January	Tamilnadu
Jane	Pepsi-cola	January	karnataka

(2) <u>Appending</u> <u>Table</u> – Adding the observations of one table to those of another table

(1)

client	Item	Month
John	Coca-cola	January
Jane	Pepsi-cola	January

(2)

client	Item	Month
Jack	Zero-cola	February
Donn	Maxi-cola	February

Client	Item	Month
John	Coca-cola	January
Jane	Pepsi-cola	January
Jack	Zero-cola	February
Donn	Maxi-cola	February

* To avoid duplication of data, one can virtually Combine data with views
* Also data enrichment can be done by adding calculated information

Transforming Data

* Transforming data makes the data suitable for Data Modeling

Eg- Transforming x to log x

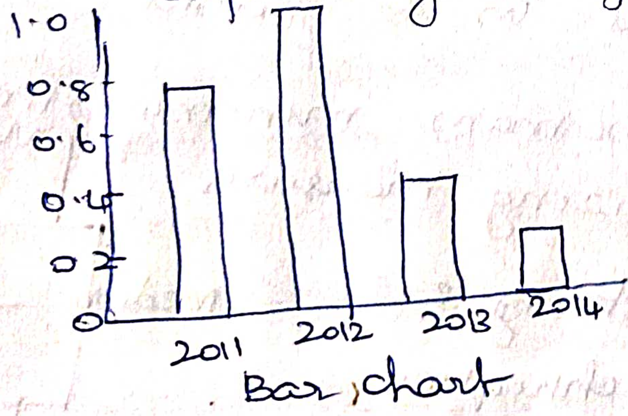
- Reducing the no. of variables (Principal component Analysis)
- Turning variables into dummies
  Dummy variables can only take two values true(1) (or) ~~fta~~ false(0)

Step 4: Exploratory data analysis

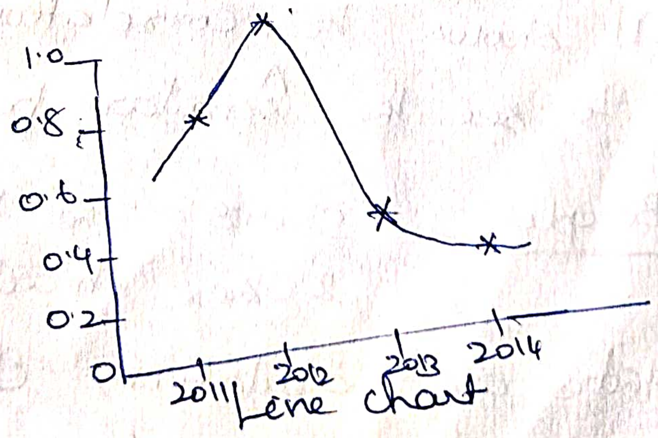
Helps to gain deep understanding of data & the interactions between variables

Data exploration
- Simple Graphs
- Combined Graphs
- Link & brush
- Nongraphical techniques

* A bar chart, a line chart & histograms used in exploratory analysis



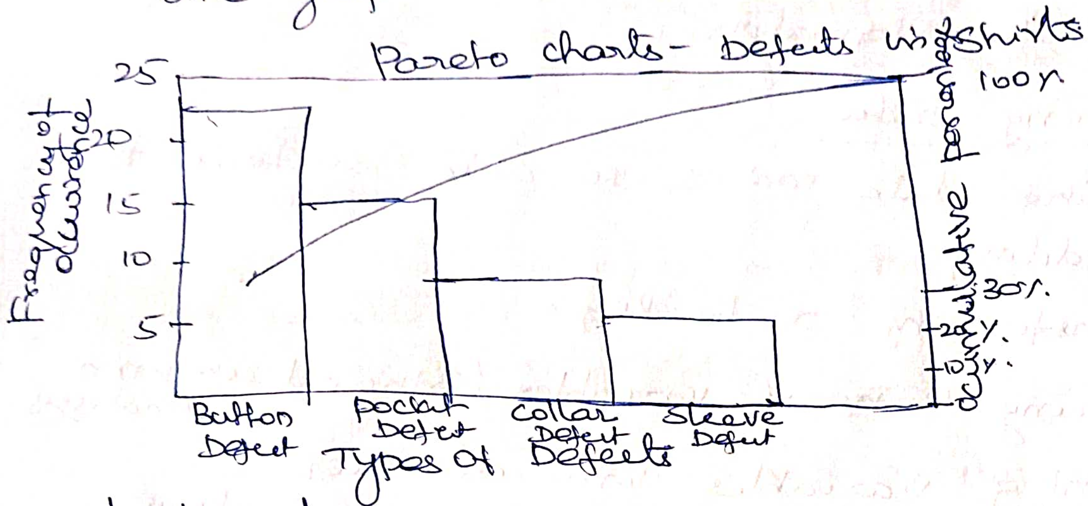
Bar chart



Line chart

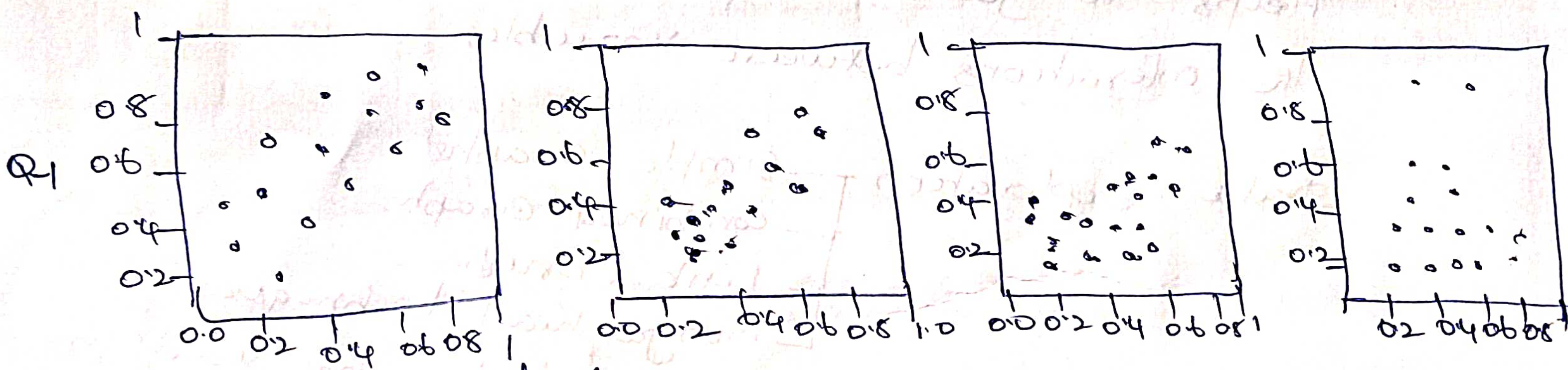
Combined graph – Pareto diagram

Pareto diagram is a combination of a bar graph & a line graph



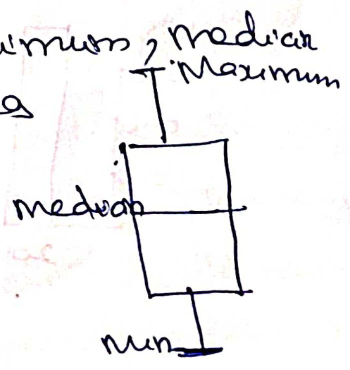
Link and Brush – helps to combine & link different graphs & tables. So changes in one graph are automatically transferred to the other graphs

The following shows the average score per country for questions



It shows the high correlation between answers & also to see the similar points in the graphs

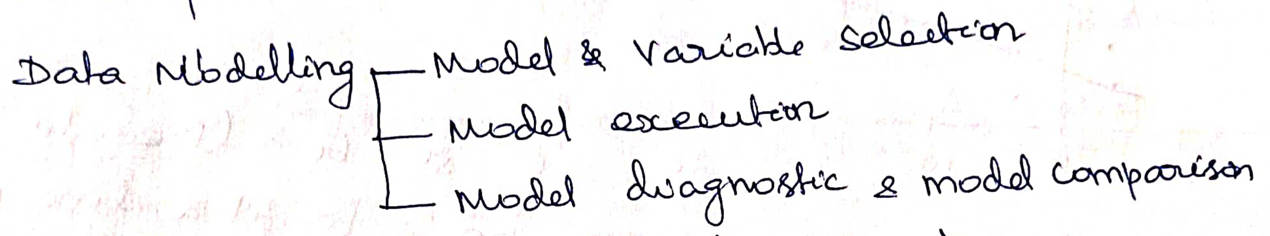
Boxplot – shows the maximum, minimum, median & other characterizing measures



* others are Tabulation, clustering & other modeling techniques

Step 5: Build the Model

* Building models with the goal of making better predictions, classifying objects
* The components of Model building



* Building a model is an iterative process
* It depends on statistics & machine learning techniques

(a) Model & variable selection

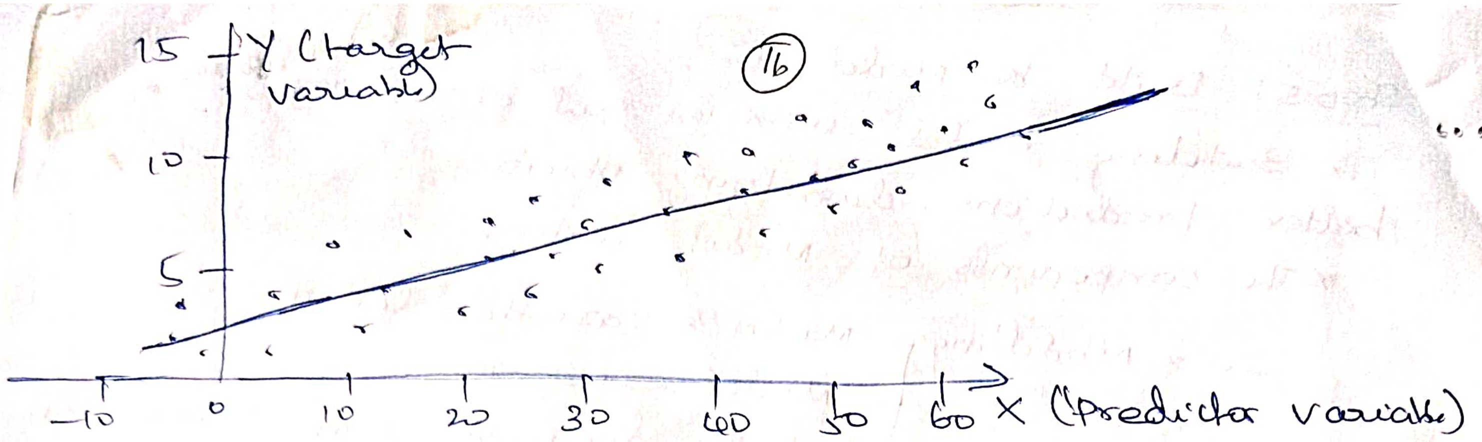
* Many modelling techniques are available,
* Choosing the right model based on factors such as model performance & project requirements, easy to implement, difficulty in maintenance & easy to explain

(b) model execution

* Implementation of choosen model in code
* Python has libraries such as StatsModels (or) Scikit-learn to implement models

Implementation of Linear Regression Model

```
Import Statsmodels.api as sm     // Imports necessary python modules
import numpy as np
predictors = np.random.random(1000).reshape(500,2)
target = predictors.dot(np.array([0.4, 0.6])) + np.random.random(500)
lmRegModel = sm.OLS(target, predictors)    // Fits linear Regression on data
result = lmRegModel.fit()
result.summary() → shows model statistics
```



* Linear regression tries to fit a line while maintaining the distance to each point

Linear equation is

$Y = 0.7658x_1 + 1.215\,x_2$

* Some important output parts of linear regression model

a) Model fit : R-squared or adjusted R-squared is used
- It is an indication of the amount of variation in the data that gets captured by the model

b) Predictor variables have a coefficient

c) Predictor significance - how significant the predictor

Model diagnostics and model comparison

- choose the best model based on multiple criteria
- Multiple error measures are available

* one is mean square error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

↳ It checks for each prediction how far it is from the actual.

* Choose the model with the lowest error

* Verification of model assumptions is called model diagnostics (meet out of)

Step 6: Presenting findings and building applications on top of them

Presentations automation — Presenting data
— Automating data analyses

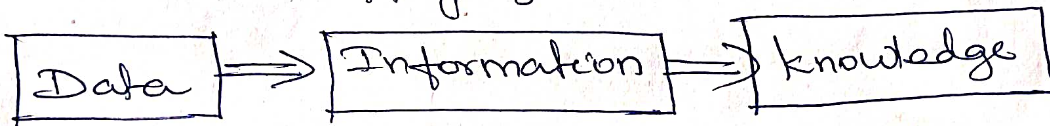
* Presenting the findings to the world through powerpoint presentations
* Automate the models in order to do the tasks repetedly

Data Mining

Data is raw fact (or) disconnected fact

Information is the processed data

Knowledge – is derived from information by applying rules to it



Data Mining – is the process of extracting hidden, valid and potentially useful patterns in huge data sets.

Steps in Data Mining

(1) **Data cleaning** – Remove noise and inconsistent data

(2) **Data Integration** – where multiple data sources may be combined

(3) **Data selection** – where data relevant to the analyses task are retrieved from the database

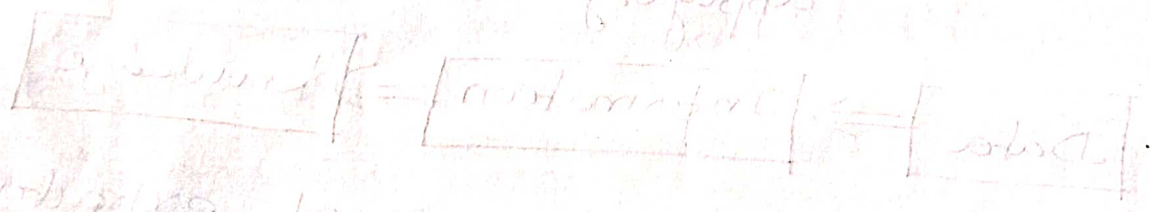
(4) **Data transformation** – where data are transformed & consolidated into forms appropriate for mining by performing summary or aggregation operations

(5) <u>Data Mining</u> – An essential process where intelligent methods are applied to extract data patterns

(6) <u>Pattern evaluation</u> – To identify the truly interesting patterns representing knowledge

(7) <u>Knowledge presentation</u> – where visualization & knowledge representation techniques are applied to present mined knowledg to user

~~Data Mining process~~



Types of Data for mining

(1) Flat Files: Transactions data, Time-Series Data scientific measurements

(2) Database data – Relational databases

(3) Data warehouse data – A data Warehouse is a repository of information collected from multiple sources, stored under a unified schema.

(4) Transactional data

A transaction includes a unique transaction ID and a list of items making up the transactions

Data Warehousing

It is the process of compiling and transforming data and making it available to users in a timely manner

Data Warehouse

A single, complete and consistent store of data obtained from a variety of diffpront sources made available to end users in a way understand and use in a business context

Eg: Walmart - 24 TB (TeraBytes)

National Medical Records - 10^18 Bytes, ie order of ExaBytes

Key characteristics of Data Warehouse

* A data warehouse is a
  * Subject oriented - Provides topicwise information [eg. Sales, Inventory]
  - Integrated - Data from varied sources in a consistent format
  - Time Varying - Data with time
  - Non-Volatile - Data remain unchanged

collection of data that is used primarily in organizational decision making

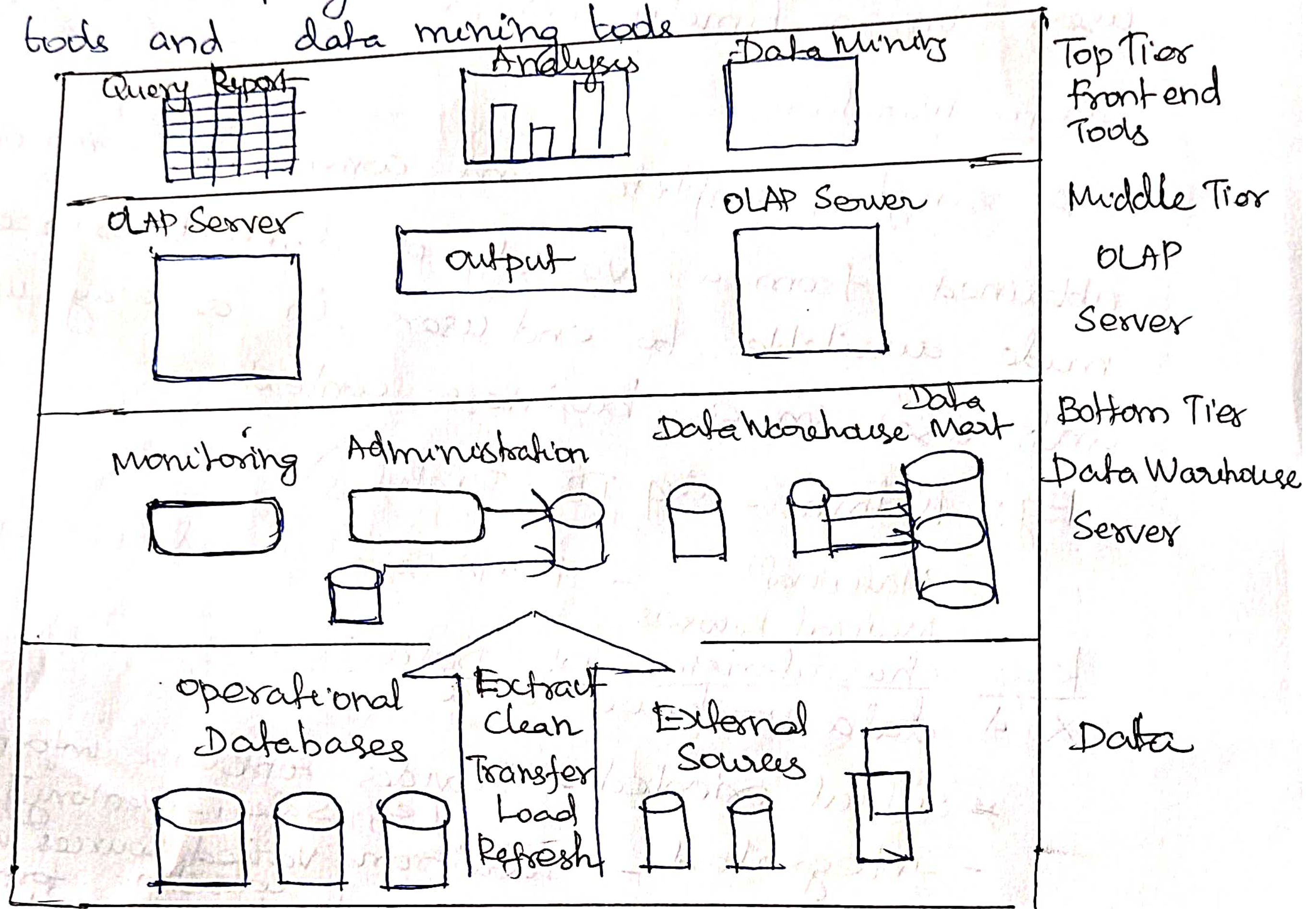
Data WareHousing - Architecture

Generally a data warehouse adopts a three-tier architecture.

Bottom tier - It is the database server. It is the relational database system. Back end tools and utilities are used to feed data into the bottom tier. They perform Extract, clean, Load & refresh function

Middle Tier - Here OLAP Server can be implemented in either Relational OLAP or Multidimensional OLAP

Top Tier - This is front-end client layer. This layer holds the query tools and reporting tools, analysis tools and data mining tools



Process Flow in Data Warehouse

* There are 4 main processes
1) Extract and load the data
2) cleaning and transforming the data
3) Backup and archieve the data
4) Managing queries and directing them to the appropriate data sources

Extract & Load process - [Load Manager]
This process extract data from operational databases & External sources & loads it into the data warehouse in consistent form

Clean & Transform Process

There are 3 steps in cleaning & transforming process

(1) Clean & Transform data into a structure

(2) Partition the data – partition each fact table into multiple partitions

(3) Aggregation – It is required to speed up common queries

Backup & Archieve the Data – [Warehouse Manager]

* In order to recover the data in the event of data loss, S/w failure or hardware failure it is necessary to keep regular backup.

* Archieving involves removing the old data from the system in a format that allow it to be quickly restored

Query Management process – [Query Manager]

This process peforms the following function

* manages the queries
* Speed up the execution time of queries
* directs the query to their most effective data source
* monitors the actual query profiles

Online Analytical Processing [OLAP]

* OLAP allows managers, and analysts to get an insight of the information through fast consistent and interactive access to information

* It enables end clients to perform better analysis of record in multiple dimensions

OLAP Applications

(a) Finance and Accounting

* Budgeting
* Financial performance analysis
* Financial modeling

(b) Sales and Marketing

* Sales analysis and forecasting
* Market research analysis
* customer analysis

(c) Production

* Production planning
* Defect analysis

* OLAP pre-calculates most of the queries namely aggregation, joining and grouping.

UNIT IV PYTHON Libraries for Data Wrangling

Basics of Numpy arrays - aggregations - computations on arrays - comparisons, masks, boolean logic - fancy indexing - structured arrays - Data manipulation with pandas - data indexing and selection - operating on data - missing data - Hierarchical indexing - combining datasets - aggregation and grouping - pivot tables.

Basics of Numpy Arrays:

Why Numpy?

* Numpy - stands for Numerical Python.
* It is a Python library used for working with an array.
* Numpy is a powerful N-dimensional array object & has functions for working in domain of linear algebra, fourier transform and matrices
* In Python lists serve the purpose of arrays but lists are slow to process
* Numpy - Provide an array object called as ndarray, which is 50x faster than traditional python lists

(28)

Why is NumPy faster than lists?

* NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access & manipulate them very efficiently.

* NumPy library is partially written in C or C++ & partially written in Python

* Data Manipulation in Python is done with NumPy array manipulation

* Data manipulation includes access data & subarrays, split, reshape and join the arrays.

Few categories of basic array manipulations:

1) Attributes of arrays: Determining the size, shape, ~~and~~ memory consumption & data types of arrays.
2) Indexing of arrays: Getting and setting the value of individual array elements
3) Slicing of arrays: Getting & setting smaller subarrays within a larger array
4) Reshaping of arrays: Changing the shape of a given array
5) Joining & Splitting of arrays: Combining multiple arrays into one, and splitting one array into many

(1) Numpy Array Attributes

- Determining the size, shape, dimension, memory consumption & data types of arrays

Eg

```
import numpy as np
np.random.seed(0)
x1 = np.random.randint(10, size=6)  # one dimensional array
x2 = np.random.randint(10, size=(3,4))  # 2 dimensional array
x3 = np.random.randint(10, size=(3,4,5))  # 3 dimensional array
```

(or)

```
import numpy as np
arr = np.array([1,2,3,4,5])
print(arr)
```

* Each array has attributes

ndim - no. of dimensions

shape - size of each dimensions

size - total size of the array

* The following code prints the dimension, shape and size of array object x3

```
Print ("x3 ndim :", x3.ndim)
Print ("x3 Shape :", x3.Shape)
Print ("x3 Size :", x3.size)
```

O/P :

```
x3 ndim : 3
x3 Shape : (3,4,5)
x3 Size : 60
```

* Datatype of the array is printed by

```
O/p Print ("x3 dtype :", x3.dtype)
O/p x3 dtype : int 64
```

dtype - datatype of the array

* Memory consumption or total size of the array is printed by
  - itemsize – size of each item
  - nbytes – Total size of the array

```
I/p:  Print ("itemsize :" x3.itemsize, "bytes")
      print ("Totalsize : x3.nbytes, "bytes)
```

```
O/p   itemsize : 8 bytes
      Totalsize : 480 bytes
```

(2) Array Indexing: Accessing Single Elements

* Array indexing is the same as accessing an array element.
* The indexes in NumPy array start with 0 i.e first element has index 0 and the second element has index 1 etc. It is written by specifying the desired index in square brackets

```
import numpy as np
arr = np.array ([1,2,3,4])
print (arr[0])
```

```
O/p: 1
```

· To index from the end of the array, negative indices are used

```
print (arr[-1])
O/p : 4
```

In case of multi-dimensional array

Eg: Access the element on 1st row & 2nd column

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0,1])
```

o/p: 2

* using negative indices

```
print(arr[1,-1])
```

o/p: 10

* modify values

```
arr[0,1] = 5
print(arr[0,1])
print(arr)
```

o/p: 5

o/p: 5 2 3 4 5
6 7 8 9 10

* Numpy arrays have a fixed type

Inserting a floating point value to an integer array, the value will be truncated.

```
arr[0,1] = 3.14159
print(arr[0,1])
```

o/p: 3

```
print(arr)
```

o/p: 3 2 3 4 5
6 7 8 9 10

I. Numpy array Attributes:

- Determining the size, shape, dimension, data type & memory consumption of arrays.

Each array has the following attributes

(1) ndim - number of dimensions eg one dimension, 2D or 3D

(2) shape - size of each dimension eg (3x3) or (2x3) or (3x3x4)

(3) size - total size of the array i.e no of elements in the array

(4) dtype - datatype of the array

(5) itemsize - size of each item in bytes

(6) nbytes - total size of the array in bytes

Eg: Sample python program

```
import numpy as np
x1 = np.random.randint(10, size=6) # one dimensional array

x2 = np.random.randint(10, size=(3,4))
        # create 3x4 2D array with random values within the range 10
x3 = np.random.randint(10, size=(3,4,5))
    # create 3D array with size 3x4x5
```

```
print ("x3 ndim :", x3.ndim)
print ("x3 shape :", x3.shape)
print ("x3 size :", x3.size)
print ("x3 dtype :", x3.dtype)
print ("x3 itemsize in bytes :", x3.itemsize)
print ("x3 totalsize in bytes :", x3.nbytes)
```

output

```
x3 ndim : 3
x3 Shape : (3, 4, 5)
x3 size : 60
x3 dtype : int64
x3 itemsize in bytes : 8
x3 totalsize in bytes : 480
```

II. Array Indexing

- Accessing single element

* The indexes in Numpy array starts with '0'
* Indexing specify the desired index in square bracket
* Negative indexes are used to index from the end of the array
* Getting & setting the value of array element indexing is used.

Example : Getting a value

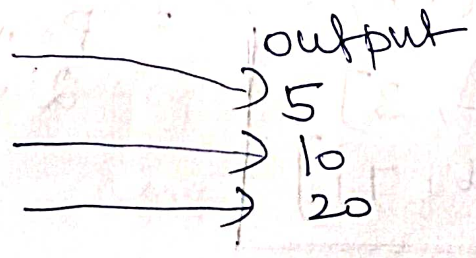
```
import numpy as np
a1 = np.array([1,2,3,4]) # create 1D array
                            [1,2,3,4]
a11 = np.array([1,2,3,4,5],[6,7,8,9,10])
print(a1[0])      # print the first item in a1 array
print(a11[0,1])   # print the item on 1st row
                    and 2nd column
print(a1[-1])     # print the last item in a1 array

print(a11[1,-1])  # print the last item in
                    a11 array
```

output

```
1
2
4
10
```

Example : Setting a value

```
a1[0] = 5       # modify (or) set the first item
                  in a1 array as '5'
a11[0,1] = 10   # set the item in 1st row
                  and 2nd column as '10'
a1[-1] = 20     # set the last item in a1 array
                  as '20'
print(a1[0])
print(a11[0,1])
print(a1[-1])
```



output

```
5
10
20
```

iii Array Slicing:

— Accessing subarray

Syntax for array slicing is

arrayname [start : stop : step]

Start - is assumed as '0' if it is unspecified

Stop - is assumed as 'size of dimension' if it is unspecified

Step - is assumed as '1' if it is unspecified

Example

```
import numpy as np
a = np.array([1,2,3,4,5]) # create 1D array [1,2,3,4,5]
b = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# create 2D array
```

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

```
Print (a[ : 3])  # print the first 3 items of array a starts with index '0'
                 # ie [1, 2, 3]
```

o/p ⇒ [1, 2, 3]

```
Print (a[2:])  # print the items of array a starts with index '2' & upto
               # end of array ie [3, 4, 5]
```

o/p ⇒ [3, 4, 5]

```
Print (a[1:3])  # Print the items starts with index '1' & upto index '2'
                # ie [2, 3]
```

o/p ⇒ [2, 3]

```
Print (b[: 2, : 3])  # Print the items in first 2 rows & first 3 columns of array b
```

o/p ⇒ [[1, 2, 3]
[5, 6, 7]]

Note:
if the step value is '-ve', default values of Start & stop are swapped.
eg:
print(a[::-1]) # Print the values of array a in reverse order.

o/p ⇒ [5, 4, 3, 2, 1]

Print(a[5::-2])

o/p ⇒ [5, 3, 1]

Fancy Indexing:
Access and modify the portions of array by using simple indices, slices, Boolean mask these are array Indexing, We are going to see about fancy Indexing. Fancy Indexing is like a simple Indexing but we pass the array of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array values.

Exploring Fancy Indexing:-
It means passing an array of indices to access multiple array elements at once.

```
In[1]: import numpy as np
       rand = np.random.RandomState(42)
       x = rand.randint(100, size=10)
       print(x)

[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements

```
In[2]: [x[3], x[7], x2]]

Out[2]: [71, 86, 14]
```

Alternatively we pass a single list or array of Indices to obtain the same result.

```
In[3]: ind = [3, 7, 4]
       x[ind]

Out[3]: array([71, 86, 60])
```

With fancy Indexing

The shape of the result reflects the shape of the Index arrays

```
In[4]: ind = np.array([[3, 7], [4, 5]])
       x[ind]

Out[4]: array([[71, 86], [60, 20]])
```

Structured Array

Array with some compound data types. Here the structure provides efficient store for compound heterogenious data.

Name (string)	Age int 32	Height float 64.
Alice	23	173.4
Bob	85	165.5
Catty	18.	175.8
Duke	28	179.4

```
dt = np.datatype ([`Name', `U10'), (`Age', `i4'),
        (`Heights', `f8')
data = np.array ([`Alice', 23, 178.4), (`Bob', 85, 165.5)
  (`Catty', 18, 175.8), `Duke', 28, 179.4), dt = dtype
        print (data.dtype) ——> name  `U10
without table => print data       `Age'  `i4'
        print (data[`Name'])       Height  `f8'
        print (np.sort (data, order=`Age'))

data= np.zeros (4, dtype= {`name': (`Name', `age',
        `Height'), `format': (`U10',
        `i4', `f8')}})
```

Create a structured Array with fields such as employee, Qualification, designation and salary then printed details in the structured array normal order as well as sorted order.

Name [String]	Qualification	Salary	Designation.
Jeno	BE	20000	Employee
Anu	BE	30000	Assistant
Safrin	ME	50000	Manager.
Sabi	MSC	25000	Employee 2
Radha	ME	30000	Employee 3

```
dt = np.datatype([('Name', 'UIO'), ('salary', 'f8',
    'Qualification', 'UIO', 'designation', UIO))
data = np.array([('Jeno', 20000, BE, Employee),
    ('Anu', 30000, BE, Assistant), ('Safrin', 50000,
      Manger), ('Sabi', 25000, MSC, Employee2),
('Radha', 30000, ME, Employee3))
print(data)
```

The Pandas Index object :-

⇒ We have seen both the Series and Dataframe object contain explicit index, unhidden
↳ used to reference & modify data.

Index Object → contain interesting data structure itself
- ↙ It may be either immutable array unchanged
- ↘ Ordered set.

Ordered set → Technically multiset, Index object may contain repeated values.

Some operations available on Index object

Eg) Let's construct an Index from a list of Integers.

```
In[30]: ind = pd.Index([2,3,5,7,11])
        ind
Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

(64 bits)

Index as Immutable Array :-

Index object in many ways operates like an array. Eg) We can use std Python Indexing notation to retrieve the values or slices.

```
In [31]: ind[1]
Out[31]: 3
In [32] : ind [::2]
out [32]: Int64Index([2, 5, 11], dtype='int64')
```

Index object also have many of the attributes familiar from NumPy Arrays.

```
In [33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
         5 (5,) 1 int64
```

(40)

Difference between Index object and NumPy arrays is

↳ Indices are immutable. So it cannot be modified in a normal way via the normal means.

```
In[34]: ind[1]=0
```

Type Error : Index does not support mutable operations.

Index as Ordered set :-

Pandas object are designed to facilitate operations such as Join across dataset ↳ which depends on many aspect of set arithmetic. Index object used in python built-in set datastructure → such as unions, intersections, differences. and other combinations.

```
In[35]: ind A = pd.Index([1, 3, 5, 7, 9])
        ind B = pd.Index([2, 3, 5, 7, 11])

In[36]: ind A & ind B   # intersection
Out[36]: Int64 Index([3, 5, 7], dtype='int64')

In[37]: ind A | ind B   # union
Out[37]: Int 64 Index([1, 2, 3, 5, 7, 9, 11], dtype='int 64')

In[38]: ind A ^ ind B   # symmetric difference
Out[38]: Int64 Index([1, 2, 9, 11], dtype='int 64')
```

These operations may also be accessed via object method eg)

↳ ind A. intersection (ind B)

Data Indexing and Selection

Data Selection In Series.

Series object act like → one dimensional NumPy Array
→ Standard Python Dictionary.

Series object provides Dictionary style Data selection Mechanism.

(1) Series as Dictionary :- Like a dictionary, Series object provides a mapping from a collection of keys to a collection of values.

```
In[1]: import pandas as pd
       data = pd.series([0.25, 0.5, 0.75, 1.0],
                        index = ['a', 'b', 'c', 'd'])
       data
out[1]:   a   0.25
          b   0.5
          c   0.75
          d   1.0
          dtype: float64
```

(index positions annotated: 0, 1, 2, 3 and -4, -3, -2, -1)

var[start Index : end Index]
value Included — value excluded

: : 2 → Add index +2

```
In [2]: data['b']
out[2]: 0.5
```

We can also use dictionary like Python expressions and methods to examine the key/indices and values.

```
In[3]: 'a' in data.
out[3]: True
In [4]: data.keys()
```

```
out[4]: Index(['a', 'b', 'c', 'd'], dtype = 'object')
In[5]: list (data.items())
out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Series objects :

It can be modified with a dictionary like syntax. Dictionary is extended by new key. In the same way Series can be extended by assigning a new index value.

```
In [6]: data['e'] = 1.25
        data
Out[6]: a   0.25
        b   0.50
        c   0.75
        d   1.00
        e   1.25
        dtype: float64
```

(ii) Series as One-dimensional Array :-

Series provides Array style item selection mechanisms. ie)
* slices
* masking
* fancy Indexing.

```
In[7]: # slicing by Explicit Index
       data['a':'c']

out[7]:  a   0.25
         b   0.50
         c   0.75
         dtype : float64
```

In [8]: # slicing by Implicit integer index
data[0:2]   default index value provided

out[8]: a   0.25
b   0.50
dtype: float64

In [9]: # masking
data[(data > 0.3) & (data < 0.8)]

out[9]: b   0.50
c   0.75
dtype: float64

In [10]: # fancy Indexing
data[['a', 'e']]

out[10]: a   0.25
e   1.25
dtype: float64

When you are slicing with an explicit index (data['a':'c']), the final index is included in the slice.

When you are slicing with an Implicit index data[0:2], the final index is excluded from the slice.

Indexers, loc, iloc and ix
↳ Locate row → Attribute to return one or more specified Rows

⇒ While a Indexing operations data[1] will use explicit indices.

⇒ While a slicing operation like data[1:3] will use implicit index.

```
In [11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
Out[11]: 0 1    a
         1 3    b
         2 5    c
         dtype: object

In [12]: # explicit index when indexing.
         data[1]
Out[12]: 'a'

In [13]: # implicit Index when slicing
         data[1:3]
Out[13]:  3    b
          5    c
          dtype: object
```

Indexing attributes

→ loc

→ iloc

→ ix

First loc attribute allows indexing and slicing that always references the explicit index (include final Index)

```
In [14]: data.loc[1]
Out[14]: 'a'

In [15]: data.loc[1:3]
Out[15]:  1    a
          3    b
          dtype: object
```

The iloc attribute allows indexing and slicing that always references the implicit Python-style index. (Exclude final Index)

```
In[16]: data.iloc[1]
```

```
out[16]: 'b'

In[17]: data.iloc[1:3]

out[17]:  3   b
          5   c
          dtype: object
```

Third Indexing attribute, (ix) is a hybrid of two.

In Python Code. → "Explicit is better than Implicit"

The explicit nature loc and iloc, make very useful in maintaining clean and Readable Code

Data Selection in DataFrame.

⇒ Data frame acts like a two dimensional or structured Array or in other way like a dictionary.

(i) Dataframe as a dictionary :-

Dataframe as a dictionary of related Series acts. Let's ~~Our~~ Consider the eg,

```
In [18]: area = pd.series({'California': 423967,
         'Texas': 695662, 'New York': 141297, 'Florida': 170312,
         'Illinois': 149995})

pop = pd.Series({'California': 38332521, 'Texas':
      26448193, 'New York': 19651127,
      'Florida': 19552868, 'Illinois': 12882135})

data = pd.Dataframe({'area': area,
       'pop': pop})

data
```

out[18]:

	area	Pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

The individual series that makes up the columns of the data frame can be accessed via dictionary - style indexing of the column name.

In[19]: data['area']

out[19]:

California	423967
Florida	170312
Illinois	149995
NewYork	141297
Texas	695662

Name: area, dtype: int64

equivalently, we can use attribute - style access with column names.

In [20] data.area

out [20]:

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662

Name: area, dtype: int64

Out[18]:

	area	Pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

The individual series that makes up the columns of the data frame can be accessed via dictionary - style indexing of the column name.

In[19]: data['area']

Out[19]:

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662

Name : area, dtype : int64

equivalently, we can use attribute - style access with column names.

In[20] data.area.

Out[20]:

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662.

Name : area, dtype : int64

Attribute style column access have the same object as the dictionary-style Access.

In [21]: data.area is data["area"]

Out[21]: True

Attribute style access is not possible, when the column names are not strings, Column name conflicts with the method of the dataframe.

In [22]: data.pop is data['pop']

out[22]: false.

Like series object

Dictionary style syntax add a new column

In [23]: data[density]= data['pop'/data['area']

data

out[23]:

		area	pop	density
California	0	423967	38332521	0.9 90.413926
Florida	1	170312	19552860	1.14 114.806121
Illinois	2	149995	12882135	0.8 85.883763
New York	3	141297	19651127	1.39 139.076746
Texas	4	695662	26448193	3.8 38.018740

(ii) Dataframe As Two-dimensional Array.

Dataframe as a two dimensional Array

In [24]: data.values.

out[24]: Array([[4.23, 3.83, 9.04],
1.70, 1.955, 1.14],
1.49, 1.28, 8.58],
1.41, 1.96, 1.39],
6.95, 2.64, 3.8]])

Array like observations on Data frame.

Transpose full dataframe

↳ swap Rows and columns

In [25]: data.T

out[25]:

	California	Florida	Illinois	Newyork	Texas
Area	4.23	1.70	1.49	1.41	6.95
pop	3.83	1.95	1.28	1.96	2.64
density	9.04	1.14	8.58	1.39	3.80

Indexing of Columns to acces * a row in a Array.

In [26]: data.values [0]

out[26]: array ([4.23 , 3.83, 9.04])

Passing a single 'Index' to a Dataframe accesses a column.

In [27]: data['area']

out [27]:

California	423
Florida	170
Illinois	149
NewYork	141
Texas	695

Name : area, dtype : int64

Array style Indexing,

Uses loc, iloc, ix

iloc Indexer uses implicit python like style index

In [28]: data. iloc [:3, :2]

out [28] :

	area	pop
California	423	383
Florida	170	195
Illinois	149	128

In [29]: data.loc [:'Illinois', :'pop'] *explicit Index*

out [29] :

	area	pop
California	423	383
Florida	170	195
Illinois	149	128

ix indexer allows a hybrid of two approaches

In [30]: data.ix [:3, :'pop']

out [30]:

	area	pop
California	423	383
Florida	170	195
Illinois	149	128

Numpy style data acces patterns also used in indexers. loc indexer. combine masking and fancy indexing.

In [31]: data.loc [data.density >1. , ['pop', 'dens

out [31] :

	pop	density
Florida	195	1.14
NewYork	196	1.39

Additional Indexing Conventions :

In [33] : data ['Florida' : 'Illinois']

out [33] :

	area	pop	density
Florida	170	195	1.14
Illinois	149	128	0.8

First, while indexing refers to columns,
slicing refers to Rows.
⇒ slices refer to row by number rather than by Index.

In[34]: data[1:3]

Out[34]:

	area	pop	density
Florida	170	195	1.14
Illinois	149	128	0.8

Operations on Data In Pandas:-

NumPy has the ability to perform basic arithmetic operations (addition, subtraction, Multiplication) and with more sophisticated operations (trigonometric functions, exponential, logarithmic functions

Pandas inherits this functionality from numpy, and introduce use ufuncs

↙
"Computation on NumPy Arrays: Universal Functions"

Pandas Includes

For Binary Operations such as addition, Multiplication Pandas will automatically align indices when passing the object to the ufunc

For Unary Operations like negation and trigonometric functions pandas use preserve index and Column labels

(1) Ufuncs: Index Preservation:-

Pandas is designed to work with NumPy, NumPy ufunc will work on Pandas Series and Dataframe objects.

Let's us start by defining a simple Series and Dataframe on which to demonstrate this,

```
In [1]: import pandas as pd
        import numpy as np
In [2]: rng = np.random.RandomState(42)
        ser = pd.Series(rng.randint(0, 10, 4))
        ser
Out[2]: 0    6
        1    3
        2    7
        3    4
        dtype: int64
```

random no. generator

Create a one dimensional array of 4 values

ndim -> No. of dimensions.
shape -> size of each dimension.
size - total size of the array.

```
In [3]: df = pd.Dataframe(rng.randint(0,10,(3,4)),
                          columns = ['A','B','C','D'])
        df
Out[3]:    A  B  C  D
        0  6  9  2  6
        1  7  4  3  7
        2  7  2  5  4
```

Create 3×4 array of Random Integers in the interval [0,10]

randint ↳ Random integer

If we apply a NumPy ufunc on either of these objects. The results will be another pandas object with the indices preserved

```
In [4]: np.exp(ser)
Out[4]: 0     403.428793
        1      20.085537
        2    1096.633158
        3      54.598150
        dtype. float64
```

Also performs the complex calculation :

np.sin(df * np.pi/4)

UFuncs : Index Alignment .

For binary operations on two series or Dataframe object, Pandas will align indices in the process of performing the operation.

Index Alignment In Series :-

Suppose we are combining two different data sources, and find only the top three US states by area and top three US states by Population.

```
In [6]: area = pd.Series ({'Alaska' : 1723337,
            'Texas' : 695662, 'California' : 423967},
            name = 'area')

Population = pd.Series ({'California' : 38332521,
            'Texas' : 26448193, 'New York' : 19651127},
            name = 'population')
```

To compute the population density, we divide these data sources.

```
In [7] : population / area.
out [7] :   Alaska        NaN  → Not a Number
            California    90.413926
            NewYork       NaN
            Texas         38.018740
            dtype : float64.
```

↳ Any item does not have an entry is form is marked with NaN.

- Set Arithmetic.

The resulting array contains the union of indices of the two input arrays. By using a standard python set Arithmetic.

```
In[8]: area.index | population.index
out[8]: Index(['Alaska', 'California', 'New York',
               'Texas'], dtype='object')
```

⇒ Index Matching is implemented by Python's built-in arithmetic expressions.

⇒ Any missing values are filled with NaN by default

```
In [9]: A = pd.series([2,4,6], index=[0,1,2])
        B = pd.series([1,3,5], index=[1,2,3])
        A+B
out [9]:  0    NaN
          1    5.0
          2    9.0
          3    NaN
        dtype: float64
```

Fill Value → By using appropriate Object Methods in place of the operators.

Calling A.add(B) —Equivalent to→ A+B.

```
In[10]: A.add(B, fill_value=0)
out[10]:  0   2.0
          1   5.0
          2   9.0
          3   5.0
        dtype: float64
```

Index Alignment In Data Frame.

Alignment can be taken in both columns and indices when you are performing operations on Data Frames.

```
In [11]: A = pd.DataFrame(rng.randint(0,20,(2,2)),
                          columns = list('AB'))
         A
Out[11]:     A    B
         0   1   11
         1   5    1
```

```
In [12]: B = pd.Dataframe(rng.randint(0,10,(3,3)),
                          columns = list('BAC'))
         B
Out[12]:     B   A   C
         0   4   0   9
         1   5   8   0
         2   9   2   6
```

```
In [13]: A+B
Out[13]:      A      B     C
         0   1.0   15.0   NaN
         1  13.0    6.0   NaN
         2   NaN    NaN   NaN
```

```
In [14]: fill = A.stack().mean()
         A.add(B, fill_value = fill)
```

out [14]:

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

7.91 with the mean of all values in A (which we compute by first stacking the rows of A)

Mapping between Python operators and Pandas Methods

Python operator	Pandas Methods (s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Ufuncs: operations Between Dataframe and Series:

When we are performing operations between Dataframe and Series, are similar to the operations between two-dimensional and one-dimensional NumPy Array.

We find the difference of a two-dimensional array and one of its rows.

```
In [15]: A = rng.randint(10, size=(3,4))
         A
out[15]: array([[3, 8, 2, 4],
                [2, 6, 4, 8],
                [6, 1, 3, 8]])

In[16] : A - A[0]
out[16]: array([[0, 0, 0, 0],
                [-1, -2, 2, 4],
                [3, -7, 1, 4]])
```

Subtraction between a two-dimensional array and one of its rows is applied row-wise. In Pandas, row-wise operation by default.

```
In [17]: df = pd.DataFrame(A, columns=list('QRST'))
         df - df.iloc[0]
out[17]:    Q   R   S   T
         0  0   0   0   0
         1  -1  -2  2   4
         2  3   -7  1   4
```

operate column-wise → specify axis keyword.

```
In[18]: df.subtract(df['R'], axis=0)
out[18]:    Q   R   S   T
         0  -5  0   -6  -4
         1  -4  0   -2  2
         2  5   0   2   7
```

Handling Missing Data:

The difference between data found in tutorials and data in the real world is Real world data is rarely clean and homogeneous - Because many datasets have some amount data missing. different data sources may indicate missing data.

In Python, built in pandas tools for handling missing data. Eg: null, NaN, NA values.

Trade-off In Missing Data Conventions:-

To indicate the presence of missing data in a table or Dataframe two strategies are there,

1) Using a mask that globally indicates missing values.
2) Choosing a sentinel value that indicates a missing entry

In the masking Approach:-

The mask might be entirely separate Boolean Array use one bit in the data representation to locally indicate the null status of a value.

In the sentinel Approach:-

The sentinel value could be some data-specific convention, such as indicating a missing integer with -9999.

These two approaches have some trade-off:-

* Use of separate mask array requires allocation of additional Boolean Array, which add overhead in both storage and computation.
* A sentinel value reduce the range of value that can be represented, may require extra logic in CPU

Missing Data In Pandas: (58)

⇒ Pandas have R's lead specifying bit patterns for each individual data type to indicate nullness.
Eg) while R ~~has~~ ~~single~~ contains four basic datatypes.
~~Python~~ Pandas chose to use sentinels for missing data, and choose to use two already existing Python null values,

* Floating Point NaN value.
* Python None object.

None: Pythonic Missing Datas:-

The first sentinel value is used by Python is None. Python singleton object is used for missing data in python code. Because None is a python object, It cannot be used in arbitrary Numpy / Pandas Arrays, but only in arrays with data type object. (ie. Array of python objects)

```
In[1]: import numpy as np
       import pandas as pd

In[2]: vals1 = np.array([1, None, 3, 4])
       vals1

out[2]: array([1, None, 3, 4], dtype = object)
```

⇒ The sum() or min() aggregations are not performed across an array with a None value

It will produce error

NaN: Missing Numerical Data :-

The other missing data representation, NaN (Not a Number) is different, It is a special floating point value recognised by all systems that use the standard IEEE floating-point representation.

```
In[5]: vals2 = np.array([1,np.nan, 3, 4])
       vals2.dtype
out[5] dtype('float64')
```

Unlike object from before, This array supports fast operations pushed into compile code.

Result of arithmetic with NaN will be another NaN

```
In[6]: 1+np.nan
out[6]: nan
In[7]: 0*np.nan
out[7]: nan
```

They don't result an error

```
In[8]: val2.sum(), val2.min(), val2.max()
out[8]: (nan, nan, nan)

In[9]: np.nansum(val2), np.nanmin(val2),
       np.nanmax(val2)
out[9]: (8.0, 1.0, 4.0)
```

NaN is specifically a floating point value.

There is no equivalent NaN value for integers, strings or other types.

NaN and None In Pandas :-

NaN and None both have their place, and

Pandas is built to handle the two of them nearly interchangeably, converting between them.

```
In [10]: pd.series([1, np.nan, 2, None])

out[10]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
       dtype: float64

In[11]: x = pd.series(range(2), dtype = int)
        x

out[11]: 0    0
         1    1
       dtype: int64

In [12]: x[0]= None
         x

out[12]  0    NaN
         1    1.0
       dtype: float64
```

Pandas will automatically type cast value If NA values are present.

Notice that in addition to casting the integer array to floating point, Pandas automatically converts None to a NaN value.

Operating On Null Values:

Pandas treat NaN and None as essentially interchangeable for indicating missing or null values.

There are several useful methods for

detecting, removing and replacing null values in Pandas data structures.

isnull()

Generate a Boolean Mask indicating missing values.

notnull()

opposite of isnull()

dropna()

Return a filtered version of the data.

fillna()

Return a copy of the data with missing values filled or imputed.

Detecting Null values

Pandas data structure have useful methods for detecting null data: isnull() and notnull(). → It will return a boolean mask to the data

Either one will return a Boolean mask over the data.

```
In[13]: data = pd.series ([1, np.nan, 'hello', None])
In[14]: data.isnull()
out[14]: 0   False
         1   True
         2   False
         3   True
dtype: boo
```

Dropping Null Values

In addition to the masking used before, there are convenience methods,

dropna() → (which removes NA values)

fillna() → (which fills in NA values)

For a Dataframe ~~series~~

```
In[17]: df = Dataframe([[1, np.nan, 2],
                        [2, 3, 5],
                        [np.nan, 4, 6]])
df
Out[17]:
     0    1    2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

We cannot drop single values from a Dataframe. We can only drop full rows or full columns

By default, dropna() will drop all rows in which any null value is present.

```
In[18]: df.dropna()
out[18]:    0    1    2
        1  2.0  3.0  5
```

Alternatively, axis = 1 drops all columns containing a null value.

```
In[19]: df.dropna(axis = 'columns')
out[19]:    2
         0  2
         1  5
         2  6
```

In [20]: df[3] = np.nan
df

out[20]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [21]: df.dropna(axis='columns', how='all')

out[21]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

The default is how='any'.
such that any row or column (depending on the axis keyword) containing a null value will be dropped.

You can also specify how='all', which can drop rows/columns that are all null values.

Filling Null Values :-

Rather than dropping NA values, you have to replace them with a valid value. This value might be a single number like zero

In[2]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data

```
out[23]:  a    1.0
          b    NaN
          c    2.0
          d    NaN
          e    3.0
    dtype: float64.
```

We can fill NA entries with a single value, such as zero:

```
In [24]:  data.fillna(0)

out[24]   a.   1.0
          b    0.0
          c    2.0
          d    0.0
          e    3.0
     dtype: float64
```

We can specify a forward-fill to propagate the previous value forward.

```
In[25]: # forward-fill.
        data.fillna(method='ffill')

out[25]:  a   1.0
          b   1.0
          c   2.0
          d   2.0
          e   3.0
     dtype: float64
```

or we can specify a back-fill to propagate the next values backward.

```
In [26]: # back-fill
         data.fillna(method = bfill)
out [26]: a   1.0
          b   2.0
          c   2.0
          d   3.0
          e   3.0
dtype: float64
```

Hierarchical Indexing :-

⇒ In higher dimensional data storage, data is indexed by more than one keys.

⇒ For handling higher dimensional data (three dimensional and four dimensional data) Hierarchical Indexing is used. It is also known as Multi-Indexing.

⇒ To use multiple Index levels within a single Index.

We begin with the std Imports

```
In [1]: import pandas as pd
        import numpy as np
```

A Multiply Indexed series :-

→ It represent two dimensional data within a one-dimensional series.

→ series of data where each point has a character and Numerical key.

Bad way :-

To track data about states from two different years, simply Use python tuples as keys.

```
In [2]: index = [('california', 2000), ('california', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [338 , 372 ,
               189 , 193 ,
               208 , 251]
pop = pd. series (populations, index= index)
pop

out [2]:
(california , 2000)   338
(california , 2010)   372
(New York, 2000)      189
(New York , 2010)     193
(Texas, 2000)         208
(Texas, 2010)         251
dtype : int64
```

Selection process is very difficult in multiple index.

The better way : Pandas MultiIndex.

Pandas provides a better way. Pandas MultiIndex type gives us the type of operations we can create a multiIndex from the tuples as follows:

tuple → Used to store multiple items in a single variable.

```
In[5]: index = pd.MultiIndex.from_tuples(index)
       index

out[5]: MultiIndex(levels = [['california', 'New York',
                   'Texas'], [2000, 2010]],
             labels = [[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

MultiIndex contain multiple Levels of Indexing. In this case, state names and the years, as well as multiple Labels for each data point which encode these levels.

ReIndex our series with the MultiIndex, Result is hierarchical Representation of the data,

```
In [6]: pop = pop.reindex(index)
        pop

out[6]: California   2000   33871648
                     2010   37253956
        New York     2000   189
                     2010   193
        Texas        2000   208
                     2010   251

dtype: int64.
```

First two columns Rep → MultiIndex Values

Third Column → Shows the data.

MultiIndex as Extra dimension :-

We can easily store the same data using a simple dataframe with index & column Labels. Pandas is built with this equivalence in mind.

Unstack() Method → Convert a multiply-indexed Series into a conventionally indexed Dataframe.

```
In[8]: pop_df = pop.unstack()
       pop_df

out[8]:              2000        2010
        California   338~~71648~~  372
        NewYork      189         193
        Texas        208         251
```

~~Un~~ stack(): Method provides the opposite operation

```
In [9]: pop_df.stack()
out[9]:  California   2000   338
                       2010   372
         NewYork       2000   189
                       2010   193
         Texas         2000   208
                       2010   251
         dtype: int64
```

Each extral level in a multi Index represent an extra dimensions of data. When compared to the

series, multiIndex with the dataframe that has process easy to adding another column.

```
In[10]: pop-df = pd.Dataframe({'total': pop,
                               'under18': [926, 928,
                                           468, 431,
                                           590, 687]})
pop-df
```

out[10]:

		total	Under18
California	2000	338	926
	2010	372	928
NewYork	2000	189	468
	2010	193	431
Texas	2000	208	590
	2010	251	687

Pandas allows easily and quickly manipulate and explore even high-dimensional data.

Methods of MultiIndex Creation

Most common way to construct a multiply indexed series or Dataframe is to simply pass a list of two or more index arrays to the constructor.

```
In[12]: df = pd.Dataframe(np.random.rand(4, 2),
                          index = [['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns = ['data1', 'data2'])
```

out[12]:

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

The word of creating the MultiIndex is done in the background.

If you pass the dictionary with appropriate type tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default.

```
In[13]: data = {('california', 2000): 338,
                ('california', 2010): 372,
                ('Texas', 2000): 208,
                ('Texas', 2010): 251,
                ('NewYork', 2000): 189,
                ('NewYork', 2010): 193}

Out[13]: California  2000   338
                     2010   372
         NewYork     2000   189
                     2010   193
         Texas       2000   208
                     2010   251
         dtype: int64.
```

Explicit MultiIndex Constructors :-

We can construct the MultiIndex from a list of arrays, giving the index values within each level.

```
In[14]: pd.MultiIndex.from([['a','a','b','b'],
                            [1,2,1,2]])
Out[14]: MultiIndex(levels = [['a','b'],[1,2]],
                    labels = [[0,0,1,1],[0,1,0,1]])
```

We can construct Multi Index from a list of tuples.

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2),
                                   ('b', 1), ('b', 2)])
out[15]: MultiIndex(levels = [['a', 'b'], [1, 2]],
                    labels = [[0, 0, 1, 1], [0, 1, 0, 1]])
```

We can construct MultiIndex from a Cartesian product of single Indices.

```
In [16]: pd.MultiIndex.from_product([['a', 'b'],
                                     [1, 2]])
out [16]: MultiIndex(levels = [['a', 'b'], [1, 2]],
                     Labels = [[0, 0, 1, 1], [0, 1, 0, 1]])
```

We can create or construct MultiIndex directly using its internal encoding by passing levels (a list of lists containing available index values for each level) & Labels (a list of lists that reference these labels)

MultiIndex Level names :-

We can pass convenient names to the level of the multiIndex. ie) By passing the names argument to any of the above multiIndex.

```
In[18]: pop.index.names = ['state', 'year']
        pop.
out[18]: state      year.
```

California	2000	338
	2010	372
New York	2000	189
	2010	193
Texas	2000	208
	2010	251

dtype: int64.

Indexing and slicing a MultiIndex:

Indexing and slicing on a MultiIndex,

Multiply Indexed Series:-

consider multiply Indexed Series of state populations.

In [21]: pop

Out[21]:

state	year	
California	2000	338
	2010	372
New York	2000	189
	2010	193
Texas	2000	208
	2010	251

dtype: int64

⇒ We can access & single elements by indexing with multiple terms:

In [22]: pop['California', 2000]

Out[22]: 338

⇒ MultiIndex also supports partial indexing

Indexing just one of the levels in the index.

result ⇒ is another series, with the lower level indices.

In [23]: pop ['california']

out [23]: Year

2000	338
2010	372

dtype : int64

⇒ MultiIndex also supports partial slicing

In[24]: pop.loc ['California' : 'NewYork']

out[24]: state year

California	2000	338
	2010	372
NewYork	2000	189
	2010	193

dtype : int64

With sorted indices, we can perform partial Indexing on lower levels by passing an empty slice in the first Index.

In [25]: pop [:, 2000]

out[25]: state

California	338
NewYork	189
Texas	208

dtype : int64

other types of Indexing and selection :

selection based on Boolean masks.

```
In [26]: pop[pop > 22000]
out [26]: State        Year
          California   2000     338
                       2010     372
          Texas        2010     251
dtype : int64
```

selection based on Fancy Indexing.

```
In [27]: pop [['California', 'Texas']]
out [27]: State        Year
          California   2000     338
                       2010     372
          Texas        2000     208
                       2010     251
dtype : int64
```

Multiply Indexed Dataframes :-

Multiply Indexed Dataframe behaviour in a similar manner as a series.

In [28]: health_data

out [28]:

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	38.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

We can access Guido's heart rate data with simple operation.

```
In [29]: health_data ['Guido', 'HR']

Out[29]: Year    Visit
         2013    1        32.0
                 2        50.0
         2014    1        39.0
                 2        48.0

Name: (Guido, HR), dtype: float64
```

We can also use loc and iloc, ix operations

```
In [30]: health_data.loc [:2, :2]

Out[30]  Subject      Bob
         type         HR    Temp
         Year  Visit
         2013  1      31.0  38.7
               2      44.0  37.7
```

```
In [31]: health_data.loc [:, ('Bob', 'HR')]

Out[31]: Year    Visit
         2013    1        31.0
                 2        44.0
         2014    1        30.0
                 2        47.0

Name: (Bob, HR), dtype: float64
```

Data Aggregation on Multi-Indices:

Pandas has built-in data-aggregation methods, such as mean(), sum(), & max()

For Hierarchically indexed data, these can be passed a level parameter that controls

In [43]: health_data

out [43]:

Mean() -> Average two visits of each year.

```
In [44]: data_mean = health_data.mean(level='year')
         data_mean
```

out[44]:

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	37.5	38.2	41.0	35.85	32.0	36.95
2014	38.5	37.6	43.5	37.55	56.0	36.70

Combining Dataset : Concat and Append.

⇒ Data comes from different data sources.

operations involve ⇒ Concatenation of two different data set are more complicated.

⇒ Pandas includes functions and methods ⇒ that make sort of data wrangling fast and straightforward.

⇒ Simple concatenation of series and Dataframes with (pd.concat) function

We begin with the standard import

```
In [1]: import pandas as pd
        import numpy as np
```

We define this function which creates a Dataframe on a particular form.

```
In [2]: def make_df (cols, ind): """Quickly make a Dataframe"""
        data = {c : [str(c) + str(i) for i in ind]
                      for c in cols}
        return pd.DataFrame (data, ind)

        # example Dataframe
        make_df ('ABC', range(3))
```

out [2]:

	A	B	C
	A_0	B_0	C_0
1	A_1	B_1	C_1
2	A_2	B_2	C_2

Recall: Concatenation of Numpy Arrays :-

Concatenation of Series and Dataframe objects similar to the concatenation of Numpy Arrays, done by (np.concatenate).

Combine the Concatents of two or more arrays into a single Array.

```
In [4]: x = [1,2,3]
        y = [4,5,6]
        z = [7,8,9]
        np.concatenate([x,y,z])
out [4]: array([1,2,3,4,5,6,7,8,9])
```

The first argument is a list or tuple of arrays to ~~arrays~~ concatenate.

Additionally, axis keyword allows to specify the axis along which result to be concatenated.

```
In [5]: x = [[1,2],
             [3,4]]
        np.concatenate([x,x], axis=1)
out [5]: array([[1,2,1,2],
                [3,4,3,4]])
```

Simple Concatenation with pd.concat:

Pandas has a function, pd.concat(), similar to np.concatenate but pandas contain several options with this.

```
# signature in Pandas V0.18
pd.concat(objs, axis=0, join='outer', join_axes=None,
          ignore_index=False, keys=None, levels=None, names=None,
          verify_integrity=False, copy=True)
```

pd.concat() → It can be used for a simple concatenation of Series or Dataframe objects.

np.concatenate() → It can be used for simple concatenation of arrays.

```
In[6]: ser1 = pd.series['A','B','C'], index = [1,2,3])
       ser2 = pd.series['D','E','F'], index = [4,5,6])
              pd.concat([ser1, ser2])

out[6]: 1   A
        2   B
        3   C
        4   D
        5   E
        6   F

        dtype : object
```

- Concatenate higher dimensional ~~data~~ objects, such as Dataframes.

```
In[7]: df1 = make_df ('AB', [1,2])
       df2 = mak_df ('AB', [3,4])
       print (df1);
       print (df2);
       print (pd. concat([df1, df2])) df1    df2    pd.concat([df1,df2])
```

	A	B
1	A_1	B_1
2	A_2	B_2

	A	B
3	A_3	B_3
4	A_4	B_4

	A	B
1	A_1	B_1
2	A_2	B_2
3	A_3	B_3
4	A_4	B_4

By default, the concatenation takes place row-wise within the Dataframe (i.e. axis = 0)

Like np.concatenate, pd.concat allows specification of the axis along with the concatenation takes place.

```
In [8]: df3 = make_df ('AB', [0, 1])
        df4 = make_df ('CD', [0, 1])
        print (df3);
        print(df4);
        print (pd.concat ([df3, df4], axis = 'col'))
```

df3   df4   pd.concat ([df3, df4], axis = 'col')

	A	B
0	A0	B0
1	A1	B1

	C	D
0	C0	D0
1	C1	D1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

axis = 1 Here we use axis = 'col'

Concatenation with joins :-

⇒ Concatenating Dataframes with shared column names

⇒ pd.concat offers several options in this case.

Concatenation of two dataframes, which have some columns in common.

```
In [13]: df5 = make_df ('ABC', [1, 2])
         df6 = make_df ('BCD', [3, 4])
print (df5) ; print (df6); print (pd.concat ([df5, df6])
```

df5
df6
pd.concat ([df5, df6])

	A	B	C
1	A_1	B_1	C_1
2	A_2	B_2	C_2

	B	C	D
3	B_3	C_3	D_3
4	B_4	C_4	D_4

	A	B	C	D
1	A_1	B_1	C_1	NaN
2	A_2	B_2	C_2	NaN
3	NaN	B_3	C_3	D_3
4	NaN	B_4	C_4	D_4

By default, no entries are filled with NA values.

Several options are there, join & join_axes parameter.

Be default → join='outer'

we may change join = 'inner'

In [14]: print (df5); print (df6); print (pd. concat([df5, df6], join='inner'))

df5
df6
pd.concat ([df5, df6], join='inner')

	A	B	C
1	A_1	B_1	C_1
2	A_2	B_2	C_2

	B	C	D
3	B_3	C_3	D_3
4	B_4	C_4	D_4

	B	C
1	B_1	C_1
2	B_2	C_2
3	B_3	C_3
4	B_4	C_4.

Append() Method :-

Concatenation is common in Series and Dataframe objects. Rather than calling pd.concat([df1, df2]), simply call df1.append(df2):

In [16]: print (df1); print (df2);
print (df1.append (df2))

df1

	A	B
1	A_1	B_1
2	A_2	B_2

df2

	A	B
3	A_3	B_3
4	A_4	B_4

df1.append (df2)

	A	B
1	A_1	B_1
2	A_2	B_2
3	A_3	B_3
4	A_4	B_4

Append() method in pandas doesnot modify the original object. It creates a new object with the combined data.

Combining Datasets: Merge and Join :-

⇒ Pandas provides high performance in memory Join and merge operations

pd.merge() function

join() function.

Categories of Joins :-

pd.merge() function implements a number of types of Joins!

- one_to_one
- many-to-one
- many-to-many Joins.

one-to-one Joins :-

one-to-one Join similar to column-wis concatenation.

In[2]:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
    'hire_date': [2004, 2008, 2012, 2014]})
print(df1); print(df2)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hir_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014.

To combine this information into a single Data frame, we can use the pd.merge() function.

In[3]: df3 = pd.merge(df1, df2) df3

Out[3]:

	employee	group	hire date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014.

Aggregation and Grouping

⇒ In Large dataset, the datas are summarized or aggregated for efficiency.

↳ aggregation → It will return single number for a Large dataset

⇒ Some aggregations → sum(), mean(), median(), min(), max()

Simple Aggregation In Pandas:-

⇒ For a pandas Series the aggregates return a single value.

```
In [4]: rng = np.random.RandomState(42)
        ser = pd.Series(rng.rand(5)) ser

Out[4]: 0    0.3
        1    0.9
        2    0.7
        3    0.5
        4    0.1
    dtype: float64

In [5]: ser.sum()
Out[5]: 2.5

Out[6]: ser.mean()
Out[6]: 0.5
```

For a Dataframe, By default the aggregation can be taken in each column.

```
In[7]: df = pd.Dataframe({'A': rng.rand(5),
                          'B': rng.rand(5)})

df
```

```
Out [7]:        A            B
         0   0.155995      0.01
         1   0.05          0.9
         2   0.8           0.8
         3   0.6           0.2
         4   0.7           0.1

In [8]: df.mean()

Out[8]: A   0.45
        B   0.4.

        dtype: float64.

In [9]:  df.mean (axis = 'columns')

Out [9]:  0   0.08
          1   0.5
          2   0.8
          3   0.4
          4   0.4.

        dtype: float64.
```

Some other built-in Pandas Aggregations.

Aggregation	Description
count()	Total number of items
first(), last()	First and Last item
mean(), median()	Mean and median
min(), max()	minimum and maximum
std(), var()	Standard deviation and Variance
mad()	Mean absolute deviation

prod() Product of all items

sum() sum of all items.

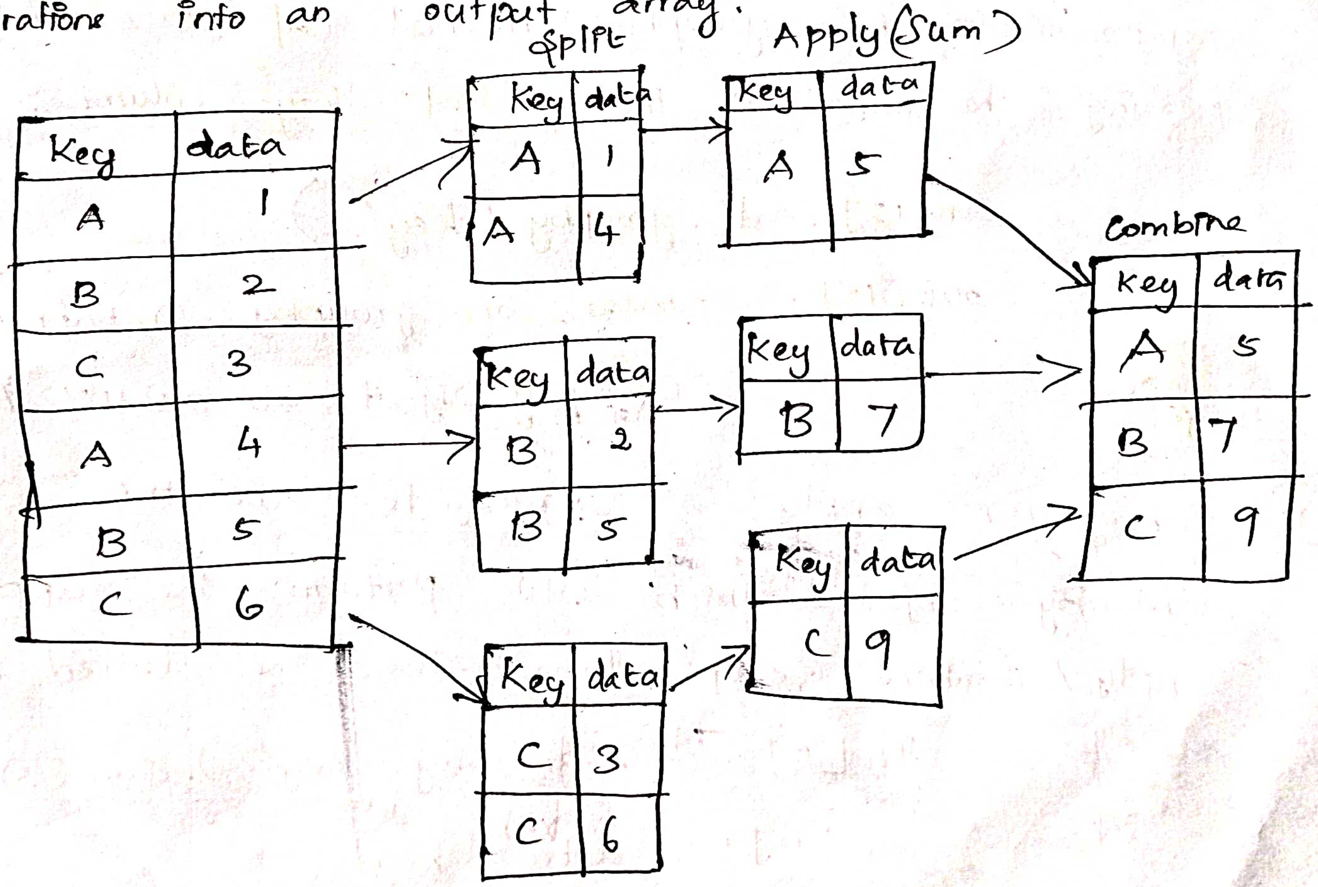
Group By : Split, Apply, Combine.

GroupBy is accomplished by the step.

* Split step involves breaking up and grouping a dataframe depending on the value of the specified key.

* Apply step involves computing some function, usually an 1) aggregate 2) transformation, 3) filtering within the individual groups.

* The combine step merges the results of these operations into an output array.



Visual Representation of GroupBy Operation.

By creating the input Dataframe :-

```
df = pd.Dataframe ({"key': ['A','B','C','A','B','C'],
        'data': range(6)}, columns = ['key','data'])
```

df

out[11]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

We can compute the most basic split-apply-combine operation with the groupby() method of dataframes, passing the name of the desired key column :-

```
In[12]: df.groupby('key')
out[12]: <pandas.core.groupby.DataframeGroupby Object at 0x117272160>
```

We can apply an aggregate to this Dataframe GroupBy object, which will perform the appropriate apply / combine steps to produce the desired result.

```
In [13]: df.groupby('key').sum()
```

out [13]:

key	data
A	3
B	5
C	7

In [20] : df.groupby('key'). aggregate (~~[np.sum]~~ ~~'data1' : 'min'~~, 'data2' : 'max'})

out[20] :

	Key	data1	data2.
0	A	0	5
1	B	1	0
2	C	14	22.
3			
4			
5			

Filtering :-

A filtering operation allows to drop data's based on the group properties based on the some condition.

```
print(df.groupby('key').filter(filter_func))

def filter_func(x):
        return x['data2'].std() > 4

print(df); print(df.groupby('key').std());
print(df.groupby('key').filter(filter_func))

df.
```

Transformation :-

Transformation return some ~~transferred~~ transformed version of the full data to recombine.

Specifying the split key :-

We split the dataframe on a single Column name. A list, array, series, or index providing the grouping keys. The key can be any series or list with a length matching of the dataframe.

```
In [25]: L = [0, 1, 0, 1, 2, 0]
print(df); print (df. groupby (L).sum())
```

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df. groupby (L) . sum ()

	data1	data2
0	7	17
1	4	3
2	4	7

Pivot tables :-

A pivot table is a similar operation that commonly seen in spreadsheets and other programs that operate on tabular data.

Pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table and provides a multidimensional ~~entri~~ summarization of the data.

Pivot table as essentially a multidimensional version of GroupBy aggregation.

Motivating Pivot tables :-

The database of passengers on the titanic available through seaborn library.

```
In [1]: import numpy as np
        import pandas as pd
        import seaborn as sns
        titanic = sns.load_dataset('titanic')

In [2]: titanic.head()
```