DIGITAL NOTES
OF

DATA VISUALIZATION TECHNIQUES
(R22A6705)

B.TECH IV Year -I SEM

(2025-26)

PREPARED BY

Dr. P.V. Naresh
Associate Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
Sponsored by CMR Educational Society

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

IV Year B. TECH–I-SEM **L/T/P/C**
3/0/0/3

DATA VISUALIZATION TECHNIQUES
(R22A6705)

COURSE OBJECTIVES:

1. To learn different statistical methods for Data visualization.
2. To learn Visulization Technoques .
3. To understand the basics of Python.
4. To understand the usage of the Matplotlib, Seaborn Packages
5. To Learn about Excel and various operations using Excel

UNIT I
Introduction to Data Visualization, Key factors of Data Visualization, Importance of Data Visualization in Business Intelligence, Data Visualization tools and types of data, Data Abstraction: data types, dataset types, Attribute types, Task Abstraction, Four Levels for Validation.

UNIT II
Visualization Techniques: Scalar and point visualizations, – vector visualizations – multidimensional visualizations – Cluster Visualizations – matrix visualization in Bayesian data analysis

UNIT III
Getting Started with Pandas: Arrays and vectorized computation, Introduction to pandas Data Structures, Essential Functionality, Summarizing and Computing Descriptive Statistics. Data Loading, Storage and File Formats. Reading and Writing Data in Text Format, Web Scraping, Binary Data Formats, Interacting with Web APIs, Interacting with Databases Data Cleaning and Preparation. Handling Missing Data, Data Transformation, String Manipulation

UNIT IV (Data Visualization Using Matplotlib)
Data Visualization Tools in Python- Introduction to Matplotlib, Basic plots Using matplotlib, Specialized Visualization Tools using Matplotlib, Advanced Visualization Tools using Matplotlib-Waffle Charts, Word Clouds.

UNIT-V (Working With Excel)
Introduction: Data Analysis, Excel Data analysis. Working with range names. Tables. Cleaning Data. Conditional formatting, Sorting, Advanced Filtering, Lookup functions, Pivot tables, Data Visualization, Data Validation. Understanding Analysis tool pack: Anova, correlation, covariance, moving average, descriptive statistics, exponential smoothing, fourier Analysis, Random number generation, sampling, ttest, f-test, and regression.

TEXT BOOKS:

1. Core Python Programming - Second Edition, R. Nageswara Rao, DreamtechPress.
2. A to Z of MS EXCEL: A Book for Learners & Trainers (MS Excel Comprehensive Guide by Rinkoo Jainn
3. Data Analysis with Excel by Manish Nigam. Bpb Publications
4. KNIME Essentials, by Gábor Bakos, 2013
5. Data Science Tools by Christopher Greco, 2020

REFERENCE BOOKS:

1. Introduction to Data Science a Python approach to concepts, Techniques and Applications, Igual, L;Seghi', S. Springer, ISBN:978-3-319-50016-4.
2. ALL-IN-ONE-EXCEL 2022 Bible for Dummies by Bryant Shelton
3. Excel® 2019 BIBLE BY Michael Alexander, Dick Kusleika
4. Python for Data Analysis by William McKinney, Second Edition, O'Reilly MediaInc.

COURSE OUTCOMES:

At Completion of this course, students would be able to -

1. Apply statistical methods for Data visualization on Various Datasets
2. Understand different Visulization Tecniques.
3. Gain knowledge on various visualization techniques using Python
4. Understand usage of various packages in Python.
5. Understand the concept of Excel, Visualization using Excel

Unit - I

Introduction to Data Visualization, Key factors of Data Visualization, Importance of Data Visualization in Business Intelligence, Data Visualization tools and types of data, Data Abstraction: data types, dataset types, Attribute types, Task Abstraction, Four Levels for Validation.

Introduction

In our increasingly data-driven world, it's more important than ever to have accessible ways to view and understand data. After all, the demand for data skills in employees is steadily increasing each year. Employees and business owners at every level need to have an understanding of data and of its impact.

That's where data visualization comes in handy. With the goal of making data more accessible and understandable, data visualization in the form of dashboards is the go-to tool for many businesses to analyze and share information.

1.1 DATA VISUALIZATION

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data. Additionally, it provides an excellent way for employees or business owners to present data to non-technical audiences without confusion.

Data visualization is the practice of translating information into a visual context, such as a map or graph, to make data easier for the human brain to understand and pull insights from. The main goal of data visualization is to make it easier to identify patterns, trends and outliers in large data sets. The term is often used interchangeably with others, including information graphics, information visualization and statistical graphics.

Data visualization is one of the steps of the data science process, which states that after data has been collected, processed and modelled, it must be visualized for conclusions to be made. Data visualization is also an element of the broader data presentation architecture (DPA) discipline, which aims to identify, locate, manipulate, format and deliver data in the most efficient way possible.

Data visualization is important for almost every career. It can be used by teachers to display student test results, by computer scientists exploring advancements in artificial intelligence (AI) or by executives looking to share information with stakeholders. It also plays an important role in big data projects. As businesses accumulated massive collections of data during the early years of the big data trend, they needed a way to get an overview of their data quickly and easily. Visualization tools were a natural fit.

Visualization is central to advanced analytics for similar reasons. When a data scientist is writing advanced predictive analytics or machine learning (ML) algorithms, it becomes important to visualize the outputs to monitor results and ensure that models are performing as intended. This is because visualizations of complex algorithms are generally easier to interpret than numerical outputs.

Why is data visualization important?

Data visualization provides a quick and effective way to communicate information in a universal manner using visual information. The practice can also help businesses identify which factors affect customer behaviour; pinpoint areas that need to be improved or need more attention; make data more memorable for stakeholders; understand when and where to place specific products; and predict sales volumes.

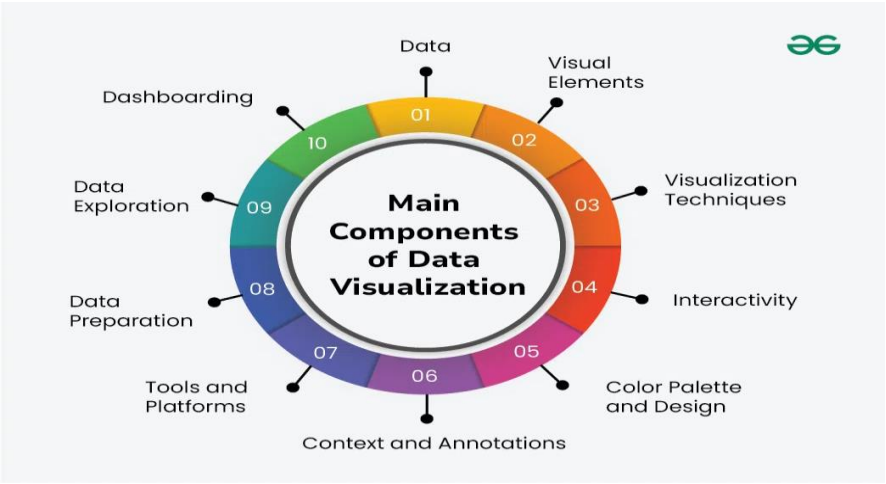
Other benefits of data visualization include the following:

- ✓ the ability to absorb information quickly, improve insights and make faster decisions;
- ✓ an increased understanding of the next steps that must be taken to improve the organization;
- ✓ an improved ability to maintain the audience's interest with information they can understand;
- ✓ an easy distribution of information that increases the opportunity to share insights with everyone involved;
- ✓ eliminate the need for data scientists since data is more accessible and understandable; and
- ✓ an increased ability to act on findings quickly and, therefore, achieve success with greater speed and less mistakes.

1.2 KEYS FACTORS OF DATA VISUALIZATION

In today's world, a huge amount of data is generated every day and it is very important to visualize the data to know its pattern to make important business decisions. At its core, effective data visualization relies on several key components, each playing a crucial role in conveying information accurately and efficiently. These components encompass aspects ranging from the choice of visual representation to the use of colour, interactivity, and storytelling techniques.

Data visualization is a crucial part of data analytics that helps you visualize your data and uncover significant trends and patterns



Understanding these components is essential for creating compelling and informative visualizations that facilitate data-driven decision-making across various domains. In this article, we will learn about ***What are the key components of data visualization?***

Main Components of Data Visualization

1. **Data:** First of all we need lots of data. Data can be of any type like numerical data, text data or geospatial data.
2. **Visual Elements:** For instance, Graphics, charts, Overlays, diagrams, figures, maps, tables and other types of data presentations and resumes that comprise infographics.
3. **Visualization Techniques**: This also includes aspects such as transforming and scaling data, and in some cases indeed selecting the right type of visualization to use.
4. **Interactivity:** Influential aspects of visualization which could include; The blinking bubbles whenever the cursor is over a particular part of the graph, zooming up or down the size of the graph, or even rotating the graph and/or options to have the different groups or categories on or off among others.
5. **Color Palette and Design:** New choices on color utilization, font selection, position of elements in unison with the style of design to enhance the usability, visibility and the aesthetic appeal of visualization.
6. **Context and Annotations:** The additional roles of Titles and Subtitles in the Visualizations and some of the other labels for captions, annotations and legends for further understanding of the analysis.
7. **Tools and Platforms:** Software that falls under "Other Tools and Applications while developing visualization" Some of the prominent ones are Tableau, Power BI and more programming languages and Libraries are D3. js ,Matplotlib and more.
8. **Data Preparation:** Data preparation which involves cleaning of data, that is, data cleaning processing, data selection, data reshaping and data condensation that gets a data in the right form so that it can be analyzed and visualized.
9. **Data Exploration:** Simple methods for analyzing the data for searching the pattern, trend, noise, similarity and correlation other than the techniques.
10. **Dashboarding:** An aggregate display of the two screens to give an overall view of the data collected as well as endlessly monitoring the indicators by having the two tabbed views on the same screen.

1.3 IMPORTANCE OF DATA VISUALIZATION IN BUSINESS INTELLIGENCE

Data visualization is technique for businesses. It helps them understand their data better, make smarter decisions, and stay ahead of the competition. It basically turns boring numbers into easy-to-understand pictures or graphs, helping businesses see what's going on and what they need to do next.

1. Simplifies Data for Better Understanding: Data visualization makes complex and raw data easy to understand by displaying it in the form of graphs and charts. This helps us see patterns and important information more clearly.

2. Help Us Make Better Decisions: When we can see our data in pictorial form or graphs and chart, it's easier to make smart decisions quickly.

3. Helps In Visualizing Data Easily: Even the people who are not experts in data can understand data easily with the help of visualization. Non-technical people can also understand the data easily with the help of visualization tools, in the form of graphs and charts.

4. Makes Data More Interesting: Visualizations make data more interesting and fun to look at, which encourages more people to use data to help their work.

5. Helps in Tracking Progress: Visual dashboards keeps the track of our progress. They help us see if we're reaching our goals by giving us a clear picture of our progress.

1. 4 DATA VISUALIZATION TOOLS AND TYPES OF DATA

Data visualization tools are cloud-based applications that help you to represent raw data in easy-to-understand graphical formats. You can use these programs to produce customizable bar charts, pie charts, column charts, and more

There are many tools that are used for data visualization. Some of the tools are discussed below

1) Power BI

Power BI is a Business Intelligence and Data Visualization tool which helps you to convert data from various data sources into interactive dashboards and reports. It provides multiple software connectors and services.

- Power BI is a business analytics tool by Microsoft that provides interactive visualizations and business intelligence capabilities.
- With the help of power BI we can structure data and make business decisions out of those insights.
- It allows users to connect to a wide range of data sources, create interactive reports and dashboards, and share them with others.

2) Tableau

Tableau is a robust tool for visualizing data in a better way. You can connect any database to create understandable visuals. It is one of the best visualization tools that enables you to share visualization with other people.

- Tableau is one of the most popular data visualization tools.
- It is capable of learning the store's business patterns amd running queries against the data to help visualize the flaws and resolve them quickly.
- It allows users to create interactive and shareable dashboards, reports, and charts.
- Tableau supports various data sources and offers a user-friendly interface for creating visualizations.

3) Qlik

Qlik is a data visualization software which is used for converting raw data into knowledge. This software acts like a human brain which works on “association” and can go into any direction to search the answers.

- **QlikView and Qlik Sense** are data visualization and business intelligence tools developed by Qlik.
- They allow users to create interactive visualizations, dashboards, and reports using data from multiple sources.
- QlikView helps us to understand complex trends, patterns and convert it itno actionable insights.
- Qlik Sense is more modern and user-friendly compared to QlikView.

4) Google Data Studio

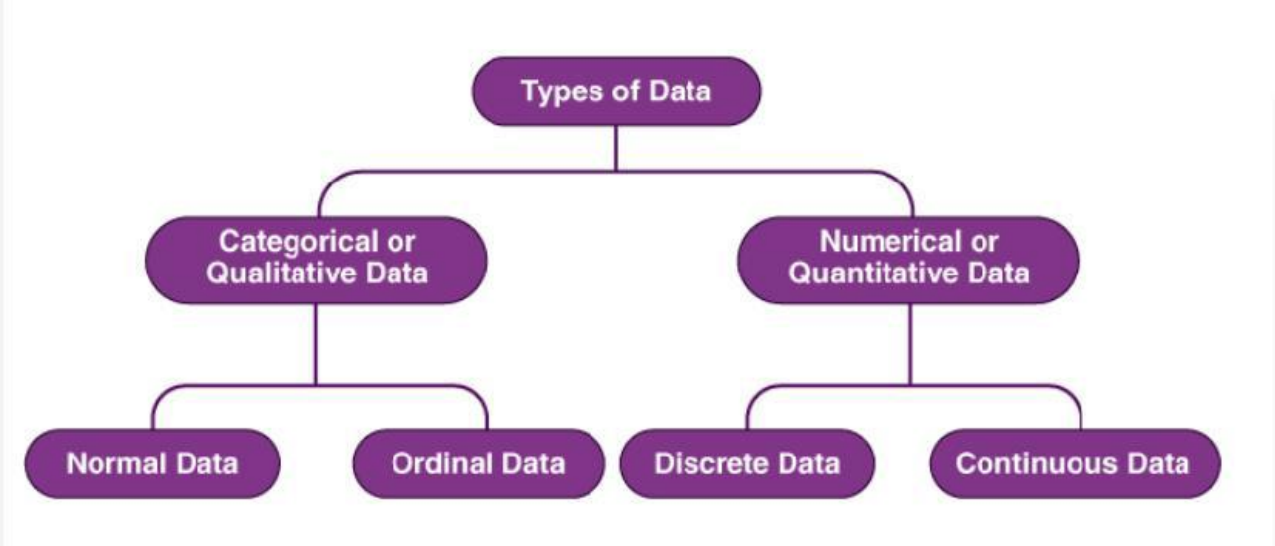
- Google Data Studio is a free data visualization tool that allows users to create interactive dashboards and reports.
- It is a dashboard and reporting tool that is easy to use, customize and share.
- It integrates seamlessly with other Google products such as Google Analytics, Google Sheets, and Google BigQuery.

Types of data

The data is classified into majorly four categories:

1. Nominal data
2. Ordinal data
3. Discrete data
4. Continuous data

Further, we can classify these data as follows:



Qualitative or Categorical Data

Qualitative data, also known as the categorical data, describes the data that fits into the categories. Qualitative data are not numerical. The categorical information involves categorical variables that describe the features such as a person's gender, home town etc. Categorical measures are defined in terms of natural language specifications, but not in terms of numbers.

Sometimes categorical data can hold numerical values (quantitative value), but those values do not have a mathematical sense. Examples of the categorical data are birthdate, favourite sport, school postcode. Here, the birthdate and school postcode hold the quantitative value, but it does not give numerical meaning.

Nominal Data

Nominal data is one of the types of qualitative information which helps to label the variables without providing the numerical value. Nominal data is also called the nominal scale. It cannot be ordered and measured. But sometimes, the data can be qualitative and quantitative. Examples of nominal data are letters, symbols, words, gender etc.

The nominal data are examined using the grouping method. In this method, the data are grouped into categories, and then the frequency or the percentage of the data can be calculated. These data are visually represented using the pie charts.

Ordinal Data

Ordinal data/variable is a type of data that follows a natural order. The significant feature of the nominal data is that the difference between the data values is not determined. This variable is mostly found in surveys, finance, economics, questionnaires, and so on. The ordinal data is commonly represented using a bar chart. These data are investigated and interpreted through many visualisation tools. The information may be expressed using tables in which each row in the table shows a distinct category.

Quantitative or Numerical Data

Quantitative data is also known as numerical data which represents the numerical value (i.e., how much, how often, how many). Numerical data gives information about the quantities of a specific thing. Some examples of numerical data are height, length, size, weight, and so on. Quantitative data can be classified into two different types based on the data sets. The two different classifications of numerical data are discrete data and continuous data.

Discrete Data

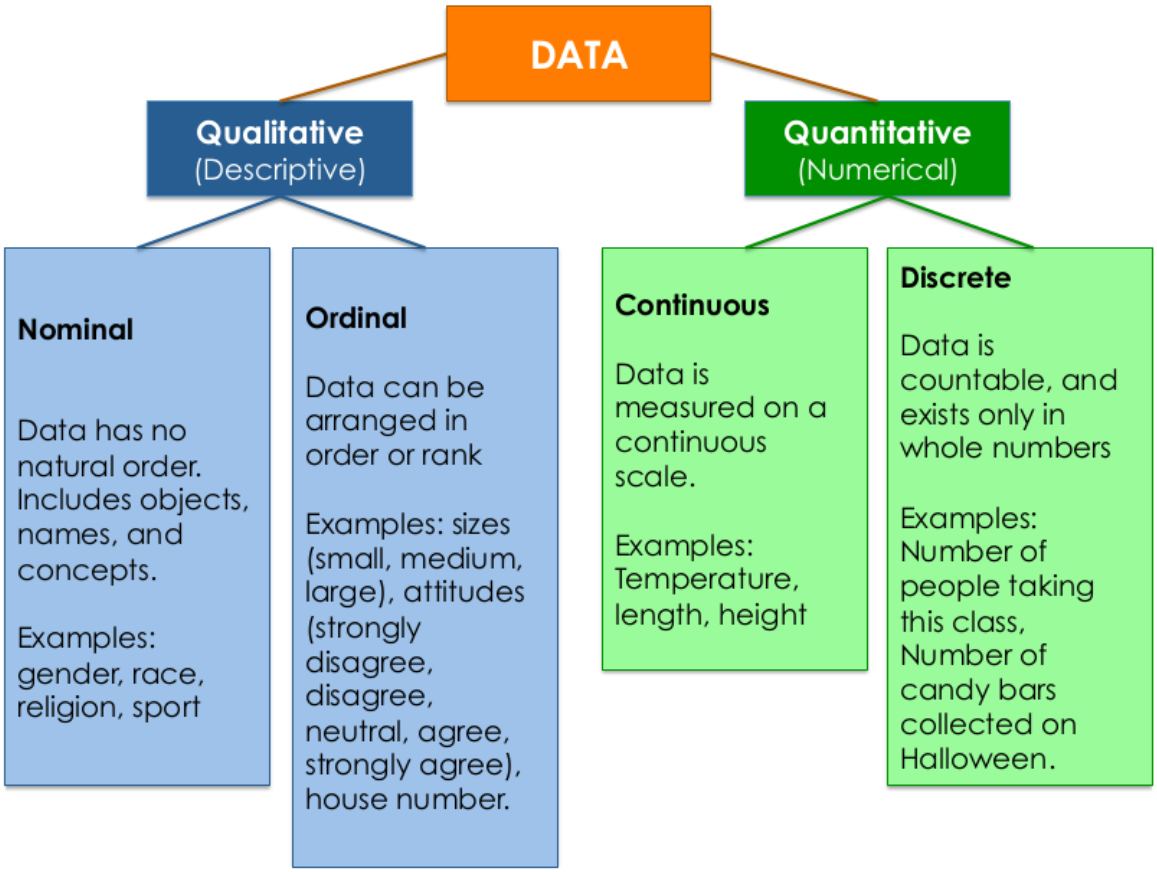
Discrete data can take only discrete values. Discrete information contains only a finite number of possible values. Those values cannot be subdivided meaningfully. Here, things can be counted in whole numbers.

Example: Number of students in the class

Continuous Data

Continuous data is data that can be calculated. It has an infinite number of probable values that can be selected within a given specific range.

Example: Temperature range



1.5 DATA ABSTRACTION

Data abstraction is the process of structuring, categorizing, and interpreting raw data into meaningful forms that can be effectively visualized and analyzed.

Key Aspects of Data Abstraction includes

1. Data Types

- **Item**: A single, discrete entity (e.g., a row in a table).
- **Attribute**: A measurable property or variable (e.g., income, temperature).
- **Link**: A relationship between items (e.g., social connections).
- **Position**: Spatial data defined by coordinates.
- **Grid**: Sampling strategy for continuous data in a structured layout.

2. Dataset Types

- **Tables**: 2D representation; rows = items, columns = attributes.
- **Networks**: Nodes (items) connected via links (relationships).
- **Fields**: Continuous domain data sampled at discrete points.
- **Geometry**: Describes shape, structure, and spatial layout (points, surfaces, volumes).
- **Other**: Clusters, sets, and lists that group or categorize items.

3. Attribute Types

- **Categorical**: No inherent order (e.g., fruit types, gender).
- **Ordered**:
    - **Ordinal**: Implicit order but limited arithmetic (e.g., grades, rankings).
    - **Quantitative**: Full arithmetic (e.g., weight, temperature).

4. Direction of Ordering

- **Sequential**: One-way progression (e.g., 0 to 100).
- **Diverging**: Ranges from a central baseline in two directions (e.g., temperature above and below zero).
- **Cyclic**: Repeats in cycles (e.g., months of the year, clock time).

5. Data Semantics

- Concerned with **real-world meaning** of data.
- Helps in distinguishing between:
    - **Key**: Used to index or organize data (e.g., ID, time).
    - **Value**: The actual measured/observed data (e.g., income).

Data Types:

Figure 2.2 shows the five basic **data types** discussed they are: items, attributes, links, positions, and grids. An **attribute** is some specific property that can be measured, observed, or logged. For example, attributes could be salary, price, number of sales, protein expression levels, or

temperature. An **item** is an individual entity that is discrete, such as a row in a simple table or a node

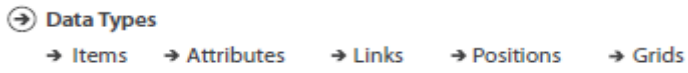


Figure 2.2. The five basic data types: items, attributes, links, positions, and grids.

in a network. For example, items may be people, stocks, coffee shops, genes, or cities. A **link** is a relationship between items, typically within a network. A **grid** specifies the strategy for sampling continuous data in terms of both geometric and topological relationships between its cells. A **position** is spatial data, providing a location in two-dimensional (2D) or three-dimensional (3D) space. For example, a position might be a latitude–longitude pair describing a location on the Earth's surface or three numbers specifying a location within the region of space measured by a medical scanner.

Dataset Types

A dataset is any collection of information that is the target of analysis. The four basic dataset types are tables, networks, fields, and geometry. Other ways to group items together include clusters, sets, and lists. In real-world situations, complex combinations of these basic types are common.

Figure 2.3 shows that these basic dataset types arise from combinations of the data types of items, attributes, links, positions, and grids.

Figure 2.4 shows the internal structure of the four basic dataset types in detail. Tables have cells indexed by items and attributes, for either the simple flat case or the more complex multidimensional case. In a network, items are usually called nodes, and they are connected with links; a special case of networks is trees. Continuous fields have grids based on spatial positions where cells contain attributes. Spatial geometry has only position information.

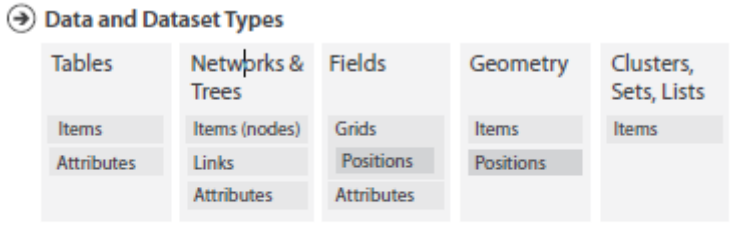


Figure 2.3. The four basic dataset types are tables, networks, fields, and geometry; other possible collections of items are clusters, sets, and lists. These datasets are made up of five core data types: items, attributes, links, positions, and grids.

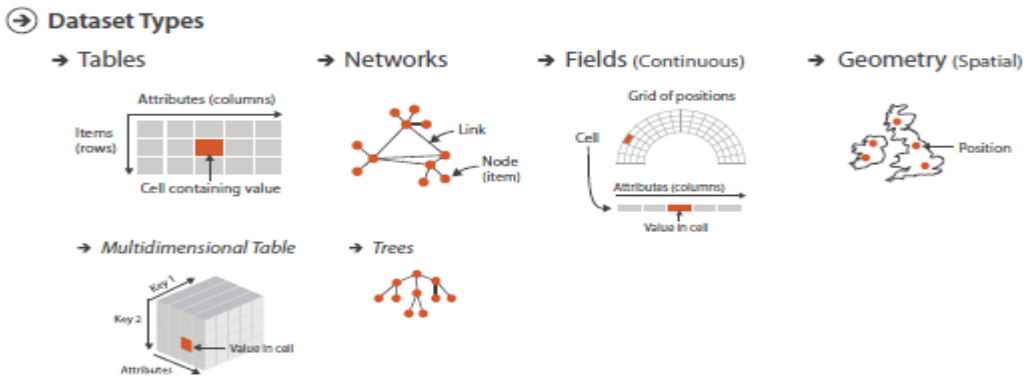


Figure 2.4. The detailed structure of the four basic dataset types.

Tables:

Many datasets come in the form of **tables** that are made up of rows and columns, a familiar form to anybody who has used a spreadsheet. In this chapter, I focus on the concept of a table as simply a type of dataset that is independent of any particular visual representation; later chapters address the question of what visual representations are appropriate for the different types of datasets.

For a simple **flat table**, the terms used in this book are that each row represents an **item** of data, and each column is an **attribute** of the dataset. Each **cell** in the table is fully specified by the combination of a row and a column—an item and an attribute—and contains a **value** for that pair. Figure 2.5 shows an example of the first few dozen items in a table of orders, where the attributes are order ID, order date, order priority, product container, product base margin, and ship date.

A **multidimensional table** has a more complex structure for indexing into a cell, with multiple keys.

A	B	C	S	T	U
Order ID	Order Date	Order Priority	Product Container	Product Base Margin	Ship Date
3	10/14/06	5-Low	Large Box	0.8	10/21/06
6	2/21/08	4-Not Specified	Small Pack	0.55	2/22/08
32	7/16/07	2-High	Small Pack	0.79	7/17/07
32	7/16/07	2-High	Jumbo Box		7/17/07
32	7/16/07	2-High	Medium Box		7/18/07
32	7/16/07	2-High	Medium Box		7/18/07
35	10/23/07	4-Not Specified	Wrap Bag	0.52	10/24/07
35	10/23/07	4-Not Specified	Small Box	0.58	10/25/07
36	11/3/07	1-Urgent	Small Box	0.55	11/3/07
65	3/18/07	1-Urgent	Small Pack	0.49	3/19/07
66	[illegible]	5-Low	Wrap Bag	0.56	1/20/05
69	[illegible]	4-Not Specified	Small Pack	0.44	6/6/05
69	[illegible]	4-Not Specified	Wrap Bag	0.6	6/6/05
70	12/18/06	5-Low	Small Box	0.59	12/23/06
70	12/18/06	5-Low	Wrap Bag	0.82	12/23/06
96	4/17/05	2-High	Small Box	0.55	4/19/05
97	1/29/06	3-Medium	Small Box	0.38	1/30/06
129	11/19/08	5-Low	Small Box	0.37	11/28/08
130	5/8/08	2-High	Small Box	0.37	5/9/08
130	5/8/08	2-High	Medium Box	0.38	5/10/08
130	5/8/08	2-High	Small Box	0.6	5/11/08
132	6/11/06	3-Medium	Medium Box	0.6	6/12/06
132	6/11/06	3-Medium	Jumbo Box	0.69	6/14/06
134	5/1/08	4-Not Specified	Large Box	0.82	5/3/08
135	10/21/07	4-Not Specified	Small Pack	0.64	10/23/07
166	9/12/07	2-High	Small Box	0.55	9/14/07
193	8/8/06	1-Urgent	Medium Box	0.57	8/10/06
194	4/5/08	3-Medium	Wrap Bag	0.42	4/7/08

Figure 2.5. In a simple table of orders, a row represents an *item*, a column represents an *attribute*, and their intersection is the *cell* containing the value for that pairwise combination.

Networks and Trees

The dataset type of **networks** is well suited for specifying that there is some kind of relationship between two or more items. *An item in a network is often called a **node**.* A **link** is a relation between two items. For example, in an articulated social network the nodes are people, and links mean friendship.

In a gene interaction network, the nodes are genes, and links between them mean that these genes have been observed to interact with each other. In a computer network, the nodes are computers, and the links represent the ability to send messages directly between two computers using physical cables or a wireless connection.

Network nodes can have associated attributes, just like items in a table. In addition, the links themselves could also be considered to have attributes associated with them; these may be partly or wholly disjointed from the node attributes. It is again important to distinguish between the abstract concept of a network and any particular visual layout of that network where the nodes and edges have particular spatial positions. This chapter concentrates on the former.

Trees: Networks with hierarchical structure are more specifically called **trees**. In contrast to a general network, trees do not have cycles: each child node has only one parent node pointed to it. One example of a tree is the organization chart of a company, showing who reports to whom; another example is a tree showing the evolutionary relationships between species in the biological tree of life, where the child nodes of humans and monkeys both share the same parent node of primates.

Fields: The **field** dataset type also contains attribute values associated with cells.1 Each **cell** in a field contains measurements or calculations from a **continuous** domain: there are conceptually infinitely many values that you might measure, so you could always take a new measurement between any two existing ones. Continuous phenomena that might be measured in the physical world or simulated in software include temperature, pressure, speed, force, and density; mathematical functions can also be continuous. For example, consider a field dataset representing a medical scan of a human body containing measurements indicating the

density of tissue at many sample points, spread regularly throughout a volume of 3D space. A low-resolution scan would have 262,144 cells, providing information about a cubical volume of space with 64 bins in each direction. Each cell is associated with a specific region in 3D space. The density measurements could be taken closer together with a higher resolution grid of cells, or further apart for a coarser grid.

Continuous data requires careful treatment that takes into account the mathematical questions of **sampling**, how frequently to take the measurements, and **interpolation**, how to show values in between the sampled points in a way that does not mislead. Interpolating appropriately between the measurements allows you to **reconstruct** a new view of the data from an arbitrary viewpoint that's faithful to what you measured. These general mathematical

problems are studied in areas such as signal processing and statistics. Visualizing fields requires grappling extensively with these concerns. In contrast, the table and network datatypes

discussed above are an example of **discrete** data where a finite number of individual items exist, and interpolation between them is not a meaningful concept. In the cases where a mathematical framework is necessary, areas such as graph theory and combinatorics provide relevant ideas.

Spatial Fields: Continuous data is often found in the form of a **spatial field**, where the cell structure of the field is based on sampling at spatial positions. Most datasets that contain inherently spatial data occur in the context of tasks that require understanding aspects of its spatial structure, especially shape.

For example, with a spatial field dataset that is generated with a medical imaging instrument, the user's task could be to locate suspected tumors that can be recognized through distinctive shapes or densities. An obvious choice for visual encoding would be to show something that spatially looks like an X-ray image of the human body and to use color coding to highlight suspected tumors. Another example is measurements made in a real or simulated wind tunnel of the temperature and pressure of air flowing over airplane wings at many points in 3D space, where the task is to compare the flow patterns in different regions. One possible visual encoding would use the geometry of the wing as the spatial substrate, showing the temperature and pressure using size-coded arrows.

The likely tasks faced by users who have spatial field data constrains many of the choices about the use of space when designing visual encoding idioms. Many of the choices for **nonspatial data**, where no information about spatial position is provided with the dataset, are unsuitable in this case. Thus, the question of whether a dataset has the type of a spatial field or a nonspatial table has extensive and far-reaching implications for idiom design. Historically, vis diverged into areas of specialization based on this very differentiation. The subfield of **scientific visualization**, or **scivis** for short, is concerned with situations where spatial position is *given* with the dataset. A central concern in scivis is handling continuous data appropriately within the mathematical framework of signal processing. The subfield of **information visualization**, or **infovis** for short, is concerned with situations where the use of space in a visual encoding is *chosen* by the designer. A central concern in infovis is determining whether the chosen idiom is suitable for the combination of data and task, leading to the use of methods from human–computer interaction and design.

Grid Types

When a field contains data created by sampling at completely regular intervals, as in the previous example, the cells form a uniform grid. There is no need to explicitly store the grid geometry in terms of its location in space, or the grid topology in terms of how each cell connects with its neighbouring cells. More complicated examples require storing different amounts of geometric and topological information about the underlying grid. A rectilinear grid supports nonuniform sampling, allowing efficient storage of information that has high complexity in some areas and low complexity in others, at the cost of storing some information about the geometric location of each row. A structured grid allows curvilinear shapes, where the geometric location of each cell needs to be specified. Finally, unstructured grids provide

complete flexibility, but the topological information about how the cells connect to each other must be stored explicitly in addition to their spatial positions.

Geometry

The geometry dataset type specifies information about the shape of items with explicit spatial positions. The items could be points, or one-dimensional lines or curves, or 2D surfaces or regions, or 3D volumes.

Geometry datasets are intrinsically spatial, and like spatial fields they typically occur in the context of tasks that require shape understanding. Spatial data often includes hierarchical structure at multiple scales. Sometimes this structure is provided intrinsically with the dataset, or a hierarchy may be derived from the original data.

Geometry datasets do not necessarily have attributes, in contrast to the other three basic dataset types. Many of the design issues in vis pertain to questions about how to encode attributes. Purely geometric data is interesting in a vis context only when it is derived or transformed in a way that requires consideration of design choices. One classic example is when contours are derived from a spatial field. Another is when shapes are generated at an appropriate level of detail for the task at hand from raw geographic data, such as the boundaries of a forest or a city or a country, or the curve of a road. The problem of how to create images from a geometric description of a scene falls into another domain: computer graphics. While vis draws on algorithms from computer graphics, it has different concerns from that domain. Simply showing a geometric dataset is not an interesting problem from the point of view of a vis designer.

Other Combinations

Beyond tables, there are many ways to group multiple *items* together, including sets, lists, and clusters. A set is simply an unordered group of items. A group of items with a specified ordering could be called a **list**. A **cluster** is a grouping based on attribute similarity, where items within a cluster are more similar to each other than to ones in another cluster.

There are also more complex structures built on top of the basic network type. A path through a network is an ordered set of segments formed by links connecting nodes. A compound network is a network with an associated tree: all of the nodes in the network are the leaves of the tree, and interior nodes in the tree provide a hierarchical structure for the nodes that is different from network links between them.

Many other kinds of data either fit into one of the previous categories or do so after transformations to create derived attributes. Complex and hybrid combinations, where the complete dataset contains multiple basic types, are common in real-world applications.

The set of basic types presented above is a starting point for describing the *what* part of an analysis instance that pertains to data; that is, the **data abstraction**. In simple cases, it may be possible to describe your data abstraction using only that set of terms. In complex cases, you may need additional description as well. If so, your goal should be to translate domain-specific terms into words that are as generic as possible.

Figure 2.6. Dataset availability can be either static or dynamic, for any dataset type.

Attribute Types

Figure 2.7 shows the attribute types. The major distinction is between categorical versus ordered. Within the ordered type is a further differentiation between ordinal versus quantitative. Ordered data might range sequentially from a minimum to a maximum value, or it might diverge in both directions from a zero point in the middle of a range, or the values may wrap around in a cycle. Also, attributes may have hierarchical structure.

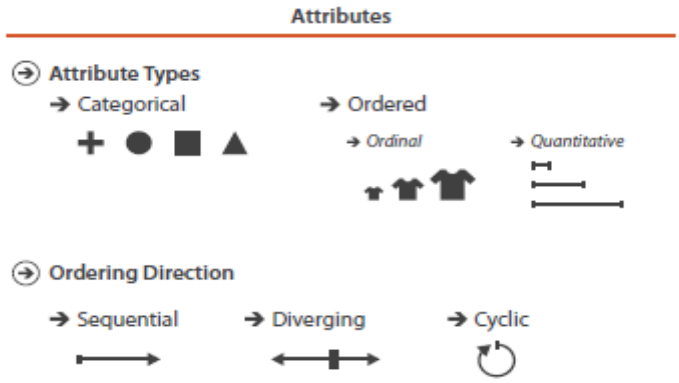


Figure 2.7. Attribute types are categorical, ordinal, or quantitative. The direction of attribute ordering can be sequential, diverging, or cyclic.

Categorical

The first distinction is between categorical and ordered data. The type of **categorical** data, such as favorite fruit or names, does not have an implicit ordering, but it often has hierarchical structure. Categories can only distinguish whether two things are the same (apples) or different (apples versus oranges). Of course, any arbitrary external ordering can be imposed upon categorical data. Fruit could be ordered alphabetically according to its name, or by its price—but only if that auxiliary information were available. However, these orderings are not implicit in the attribute itself, the way they are with quantitative or ordered data. Other examples of categorical attributes are movie genres, file types, and city names.

Ordered: Ordinal and Quantitative

All **ordered** data does have an implicit ordering, as opposed to unordered *categorical* data. This type can be further subdivided. With **ordinal** data, such as shirt size, we cannot do full-fledged arithmetic, but there is a well-defined ordering. For example, large minus medium is not a meaningful concept, but we know that medium falls between small and large. Rankings

are another kind of ordinal data; some examples of ordered data are top-ten lists of movies or initial lineups for sports tournaments depending on past performance.

Sequential versus Diverging

Ordered data can be either **sequential**, where there is a homogeneous range from a minimum to a maximum value, or **diverging**, which can be deconstructed into two sequences pointing in opposite directions that meet at a common zero point. For instance, a mountain *height* dataset is sequential, when measured from a minimum point of sea level to a maximum point of Mount Everest. A *bathymetric* dataset is also sequential, with sea level on one end and the lowest point on the ocean floor at the other. A full *elevation* dataset would be diverging, where the values go up for mountains on land and down for undersea valleys, with the zero value of sea level being the common point joining the two sequential datasets.

Cyclic

Ordered data may be cyclic, where the values wrap around back to a starting point rather than continuing to increase indefinitely. Many kinds of time measurements are cyclic, including the hour of the day, the day of the week, and the month of the year.

1.6 TASK ABSTRACTION

Task abstraction refers to the systematic way of categorizing **user goals and actions** when interacting with visualized data. It helps designers understand *what* users want to do with the data and guides the selection of appropriate visualization techniques.

Why Task Abstraction?

- To **bridge user intent and visual design**
- To ensure that the visualization supports **meaningful interaction**
- To tailor tools based on **user goals and data type**

1. Actions

Describes **how** users interact with data.

- **High-Level Actions**
  - **Analyze:** Gain insights from data
    - *e.g., Find causes of sales drop*
  - **Consume:** Use data for a purpose
    - *e.g., Read a report*
  - **Discover, Present, Enjoy:** Share or experience visual stories
  - **Produce:** Generate visual output
  - **Annotate, Record, Derive:** Add notes, save findings, generate new data

🔸 **Mid-Level Actions**

- **Search:** Look for specific information
    - **Lookup:** Find a known value
    - **Locate:** Find where a value exists
    - **Browse:** Scan through data
    - **Explore:** Interactively investigate unknowns

🔸 **Low-Level Actions**

- **Query:** Examine specific data details
    - **Identify:** Recognize items
    - **Compare:** Analyze similarities/differences
    - **Summarize:** View overall patterns

🔹 **2. Targets**

What users want to focus on:

- **All Data:** Understand entire dataset
- **Trends:** Look for patterns over time
- **Outliers:** Spot anomalies
- **Features:** Key elements in the dataset

🔹 **3. Attributes**

Refers to the **data dimensions or fields** being analyzed:

- **One attribute:** E.g., view distribution of age
- **Multiple attributes:** E.g., analyze age vs income

🔹 **4. Data Types**

Nature of data affects visualization style:

- **Network:** Relationships (e.g., social networks)
- **Spatial:** Geographic or location-based data

Component	Description
Actions	What users do: Analyze, Search, Query
Targets	What they focus on: All data, trends, outliers
Attributes	Data dimensions involved: one or many
Data Types	Data structure: network or spatial

Use Vis.	Actions	High-Level: Analyze	Consume	Discover, Present, Enjoy
			Produce	Annotate, Record, Derive
		Mid-Level: Search	Lookup, Locate, Browse, Explore	
		Low-Level: Query	Identify, Compare, Summarize	
	Targets	All Data	Trends, Outliers, Features	
		Attributes	One, Multiple	
		Network		
		Spatial		

1.7 FOUR LEVELS OF VALIDATION

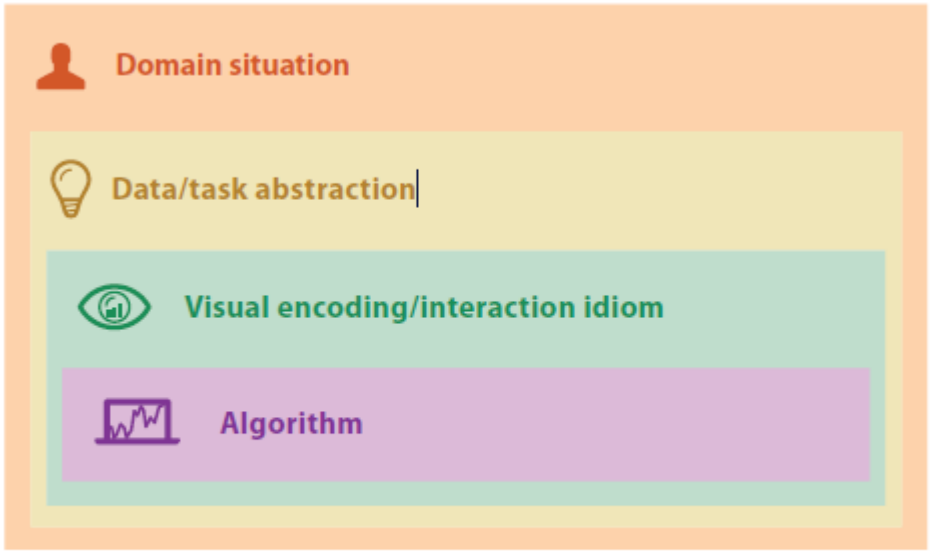


Figure 4.2. The four nested levels of vis design.

As shown on the figure above, there are four nested levels of vis design, including domain situation, task and data abstraction, visual encoding and interaction idiom, and algorithm. Each level has different threats to validity, and validation approaches should be chosen accordingly. In addition, these four levels are nested. That means the output from an upstream level above is input to the downstream level below. A block is the outcome of the design process at that level.

Domain Situation

Blocks at this top level describe a specific domain situation, which encompasses a group of target users, their domain of interest, their questions, and their data. The term domain is frequently used in the vis literature to mean a particular field of interest of the target users of a vis tool, for example microbiology or high-energy physics or e-commerce. Each domain usually has its own vocabulary for describing its data and problems, and there is usually some existing workflow of how the data is used to solve their problems. The group of target users might be as narrowly defined as a handful of people working at a specific company, or as broadly defined as anybody who does scientific research. One example of a situation block is a computational biologist working in the field of comparative genomics, using genomic sequence data to ask questions about the genetic source of adaptively in a species. While one kind of situation is a specific set of users whose questions about their data arise from their work, situations arise in other contexts. For example, another situation is members of the general public making medical decisions about their healthcare in the presence of uncertainty
At this level, situation blocks are identified: the outcome of the design process is an understanding that the designer reaches about the needs of the user. The methods typically used by designers to identify domain situation blocks include interviews, observations, or careful research about target users within a specific domain.

Developing a clear understanding of the requirements of a particular target audience is a tricky problem for a designer. While it might seem obvious to you that it would be a good idea to understand requirements, it's a common pitfall for designers to cut corners by making assumptions rather than actually engaging with any target users. In most cases users know they need to somehow view their data, but they typically cannot directly articulate their analysis needs in a clear-cut way. Eliciting system requirements is not easy, even when you have unfettered access to target users fluent in the vocabulary of the domain and immersed in its workflow. Asking users to simply introspect about their actions and needs is notoriously insufficient: what users say they do when reflecting on their past behavior gives you an

incomplete picture compared with what they actually do if you observe them. The outcome of identifying a situation block is a detailed set of questions asked about or actions carried out by the target users, about a possibly heterogeneous collection of data that's also understood in detail. Two of the questions that may have been asked by the computational biologist working in comparative genomics working above were "What are the differences between individual nucleotides of feature pairs?" and "What is the density of coverage and where are the gaps across a chromosome?". In contrast, a very general question such as "What is the genetic basis of disease?" is not specific enough to be useful as input to the next design level.

Task and Data Abstraction

Design at the next level requires abstracting the specific domain questions and data from the domain-specific form that they have at the top level into a generic representation. Abstracting into the domain-independent vocabulary allows you to realize how domain situation blocks that are described using very different language might have similar reasons why the user needs the vis tool and what data it shows.

Questions from very different domain situations can map to the same abstract vis tasks. Examples of abstract tasks include browsing, comparing, and summarizing. Task blocks are identified by the designer as being suitable for a particular domain situation block, just as the situation blocks themselves are identified at the level above.

Abstract data blocks, however, are designed. Selecting a data block is a creative design step rather than simply an act of identification. While in some cases you may decide to use the data in exactly the way that it was identified in the domain situation, you will often choose to transform the original data from its upstream form to something quite different. The data abstraction level requires you to consider whether and how the same dataset provided by a user should be transformed into another form. Many vis idioms are specific to a particular data type, such as a table of numbers where the columns contain quantitative, ordered, or categorical data; a node–link graph or tree; or a field of values at every point in space. Your goal is to determine which data type would support a visual representation of it that addresses the user's problem. Although sometimes the original form of the dataset is a good match for a visual encoding that solves the problem, often a transformation to another data type provides a better solution.

Explicitly considering the choices made in abstracting from domain-specific to generic tasks and data can be very useful in the vis design process. The unfortunate alternative is to do this abstraction implicitly and without justification. For example, many early web vis papers implicitly posited that solving the "lost in hyperspace" problem should be done by showing the searcher a visual representation of the topological structure of the web's hyperlink connectivity

graph. In fact, people do not need an internal mental representation of this extremely complex structure to find a page of interest. Thus, no matter how cleverly the information was visually encoded at the idiom design level, the resulting vis tools all incurred additional cognitive load for the user rather than reducing it.

Visual Encoding and Interaction Idiom

At the third level, you decide on the specific way to create and manipulate the visual representation of the abstract data block that you chose at the previous level, guided by the abstract tasks that you also identified at that level. I call each distinct possible approach an idiom. There are two major concerns at play with idiom design. One set of design choices covers how to create a single picture of the data: the visual encoding idiom controls exactly what users see. Another set of questions involves how to manipulate that representation dynamically: the interaction idiom controls how users change what they see. For example, the Word Tree system combines the visual encoding idiom of a hierarchical tree representation of keywords laid out horizontally, preserving information about the context of their use within the original text, and the interaction idiom of navigation based on keyword selection. While it's often possible to analyze encoding and interaction idioms as separable decisions, in some cases these decisions are so intertwined that it's best to consider the outcome of these choices to be a single combined idiom. Idiom blocks are designed: they are the outcome of decisions that you make. The design space of static visual encoding idioms is already enormous, and when you consider how to manipulate them dynamically that space of possibilities is even bigger. The nested model emphasizes identifying task abstractions and deciding on data abstractions in the previous level exactly so that you can use them to rule out many of the options as being a bad match for the goals of the users. You should make decisions about good and bad matches based on understanding human abilities, especially in terms of visual perception and memory. While it's common for vis tools to provide multiple idioms that users might choose between, some vis tools are designed to be very narrow in scope, supporting only a few or even just a single idiom.

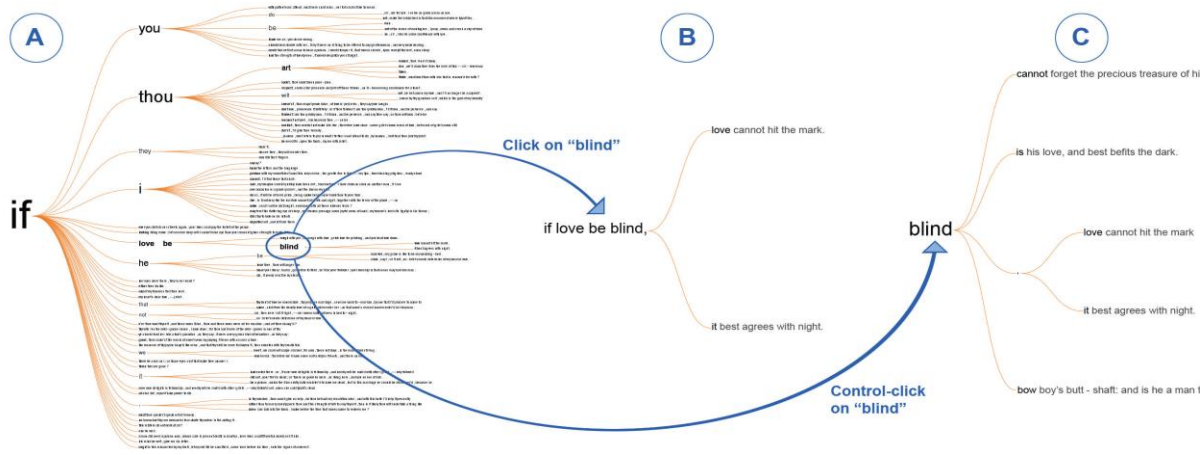


Figure 4.3. Word Tree combines the visual encoding idiom of a hierarchical tree of keywords laid out horizontally and the interaction idiom of navigation based on keyword selection.

Algorithm

The innermost level involves all of the design choices involved in creating an algorithm: a detailed procedure that allows a computer to automatically carry out a desired goal. In this case, the goal is to efficiently handle the visual encoding and interaction idioms that you chose in the previous level. Algorithm blocks are also designed, rather than just identified.

You could design many different algorithms to instantiate the same idiom. For example, one visual encoding idiom for creating images from a three-dimensional field of measurements, such as scans created for medical purposes with magnetic resonance imaging, is direct volume rendering. Many different algorithms have been proposed as ways to achieve the requirements of this idiom, including ray casting, splatting, and texture mapping. You might

determine that some of these are better than others according to measures such as the speed of the computation, how much computer memory is required, and whether the resulting image is an exact match with the specified visual encoding idiom or just an approximation.

The nested model emphasizes separating algorithm design, where your primary concerns are about computational issues, from idiom design, where your primary concerns are about human perceptual issues.

Of course, there is an interplay between these levels. For example, a design that requires something to change dynamically when the user moves the mouse may not be feasible if computing that would take minutes or hours instead of a fraction of a second. However, clever algorithm design could save the day if you come up with a way to precompute data that supports a fast enough response.

Unit – II

Visualization Techniques: Scalar and point techniques, – vector visualization techniques – multidimensional techniques – visualizing cluster analysis – matrix visualization in Bayesian data analysis

Data visualization transforms raw numbers into actionable insights. Whether you're analyzing household power consumption, weather patterns, or financial trends, the right visualization technique can reveal hidden patterns that tables of numbers never could.

In this article, we'll explore:

✓ Scalar & Point Techniques (single-value data like temperature or power usage).
✓ Vector Visualization (direction + magnitude, like electrical current flow).
✓ Multi-dimensional Methods (complex datasets with many variables).

We'll use the ***UCI Household Power Consumption Dataset*** to demonstrate real-world applications.

```
# Core Data Handling
import pandas as pd       # Data manipulation and analysis (DataFrames)
import numpy as np        # Numerical computing (arrays, math operations)

# Basic Visualization
import matplotlib.pyplot as plt  # Foundational plotting library (2D/3D)
import seaborn as sns            # High-level statistical graphics (built on mat

# Interactive Visualization
import plotly.express as px      # Interactive plots (hover tools, zoom)
import plotly.graph_objects as go # More control over interactive plots

# Dimensionality Reduction
from sklearn.manifold import TSNE  # t-Distributed Stochastic Neighbor Embedding
from sklearn.decomposition import PCA  # Principal Component Analysis

# Data Preprocessing
from sklearn.preprocessing import StandardScaler  # Feature scaling (mean=0, std

# Advanced Visualization
from scipy.stats import gaussian_kde  # Kernel Density Estimation (for contour p
from scipy.interpolate import griddata  # Grid interpolation (vector fields)
import joypy                          # Horizon/ridge plots
import geopandas as gpd               # Geospatial data handling
import networkx as nx                 # Graph/network visualizations
```

```
# Animation
from matplotlib.animation import FuncAnimation  # Animated visualizations

# Utility
from tabulate import tabulate  # Pretty-printing tables
from pandas.plotting import radviz  # Radial coordinates visualization
```

Conversion of Data txt File to CSV File and Load it For Processing

```
df1 = pd.read_csv("/content/household_power_consumption.txt")

df1.to_csv('household_power_consumption.csv',index = None)

# Load data
#Access Dataset : https://archive.ics.uci.edu/dataset/235/individual+housel
url = "/content/household_power_consumption.csv"
df = pd.read_csv(url, sep=';', parse_dates={'DateTime': ['Date', 'Time']},
                 infer_datetime_format=True, low_memory=False, na_values=[

# Preprocessing
df = df.dropna().sample(frac=0.1, random_state=42)  # Downsample for demo
numeric_cols = ['Global_active_power', 'Global_reactive_power', 'Voltage',
                'Global_intensity', 'Sub_metering_1', 'Sub_metering_2', 'S
df[numeric_cols] = df[numeric_cols].apply(pd.to_numeric)
df['Hour'] = df['DateTime'].dt.hour
```

2.1 SCALAR & POINT VISUALIZATIONS

Scalar visualization deals with data where each point in a dataset has a single numerical value associated with it. This value, or "scalar," represents a magnitude or intensity of a particular property at that location. The goal of scalar visualization is to effectively communicate the distribution and variation of this single value across the dataset.

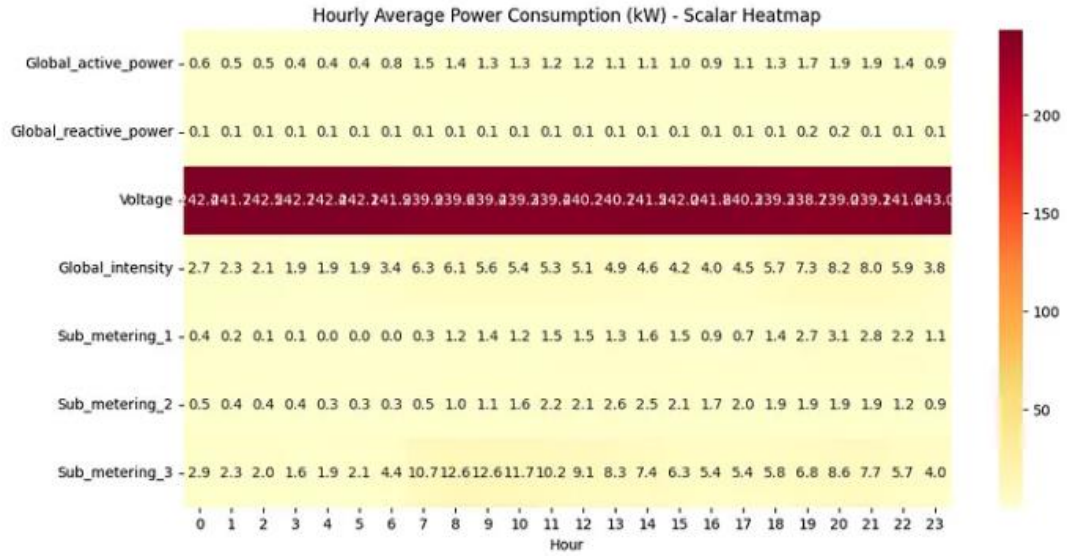
A. Heatmap (Daily Power Patterns)

Heatmaps are powerful data visualization tools that use **color intensity** to represent the magnitude of a value across a two-dimensional grid or matrix. They provide an immediate and intuitive way to identify patterns, correlations, and anomalies within large datasets, making trends and insights visible "at a glance."

At their core, heatmaps map numerical data to a color spectrum. Typically:

- **High values** are represented by warmer colors (like red, orange, yellow).
- **Low values** are represented by cooler colors (like blue, green, purple).
- **Intermediate values** are shown with colors in between.

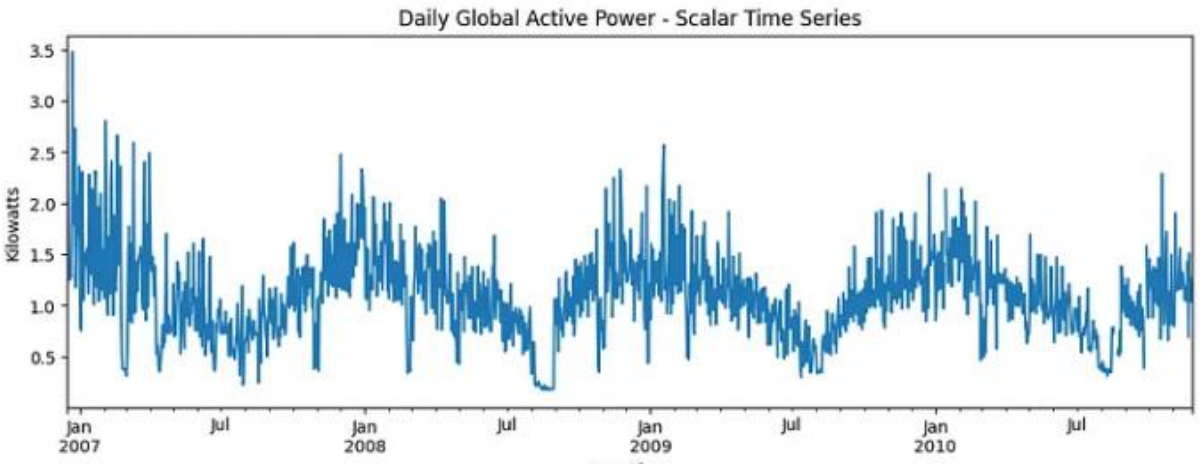
```
daily_avg = df.groupby('Hour')[numeric_cols].mean()

plt.figure(figsize=(12, 6))
sns.heatmap(daily_avg.T, cmap="YlOrRd", annot=True, fmt=".1f")
plt.title("Hourly Average Power Consumption (kW) - Scalar Heatmap")
plt.show()
```



B. Time Series (Global Active Power)

The term "Time Series (Global Active Power)" typically refers to a dataset that records the total active electrical power consumed by a household (or a similar entity) over a period of time. This data is sequential, with each data point associated with a specific timestamp. Analyzing this time series can reveal patterns in energy consumption, identify peak usage periods, and provide insights for energy management and forecasting.

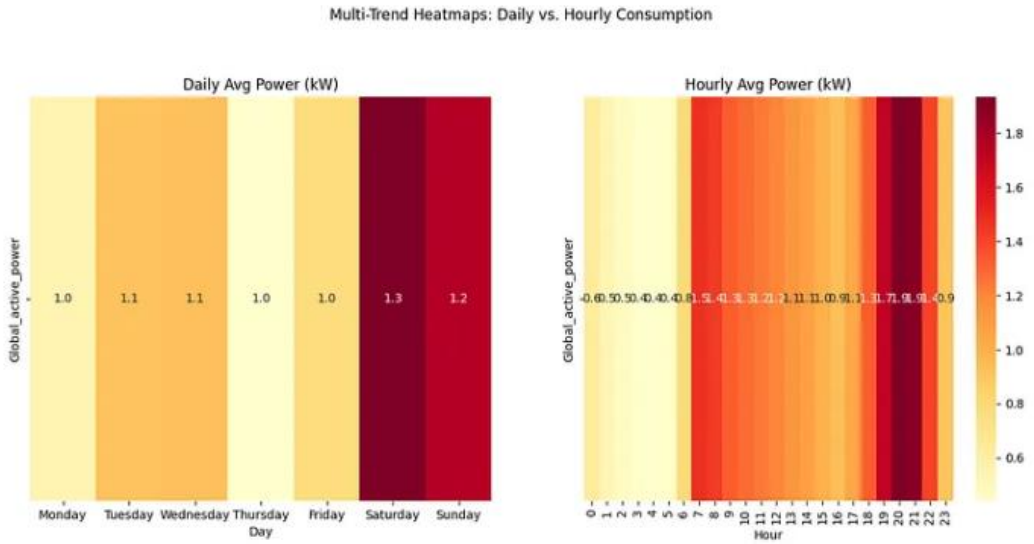
```
plt.figure(figsize=(12, 4))
df.set_index('DateTime')['Global_active_power'].resample('D').mean().plot()
plt.ylabel('Kilowatts')
plt.title("Daily Global Active Power - Scalar Time Series")
plt.show()
```



C. Multi-Trend Heatmap (Daily & Hourly Power)

A **Multi-Trend Heatmap** is an extension of the standard heatmap that goes beyond visualizing a single variable across two dimensions. Instead, it aims to display **multiple trends or variables simultaneously** within the same grid, often by employing different visual encodings for each trend. This allows for the exploration of complex relationships and correlations between several factors at a glance.

```
# Daily and Hourly trends
plt.figure(figsize=(15, 6))

# Daily trend
plt.subplot(1, 2, 1)
# Extract day of the week from 'DateTime' column
df['Day'] = df['DateTime'].dt.day_name()
daily_avg = df.groupby('Day')['Global_active_power'].mean().reindex(
    ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sund
)
sns.heatmap(daily_avg.to_frame().T, cmap="YlOrRd", annot=True, fmt=".1f", cbar
plt.title("Daily Avg Power (kW)")

# Hourly trend
plt.subplot(1, 2, 2)
hourly_avg = df.groupby('Hour')['Global_active_power'].mean()
sns.heatmap(hourly_avg.to_frame().T, cmap="YlOrRd", annot=True, fmt=".1f")
plt.title("Hourly Avg Power (kW)")

plt.suptitle("Multi-Trend Heatmaps: Daily vs. Hourly Consumption", y=1.05)
plt.show()
```

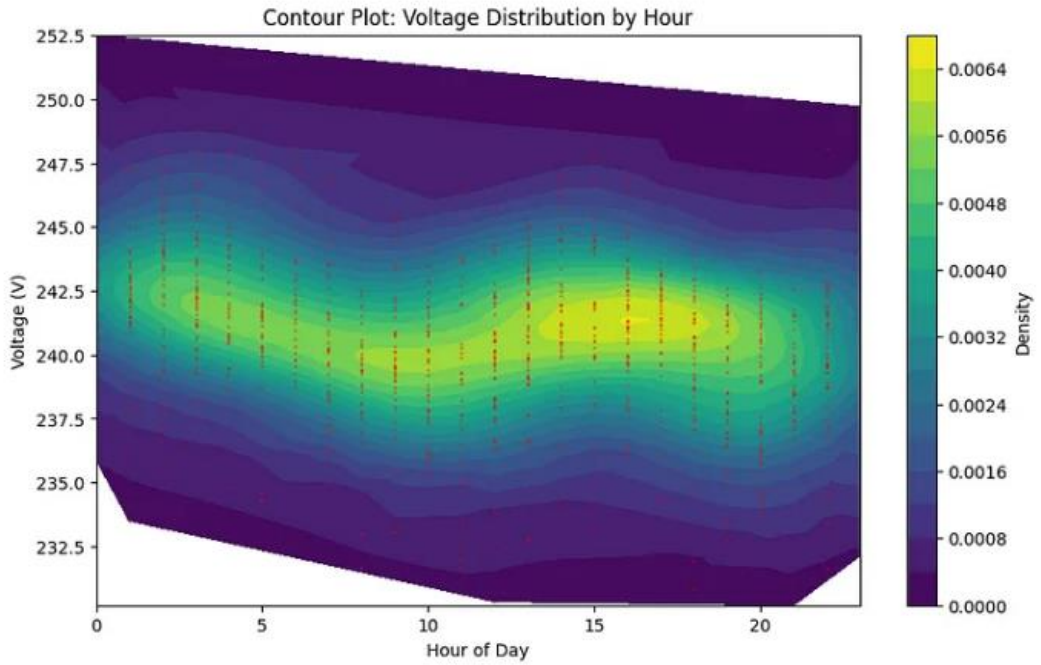


D. Contour Plot (Voltage vs. Time)

A **Contour Plot**, also known as an **isoline plot** (for 2D) or **isosurface plot** (for 3D), is a graphical technique used to represent a **three-dimensional surface** by plotting constant **z values** (the third dimension) on a **two-dimensional plane**. In essence, it shows where a continuous function has the same value.

Imagine slicing through a 3D surface at different constant z-values. The intersection of each slice with the surface creates a line (in 2D projection) or a surface (which is then projected onto 2D). These lines or projected surfaces connect points of equal value and are called **contour lines** or **isocontours**.

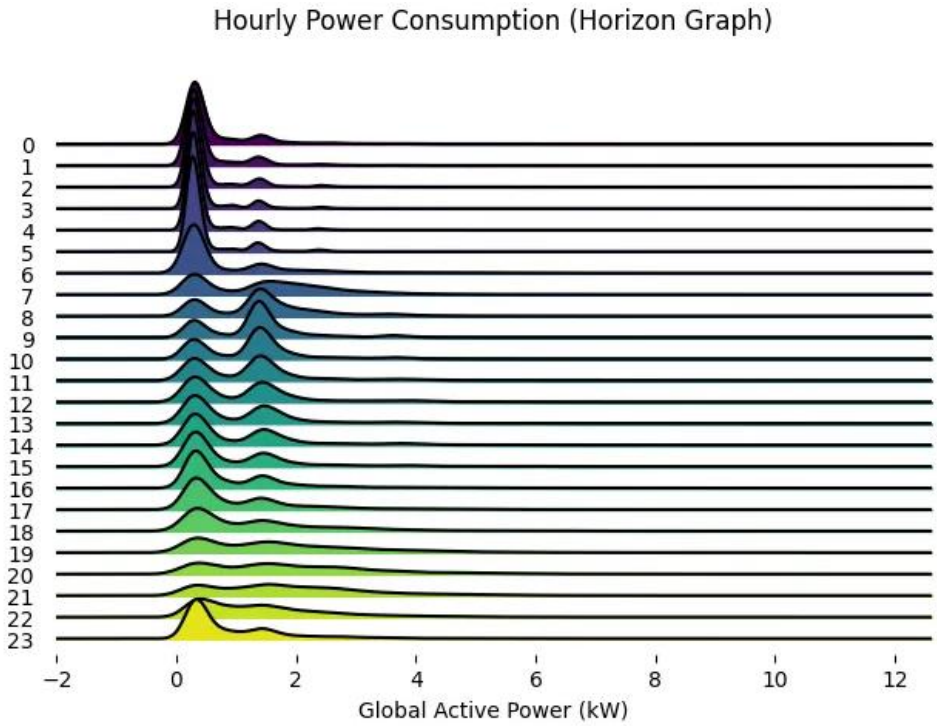
```
from scipy.stats import gaussian_kde

# Sample data for performance
sample = df.sample(1000)

# Kernel Density Estimation
x = sample['Hour']
y = sample['Voltage']
xy = np.vstack([x, y])
z = gaussian_kde(xy)(xy)

plt.figure(figsize=(10, 6))
plt.tricontourf(x, y, z, levels=15, cmap="viridis")
plt.colorbar(label="Density")
plt.scatter(x, y, c='red', s=1, alpha=0.3)
plt.title("Contour Plot: Voltage Distribution by Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Voltage (V)")
plt.show()
```



E. Horizon Graphs

Horizon Graphs are a space-efficient visualization technique designed to display the trends of multiple time series data within a limited vertical space while preserving readability and allowing for easy comparison. They achieve this by **folding and layering the time series data along the vertical axis**, using color to differentiate the layers and indicate whether the values are above or below a baseline (typically zero).

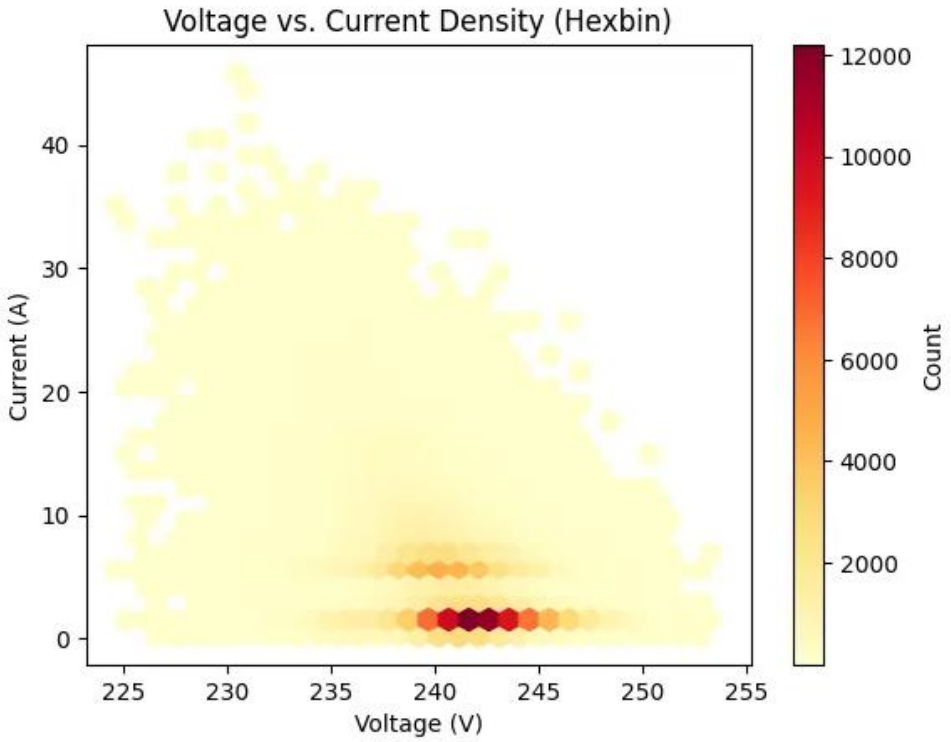
```
!pip install joypy
import joypy
import matplotlib.pyplot as plt # Make sure matplotlib.pyplot is imported

plt.figure(figsize=(12, 6))
joypy.joyplot(
    df,
    by='Hour',
    column='Global_active_power',
    colormap=plt.cm.viridis, # Changed to plt.cm.viridis
    title="Hourly Power Consumption (Horizon Graph)"
)
plt.xlabel("Global Active Power (kW)")
plt.show()
```



F. Hexagonal Binning

Hexagonal Binning, also known as a **hexbin plot**, is a visualization technique used to represent the **density of data points in a two-dimensional scatter plot**. Instead of plotting each individual point, which can lead to overplotting in dense areas and make it difficult to discern patterns, hexagonal binning divides the 2D space into a grid of **regular hexagons** and then **counts the number of data points that fall within each hexagon**. The density within each hexagon is then typically represented by a **color intensity**, where darker or more saturated colors indicate a higher concentration of points.

```
plt.hexbin(
    df['Voltage'],
    df['Global_intensity'],
    gridsize=30,
    cmap='YlOrRd',
    mincnt=1
)
plt.colorbar(label='Count')
plt.title("Voltage vs. Current Density (Hexbin)")
plt.xlabel("Voltage (V)")
plt.ylabel("Current (A)")
plt.show()
```



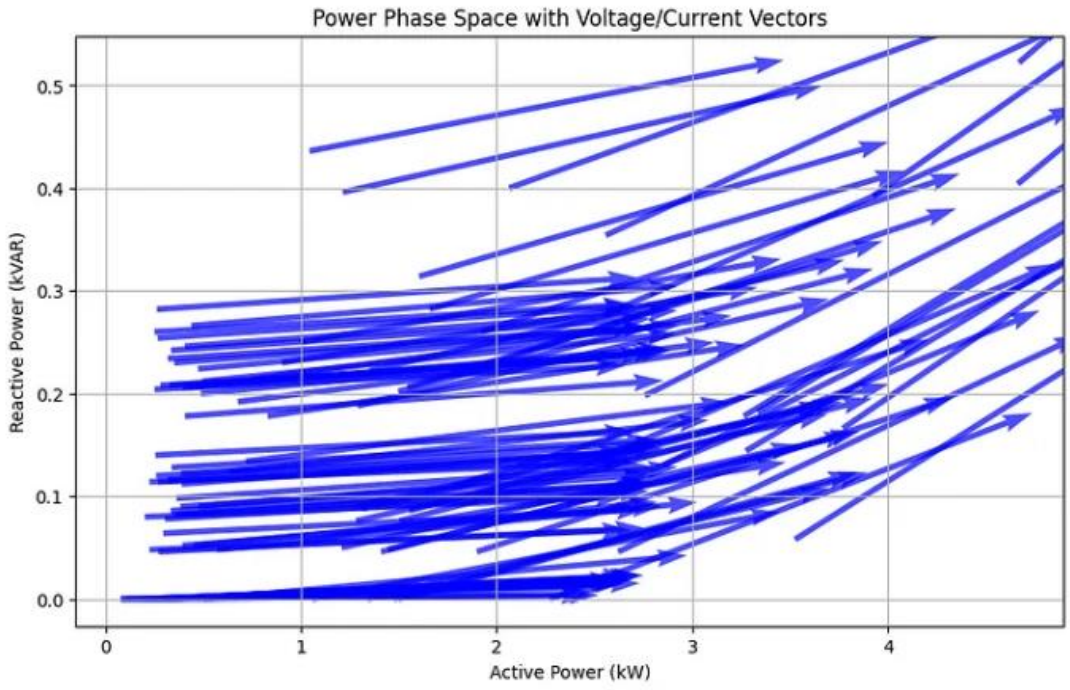
2.2 VECTOR VISUALIZATIONS

Vector visualization deals with data that has both **magnitude** and **direction** at each point in a dataset. Unlike scalar data, where each point has a single numerical value, vector data associates each location with a vector, which is typically represented by an arrow. The properties of the arrow (length and orientation) directly correspond to the magnitude and direction of the vector at that point.

A. Arrow Plot (Active vs. Reactive Power)

An **Arrow Plot**, also known as a **Vector Field Plot**, is a fundamental vector visualization technique used to display **vector data** on a two-dimensional (or sometimes three-dimensional) plane. It represents the **magnitude and direction** of a vector at discrete points in a spatial domain using **arrows**.

```
sample = df.sample(100)

plt.figure(figsize=(10, 6))
plt.quiver(
    sample['Global_active_power'],
    sample['Global_reactive_power'],
    sample['Voltage'] / 10,
    sample['Global_intensity'],
    scale=50, color='blue', alpha=0.7
)
plt.xlabel("Active Power (kW)")
plt.ylabel("Reactive Power (kVAR)")
plt.title("Power Phase Space with Voltage/Current Vectors")
plt.grid()
plt.show()
```

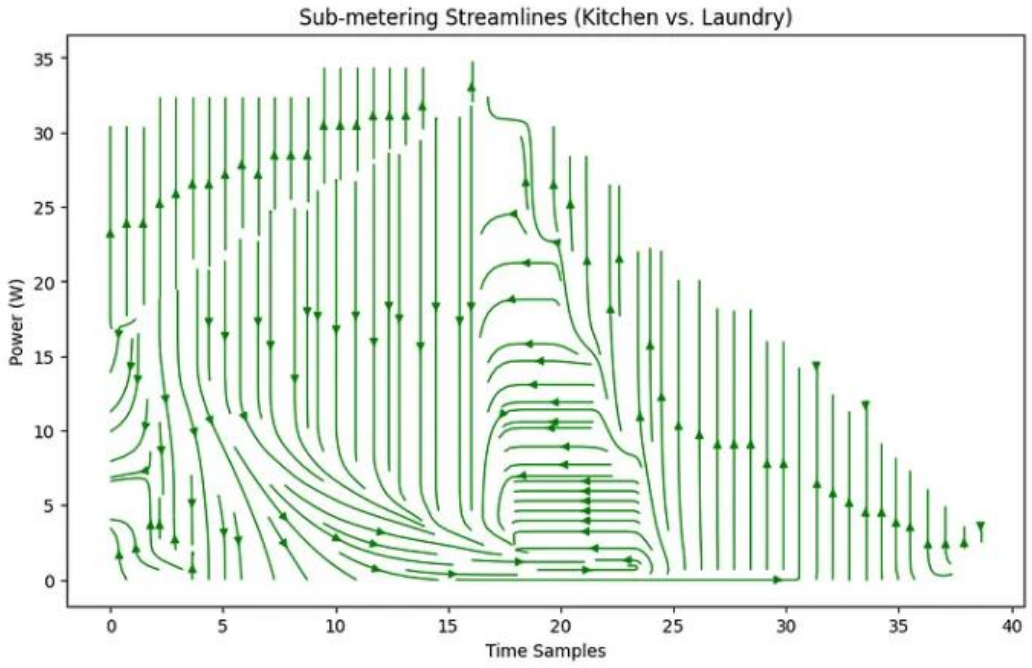


B. Streamlines (Sub-metering Relationships)

Streamlines

Streamlines are a type of vector visualization specifically used to depict the **instantaneous direction of a vector field** at a given point in time. In the context of fluid flow (which is where they are most commonly used and understood), a streamline is an imaginary curve that is **everywhere tangent to the instantaneous velocity vector** at each point along the curve.

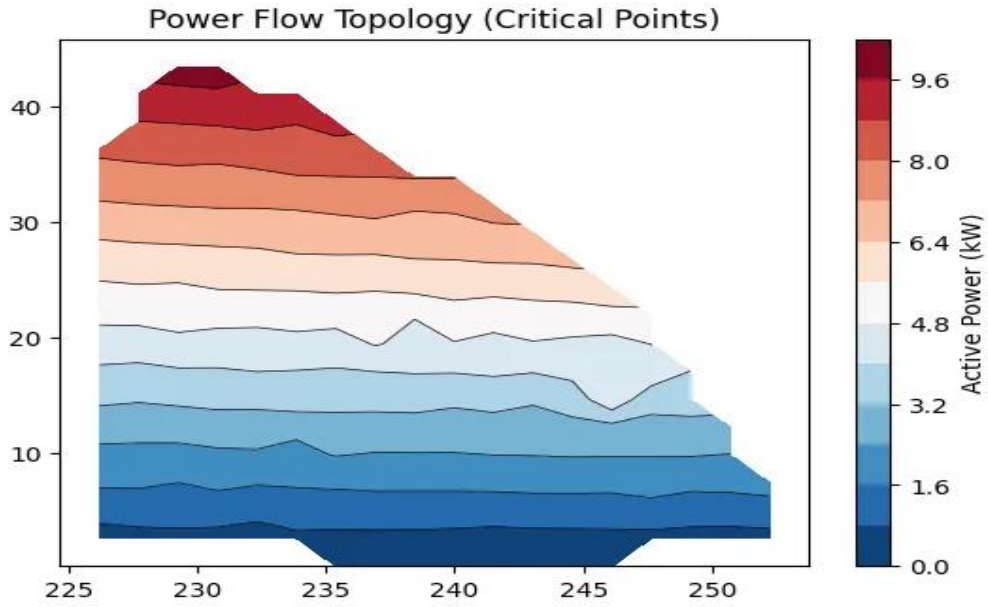
```
x = df['Sub_metering_1'].values[:500]
y = df['Sub_metering_2'].values[:500]
u = np.gradient(x)  # Rate of change
v = np.gradient(y)

# Create a grid for streamplot
# This ensures 'u' and 'v' will match the grid shape
xi = np.linspace(x.min(), x.max(), 25)
yi = np.linspace(y.min(), y.max(), 20)
X, Y = np.meshgrid(xi, yi)

# Interpolate 'u' and 'v' onto the grid
from scipy.interpolate import griddata
U = griddata((x, y), u, (X, Y), method='linear')
V = griddata((x, y), v, (X, Y), method='linear')


plt.figure(figsize=(10, 6))
# Use the grid and interpolated values for streamplot
plt.streamplot(X, Y, U, V, density=2, color='green', linewidth=1)

plt.title("Sub-metering Streamlines (Kitchen vs. Laundry)")
plt.xlabel("Time Samples")
plt.ylabel("Power (W)")
plt.show()
```



C. Vector Field Topology

Vector Field Topology is the study and visualization of the **qualitative structure** of vector fields. Instead of focusing on the precise magnitude and direction at every point, it aims to identify and characterize the **critical points** (singularities) and the **invariant structures** (separatrices) that organize the flow or behavior described by the vector field. Understanding the topology provides a high-level overview of the field's global behavior and its key features.

```
from scipy.interpolate import griddata

# Create grid for topology analysis
xi = np.linspace(df['Voltage'].min(), df['Voltage'].max(), 20)
yi = np.linspace(df['Global_intensity'].min(), df['Global_intensity'].max(), 20)
zi = griddata(
    (df['Voltage'], df['Global_intensity']),
    df['Global_active_power'],
    (xi[None,:], yi[:,None]),
    method='cubic'
)

plt.contour(xi, yi, zi, levels=15, linewidths=0.5, colors='k')
plt.contourf(xi, yi, zi, levels=15, cmap="RdBu_r")
plt.title("Power Flow Topology (Critical Points)")
plt.colorbar(label="Active Power (kW)")
plt.show()
```



2.3 MULTI-DIMENSIONAL VISUALIZATIONS

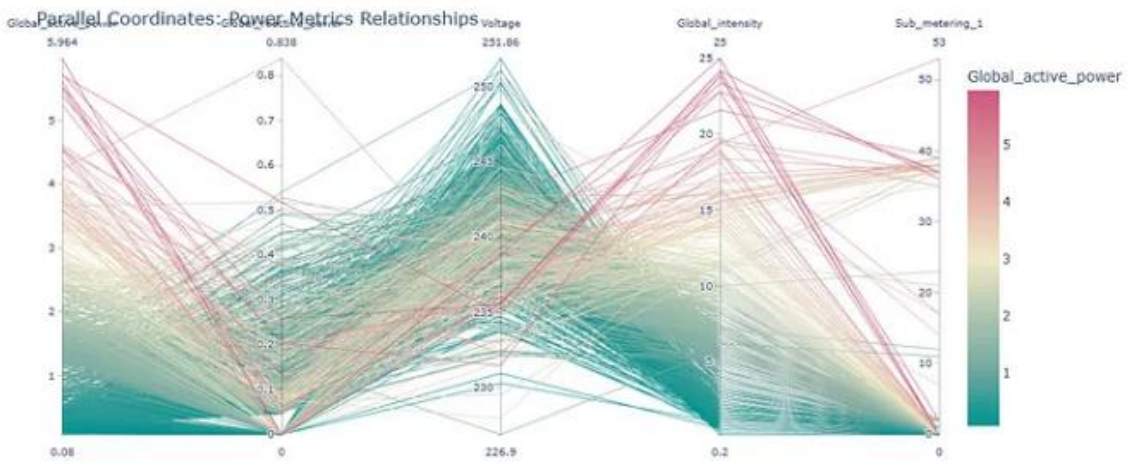
Multi-Dimensional Visualizations are techniques used to represent datasets with **more than two variables** in a single visual display. Since our physical world and typical display devices are limited to two or three spatial dimensions, these techniques employ various visual encoding strategies to map additional data dimensions onto visual attributes like position, size, shape, color, orientation, texture, and animation. The goal is to enable the exploration of complex relationships, patterns, and correlations that might be hidden when examining variables in isolation or through simple 2D or 3D plots.

Many real-world datasets are inherently multi-dimensional. For example, a dataset about cars might include variables like price, fuel efficiency, horsepower, weight, number of cylinders, safety rating, and origin. To understand how these factors interact and influence each other, we need visualization methods that can handle more than just two or three of these variables at once.

A. Parallel Coordinates (All Power Metrics)

Parallel Coordinates is a multi-dimensional visualization technique used to represent and explore datasets with **multiple quantitative variables**. In this method, each variable is depicted as a separate, parallel vertical axis. Each data point in the dataset is then represented as a polyline that intersects each axis at the point corresponding to its value for that variable.

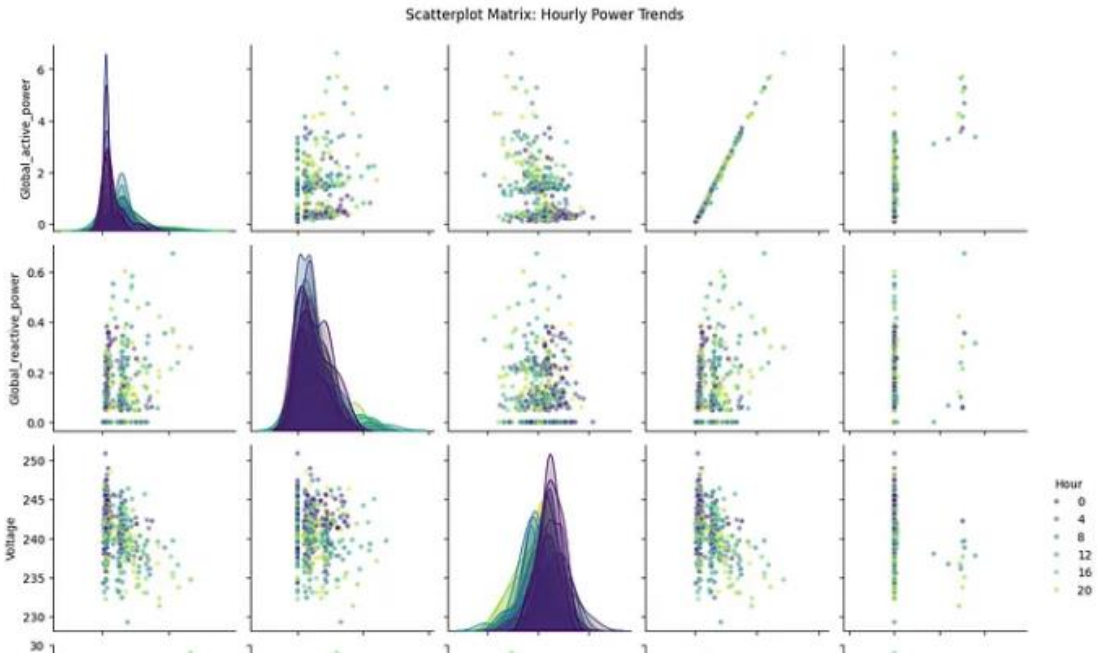
```
metrics = ['Global_active_power', 'Global_reactive_power',
           'Voltage', 'Global_intensity', 'Sub_metering_1']

fig = px.parallel_coordinates(
    df.sample(1000),
    dimensions=metrics,
    color='Global_active_power',
    color_continuous_scale=px.colors.diverging.Tealrose
)
fig.update_layout(
    title="Parallel Coordinates: Power Metrics Relationships",
    height=500
)
fig.show()
```



B. Scatterplot Matrix (SPLOM)

Scatterplot Matrix is a valuable tool for the initial exploration of multi-dimensional data by visualizing all pairwise relationships between quantitative variables in a grid of scatter plots. While it has limitations with high dimensionality and only shows pairwise interactions, it provides a fundamental and intuitive way to identify potential correlations, patterns, and outliers within a dataset.

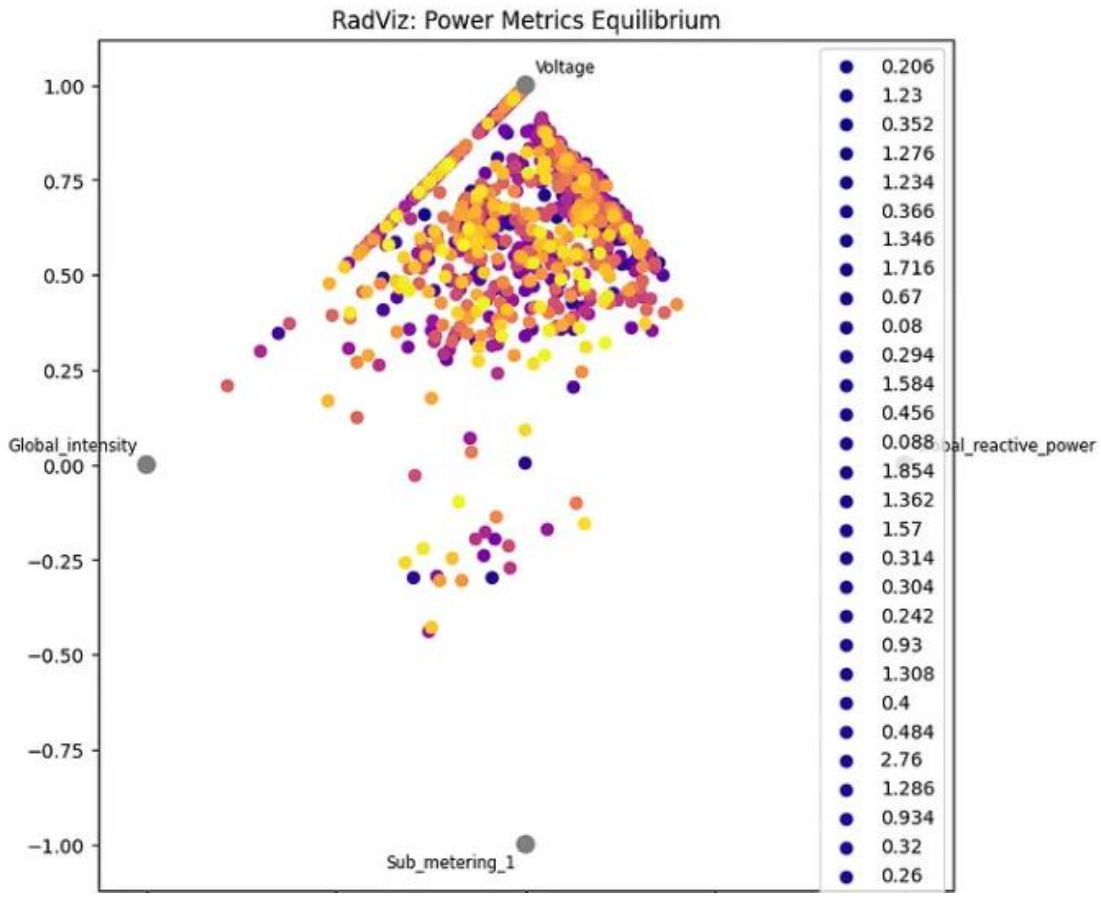
```
sns.pairplot(
    df[metrics + ['Hour']].sample(500),
    hue='Hour', palette="viridis",
    plot_kws={'alpha': 0.5, 's': 20}
)
plt.suptitle("Scatterplot Matrix: Hourly Power Trends", y=1.02)
plt.show()
```



C. RadViz (Radial Coordinates)

RadViz is a useful multi-dimensional visualization technique that projects high-dimensional data onto a 2D circular layout based on weighted averages related to each dimension's anchor point. It can effectively reveal clusters and the influence of individual dimensions, but its non-linear projection and sensitivity to normalization require careful consideration during interpretation.

```
from pandas.plotting import radviz

plt.figure(figsize=(8, 8))
radviz(df.sample(1000)[metrics], 'Global_active_power', colormap='plasma')
plt.title("RadViz: Power Metrics Equilibrium")
plt.show()
```



For more information visit : [Data Visualization Techniques: From Scalars to Multi-Dimensional Mastery | by Himanshu Surendra Rajak | May, 2025 | Medium]

2. 4 CLUSTER VISUALIZATIONS

Clustering visualization is a method used to represent the groups or clusters formed by clustering algorithms in a visual format. This technique is widely used in data analysis and machine learning, particularly in unsupervised learning where the goal is to discover hidden patterns or structures in unlabelled data.

There are several clustering algorithms available, each with its unique way of grouping data. Some of the most popular ones include K-means, Hierarchical Clustering, DBSCAN, and PaCMAP. These algorithms can handle multidimensional data, making them suitable for complex datasets.

Visualizing these clusters can help us understand the data better. For instance, we can identify which data points are similar to each other, how they are grouped, and how these groups are different from each other. This information can be invaluable in many applications, such as document analysis, spam filtering, and detecting fraudulent activity.

Common Applications of Clustering Visualization

Clustering visualization has a wide range of applications. In document analysis, for example, clustering can group similar documents together, making it easier to manage and retrieve information. Visualizing these clusters can further enhance our understanding of the document corpus.

In marketing and sales, clustering visualization can help understand customer behavior. By grouping customers based on their purchasing patterns, businesses can tailor their marketing strategies to target specific customer groups effectively.

Clustering visualization is also used in spam filtering. By clustering emails based on their content, spam filters can identify and block spam emails more effectively. Visualizing these clusters can help improve the spam filter's performance by identifying features that distinguish spam emails from legitimate ones.

Clustering Visualization Techniques and Tools

There are several techniques for visualizing clustering results. The choice of technique depends on the nature of the data and the specific requirements of the task at hand. Some common techniques include scatter plots, dendrograms, and heatmaps.

Scatter plots are commonly used to visualize clusters in two

or three-dimensional data. Dendrograms are used for hierarchical clustering, showing the hierarchical relationship between clusters. Heatmaps, on the other hand, are useful for visualizing high-dimensional data, with colors representing different values in the dataset.

There are also several tools available for clustering visualization. Python, for instance, offers libraries like Matplotlib and Seaborn that provide various functions for data visualization. These libraries can be used to create scatter plots, dendrograms, heatmaps, and more.

In Python's Matplotlib, for example, we can use the scatter function to create a scatter plot of our data. Each point in the plot represents a data point, and the color of the point indicates its cluster. This can be a powerful way to visualize the results of our clustering algorithm.

Popular Clustering Algorithms Used for Visualization

There are several clustering algorithms that are commonly used for visualization. Each of these algorithms has its strengths and weaknesses, and the choice of algorithm depends on the specific requirements of your task.

K-means

K-means is a simple and efficient algorithm that partitions the data into K distinct clusters based on distance to the centroid of the clusters. The algorithm iteratively assigns each data point to the nearest centroid and recalculates the centroids until the clusters are stable. However, K-means assumes that clusters are spherical and equally sized, which might not always be the case in real-world data.

Hierarchical Clustering

Hierarchical clustering creates a tree of clusters, which can be visualized using a dendrogram. This algorithm can be either agglomerative (bottom-up) or divisive (top-down). Agglomerative clustering starts with each data point as a separate cluster and merges the closest pairs of clusters until only one cluster (or K clusters) remain. Hierarchical clustering can capture complex cluster structures, but it can be slower than K-means for large datasets.

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm. It groups together data points that are close in the data space and have a minimum number of neighbors. DBSCAN can find arbitrarily shaped clusters and can identify noise (outliers). However, it may not perform well when clusters have different densities.

PaCMAP

PaCMAP (Pairwise Controlled Manifold Approximation Projection) is a relatively new algorithm for dimensionality reduction and visualization. It aims to preserve both the local and global structure of the data, making it suitable for visualizing complex, high-dimensional data. PaCMAP can be used as a preprocessing step for clustering, helping to reduce the dimensionality of the data while preserving its structure.

clustering visualization is a powerful tool in the field of data analysis and machine learning. It provides a way to visually represent the structure of data, making it easier to understand and interpret. By using appropriate clustering algorithms and following best practices for

visualization, you can uncover hidden patterns and insights in your data, leading to more effective decision-making and strategy planning.

2. 5 MATRIX VISUALIZATION IN BAYESIAN DATA ANALYSIS

In **Bayesian data analysis**, **matrix visualizations** are powerful tools for understanding the structure and relationships between parameters, posterior distributions, and model outputs. These visualizations are especially useful when dealing with **multivariate distributions** or high-dimensional parameter spaces.

What is Matrix Visualization?

Matrix visualization typically refers to **grid-like plots** showing pairwise relationships, marginal distributions, or correlation structures among multiple variables. The most common matrix visualizations in Bayesian data analysis are:

Pair Plot / Corner Plot (Joint and Marginal Posterior Distributions)

These show histograms of marginal posterior distributions along the diagonal and scatterplots of joint posteriors in the off-diagonal.

- **Use:**
  - Visualize posterior samples from MCMC
  - Check correlations between parameters
  - Identify parameter identifiability or multimodality

```python
import seaborn as sns
import pandas as pd
import numpy as np

# Simulated posterior samples
data = pd.DataFrame({
    'alpha': np.random.normal(0, 1, 1000),
    'beta': np.random.normal(3, 0.5, 1000),
    'sigma': np.random.exponential(1, 1000)
})

# Pairplot
sns.pairplot(data)
```

2. **Correlation Matrix Plot**

Used to understand linear dependencies among parameters.

```python
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = data.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title("Parameter Correlation Matrix")
plt.show()
```

When is Matrix Visualization Useful in Bayesian Analysis?

- During **model checking** (e.g., trace plots and autocorrelation)
- To explore **parameter dependence** (pair plots, correlation matrices)
- To **summarize high-dimensional posterior distributions**
- For **diagnosing sampling issues** (e.g., lack of convergence, strong correlation)

Summary Table

Matrix Visualization Type	Purpose	Tool/Library
Pair plot / Corner plot	Visualize joint/marginal posteriors	Seaborn, ArviZ
Correlation Matrix	Check parameter dependency	Seaborn, Matplotlib
Trace Plot Matrix	Convergence & sampling diagnostics	ArviZ
Posterior Predictive Check	Model validation	ArviZ

.

UNIT III

Getting Started with Pandas: Arrays and vectorized computation, Introduction to pandas Data Structures, Essential Functionality, Summarizing and Computing Descriptive Statistics. Data Loading, Storage and File Formats. Reading and Writing Data in Text Format, Web Scraping, Binary Data Formats, Interacting with Web APIs, Interacting with Databases Data Cleaning and Preparation. Handling Missing Data, Data Transformation, String Manipulation

GETTING STARTED WITH PANDAS

Pandas is open-source Python library which is used for data manipulation and analysis. It consists of data structures and functions to perform efficient operations on data. It is well-suited for working with **tabular data** such as **spreadsheets** or **SQL tables**. It is used in data science because it works well with other important libraries. **It is built on top of the NumPy library** as it makes easier to manipulate and analyze. Pandas is used in other libraries such as:

- **Matplotlib** for plotting graphs
- **SciPy** for statistical analysis
- **Scikit-learn** for machine learning algorithms.
- It uses many functionalities provided by **NumPy library**.

Here is a various task that we can do using Pandas:

- **Data Cleaning, Merging and Joining**: Clean and combine data from multiple sources, handling inconsistencies and duplicates.
- **Handling Missing Data**: Manage missing values (NaN) in both floating and non-floating point data.
- **Column Insertion and Deletion**: Easily add, remove or modify columns in a DataFrame.
- **Group By Operations**: Use "split-apply-combine" to group and analyze data.
- **Data Visualization**: Create visualizations with Matplotlib and Seaborn, integrated with Pandas.

Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.

Install it using this command:

C:\Users*Your Name*>pip install pandas

Import Pandas

Once Pandas is installed, import it in your applications by adding the import keyword:

import pandas

Now Pandas is imported and ready to use.

Example

```
import pandas

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

3.1 ARRAYS AND VECTORIZATION

In **Pandas**, arrays and vectorized computation allow for efficient and concise data manipulation, especially when working with large datasets. Here's a clear explanation of these concepts:

1. Arrays in Pandas

Pandas is built on **NumPy**, which uses powerful array structures (ndarray). In Pandas:

- **Series** is a one-dimensional labeled array.
- **DataFrame** is a two-dimensional labeled data structure with columns of potentially different types.

Each Series or column in a DataFrame is essentially a NumPy array with some added features (like labels).

```
import pandas as pd

data = pd.Series([1, 2, 3, 4])

print(data)
```

2. Vectorized Computation

Vectorized computation means applying operations to entire arrays (Series/DataFrames) at once, without explicit loops. This is efficient and faster due to internal optimizations.

Example: Arithmetic Operations

```
import pandas as pd

s = pd.Series([1, 2, 3, 4])

print(s + 5)        # Adds 5 to each element

print(s * 2)        # Multiplies each element by 2
```

```
print(s ** 2)        # Squares each element
```

On DataFrames:

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})


print(df + 10)       # Adds 10 to each element
print(df['A'] + df['B'])  # Element-wise addition of two columns
```

3. Boolean Operations (Filtering)

```
s = pd.Series([10, 20, 30, 40])
print(s[s > 25])   # Filters elements greater than 25
```

4. Applying NumPy Functions

```
import numpy as np
s = pd.Series([1, 2, 3])
print(np.exp(s))   # Exponential of each element
print(np.log(s))   # Natural log of each element
```

Benefits of Vectorized Computation in Pandas

- Faster performance (uses C-based implementations)
- Cleaner, more concise code
- Avoids slow Python loops

3.2 INTRODUCTION TO PANDAS DATA STRUCTURES

Pandas is an open-source Python library used for working with relational or labeled data in an easy and intuitive way. It provides powerful data structures and a wide range of operations for manipulating numerical data and time series. Pandas also offers tools for cleaning, processing and analyzing data efficiently. It is one of the most popular libraries for data analysis in Python and primarily supports two core data structures

- Series

- DataFrame

Series

A **Series** is a one-dimensional array-like object that can store any data type such as integers, strings, floats, or even Python objects. It comes with labels (called an index).

Syntax

pandas.Series(data=None, index=None, dtype=None, name=None, copy=False)

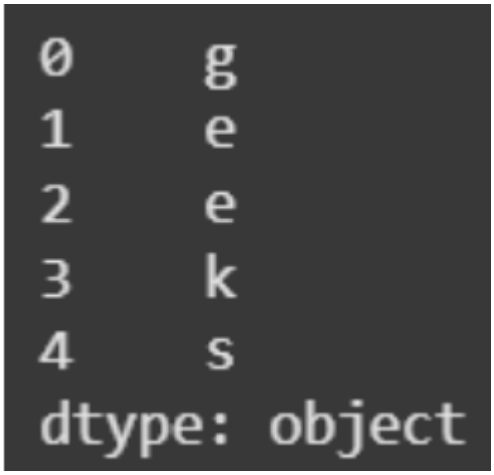
Parameters:

data: Array-like, dict or scalar – Input data.
index (Optional): Labels for the axis.
dtype (Optional): Data type of the Series.
name (Optional): Name of the Series.
copy (Bool): Copy data if True.
Returns: A pandas.Series object containing the provided data with an associated index.

Example 1: Series holding the char data type.

```
import pandas as pd
a = ['g', 'e', 'e', 'k', 's']

res = pd.Series(a)
print(res)
```

Output



Explanation: We pass the list **a** into **pd.Series(a)**, which converts it into a Series (a column-like structure) where each item gets a default index starting from 0, automatically assigned by Pandas.

Example 2: Series holding the Int data type.

```
import pandas as pd
a = [1,2,3,4,5]

res = pd.Series(a)
print(res)
```

Output

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Explanation: We pass the list a into pd.Series **a**, which converts it into a Series (a column-like structure) where each number gets a default index starting from 0, automatically assigned by Pandas.

Example 3: Series holding the dictionary.

```
import pandas as pd
a = { 'Id': 1013, 'Name': 'MOhe', 'State': 'Maniput','Age': 24}

res = pd.Series(a)
print(res)
```

Output

```
Id            1013
Name          MOhe
State      Maniput
Age             24
dtype: object
```

Series Output

Explanation: We pass the dictionary **a** into **pd.Series(a)**, converting keys into index labels and values into data, creating a labeled Series for easy access.

Dataframe

A **DataFrame** is a two-dimensional, size-mutable and heterogeneous tabular data structure with labeled rows and columns, similar to a spreadsheet or SQL table. Each column in a DataFrame is a Pandas Series, allowing you to work with multiple types of data in one table.

Syntax:

pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)

Parameters:

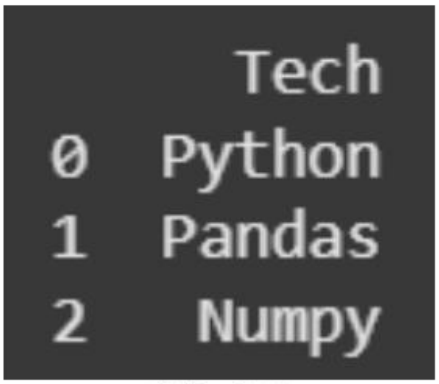
- **data:** Various forms of input data (e.g., lists, dict, ndarray, Series, another DataFrame).
- **index(Optional):** labels for rows.
- **columns(Optional):** labels for columns.
- **dtype(Optional):** Optional data type for all columns.
- **copy(Optional):** Boolean; whether to copy data or not.

Returns: A pandas.DataFrame object representing a 2D labeled data structure.

Example 1: Creating a dataFrame from a list

```
import pandas as pd
a = ['Python', 'Pandas', 'Numpy']

df = pd.DataFrame(a, columns=['Tech'])
print(df)
```

Output



DataFrame Output

Explanantion: We pass the list **a** into **pd.DataFrame(a, columns=['Tech'])**, which converts it into a DataFrame with a single column named 'Tech'. Each item becomes a row and Pandas automatically assigns a default integer index starting from 0.

Explanantion: We pass the list **a** into **pd.DataFrame(a, columns=['Tech'])**, which converts it into a DataFrame with a single column named 'Tech'. Each item becomes a row and Pandas automatically assigns a default integer index starting from 0.

Example 2: Creating a dataFrame from a dictionary

```
a = {
    'Name': ['Tom', 'Nick', 'Krish', 'Jack'],
    'Age': [20, 21, 19, 18]
}
res = pd.DataFrame(a)
print(res)
```

Output

	Name	Age
0	Tom	20
1	Nick	21
2	Krish	19
3	Jack	18

DataFrame Output

Explanation: We pass the dictionary **a** into **pd.DataFrame(a)**, which converts it into a DataFrame where the dictionary keys become column names and the values (lists) become the column data. Pandas assigns a default integer index starting from 0 for the rows.

Example 3: Selecting columns and rows in a dataFrame

```
import pandas as pd

a = {
    'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age': [27, 24, 22, 32],
    'Address': ['Delhi', 'Kanpur', 'Allahabad', 'Kannauj'],
    'Qualification': ['Msc', 'MA', 'MCA', 'Phd']
}
df = pd.DataFrame(a)
print(df[['Name', 'Qualification']])
```

Output

	Name	Qualification
0	Jai	Msc
1	Princi	MA
2	Gaurav	MCA
3	Anuj	Phd

Selected Columns

Explanation: We create a DataFrame **df** from the dictionary **a**, then select and print only the columns 'Name' and 'Qualification' by passing their names in a list to **df[]**. This returns a new DataFrame with just those two columns.

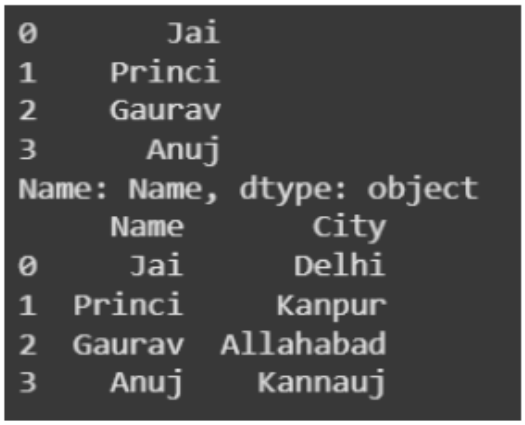
Accessing columns and rows in a dataFrame

A DataFrame in Pandas is a 2D tabular structure where you can easily access and manipulate data by selecting specific columns or rows. You can extract one or more columns using column names and filter rows using labels or conditions.

Example 1: We can access one or more columns in a DataFrame using square brackets.

```
import pandas as pd
a = {
    'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],
    'Age': [27, 24, 22, 32],
    'City': ['Delhi', 'Kanpur', 'Allahabad', 'Kannauj']
}
df = pd.DataFrame(a)

print(df['Name']) # single column
print(df[['Name', 'City']]) # multiple columns
```

Output



Column Access

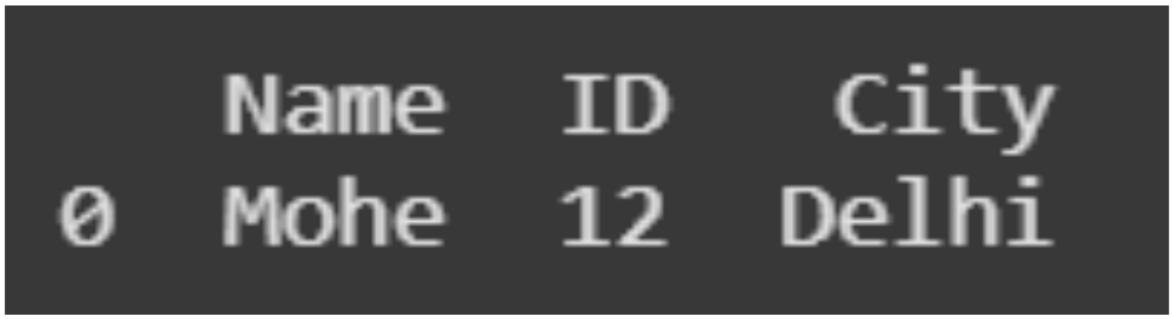
Explanation:

- **df['Name']** returns a Series containing values from the 'Name' column.
- **df[['Name', 'City']]** returns a new DataFrame containing only the specified columns.

Example 2: We can use .loc[] to access rows by index or filter them using conditions.

```
import pandas as pd

a = {
    'Name': ['Mohe', 'Shyni', 'Parul', 'Sam'],
    'ID': [12, 43, 54, 32],
    'City': ['Delhi', 'Kochi', 'Pune', 'Patna']
}

df = pd.DataFrame(a)
res = df.loc[df['Name'] == 'Mohe']
print(res)
```

Output



Explanation: df.loc[df['Name'] == 'Mohe'] filters and returns only the row(s) where the 'Name' column has the value 'Mohe'.

3.3 PANDAS: ESSENTIAL FUNCTIONALITY

Pandas provides a rich set of functions for efficient and flexible data manipulation. These core features help in cleaning, transforming, and analyzing data.

Pandas functions in Python and methods that are essential for every Data Analyst and Data Scientist to know are

read_csv()

This is one of the most crucial pandas methods in Python. read_csv() function helps read a comma-separated values (csv) file into a Pandas DataFrame. All you need to do is mention the path of the file you want it to read. It can also read files separated by delimiters other than comma, like | or tab.

Python Code:

```
# importing library
import pandas as pd

# reading the dataset
data_1 = pd.read_csv(r'blog_dataset.csv')
```

The data has been read from the data source into the Pandas DataFrame. You will have to change the path of the file you want to read. to_csv() function works exactly opposite of read_csv(). It helps to write data contained in a Pandas DataFrame or Series to a csv file. You can read more about to_csv() here. read_csv() and to_csv() are one of the most used functions in Pandas because they are used while reading data from a data source, and are very important to know.

2. head()

head(n) is used to return the first n rows of a dataset. By default, df.head() will return the first 5 rows of the DataFrame. If you want more/less number of rows, you can specify n as an integer.

data_1.head(6)

Output:

	Name	Age	City	State	DOB	Gender	City temp	Salary
0	Alam	29	Indore	Madhya Pradesh	20-11-1991	Male	35.5	50000
1	Rohit	23	New Delhi	Delhi	19-09-1997	Male	39.0	85000
2	Bimla	35	Rohtak	Haryana	09-01-1985	Female	39.7	20000
3	Rahul	25	Kolkata	West Bengal	19-09-1995	Male	36.5	40000
4	Chaman	32	Chennai	Tamil Nadu	12-03-1988	Male	41.1	65000
5	Vivek	38	Gurugram	Haryana	22-06-1982	Male	38.9	35000

The first 6 rows (indexed 0 to 5) are returned as output as per expectation.

tail() is similar to head(), and returns the bottom n rows of a dataset. head() and tail() help you get a quick glance at your dataset, and check if data has been read into the DataFrame properly.

3. describe()

describe() is used to generate descriptive statistics of the data in a Pandas DataFrame or Series. It summarizes central tendency and dispersion of the dataset. describe() helps in getting a quick overview of the dataset.

data_1.describe()

Output:

	Age	City temp	Salary
count	9.000000	8.000000	9.000000
mean	32.000000	38.575000	44444.444444
std	5.894913	1.771803	21360.659582
min	23.000000	35.500000	18000.000000
25%	29.000000	38.300000	35000.000000
50%	32.000000	38.950000	40000.000000
75%	38.000000	39.175000	52000.000000
max	39.000000	41.100000	85000.000000

describe() lists out different descriptive statistical measures for all numerical columns in our dataset. By assigning the include attribute the value 'all', we can get the description to include all columns, including those containing categorical information.

4.astype()

astype() is used to cast a Python object to a particular data type. It can be a very helpful function in case your data is not stored in the correct format (data type). For instance, if floating point numbers have somehow been misinterpreted by Python as strings, you can convert them back to floating point numbers with astype(). Or if you want to convert an object datatype to category, you can use astype().

```
data_1['Gender'] = data_1.Gender.astype('category')
```

You can verify the change in data type by looking at the data types of all columns in the dataset using the dtypes attribute.

5. **loc[:]**

loc[:] helps to access a group of rows and columns in a dataset, a slice of the dataset, as per our requirement. For instance, if we only want the last 2 rows and the first 3 columns of a dataset, we can access them with the help of loc[:]. We can also access rows and columns based on labels instead of row and column number.

```
data_1.loc[0:4, ['Name', 'Age', 'State']]
```

Output:

	Name	Age	State
0	Alam	29	Madhya Pradesh
1	Rohit	23	Delhi
2	Bimla	35	Haryana
3	Rahul	25	West Bengal
4	Chaman	32	Tamil Nadu

The above code will return the “Name”, “Age”, and “State” columns for the first 5 customer records. Keep in mind that index starts from 0 in Python, and that loc[:] is **inclusive** on both values mentioned. So 0:4 will mean indices 0 to 4, both included.

loc[:] is one of the most powerful functions in Pandas, and is a must-know for all Data Analysts and Data Scientists.

iloc[:] works in a similar manner, just that iloc[:] is **not inclusive** on both values. So iloc[0:4] would return rows with index 0, 1, 2, and 3, while loc[0:4] would return rows with index 0, 1, 2, 3, and 4.

6. value_counts()

value_counts() returns a Pandas Series containing the counts of unique values. Consider a dataset that contains customer information about 5,000 customers of a company. value_counts() will help us in identifying the number of occurrences of each unique value in a Series. It can be applied to columns containing data like State, Industry of employment, or age of customers.

data_1['State'].value_counts()

Output:

```
Haryana           3
Delhi             2
West Bengal       1
Tamil Nadu        1
Bihar             1
Madhya Pradesh    1
Name: State, dtype: int64
```

The number of occurrences of each state in our dataset has been returned in the output, as expected. value_counts() can also be used to plot bar graphs of categorical and ordinal data.

data_1['State'].value_counts(normalize=True).plot(kind='bar', title='State')

7. drop_duplicates()

drop_duplicates() returns a Pandas DataFrame with duplicate rows removed. Even among duplicates, there is an option to keep the first occurrence (record) of the duplicate or the last. You can also specify the inplace and ignore_index attribute.

data_1.drop_duplicates(inplace=True)

inplace=True makes sure the changes are applied to the original dataset. You can verify the changes by looking at the shape of the original dataset, and the modified dataset (after dropping duplicates). You will notice the number of rows have reduced from 9 to 8 (because 1 duplicate has been dropped).

8. **groupby()**

groupby() is used to group a Pandas DataFrame by 1 or more columns, and perform some mathematical operation on it. groupby() can be used to summarize data in a simple manner.

data_1.groupby(by='State').Salary.mean()

Output:

```
State
Bihar            18000
Delhi            68500
Haryana          27500
Madhya Pradesh   50000
Tamil Nadu       65000
West Bengal      40000
Name: Salary, dtype: int64
```

The above code will group the dataset by “State” column, and will return the mean age across states.

9. merge()

merge() is used to merge 2 Pandas DataFrame objects or a DataFrame and a Series object on a common column (field). If you are familiar with the concept of JOIN in SQL, merge function similar to that. It returns the merged DataFrame.

data_1.merge(data_2, on='Name', how='left')

To know more about attributes like on (including left_on and right_on), how, and suffixes,

10. sort_values()

sort_values() is used to sort column in a Pandas DataFrame (or a Pandas Series) by values in ascending or descending order. By specifying the inplace attribute as True, you can make a change directly in the original DataFrame.

data_1.sort_values(by='Name', inplace=True)

Output:

	Name	Age	City	State	DOB	Gender	City temp	Salary
0	Alam	29	Indore	Madhya Pradesh	1991-11-20	Male	35.5	50000
2	Bimla	35	Rohtak	Haryana	1985-09-01	Female	39.7	20000
4	Chaman	32	Chennai	Tamil Nadu	1988-12-03	Male	41.1	65000
6	Charu	29	New Delhi	Delhi	1992-03-18	Female	39.0	52000
7	Ganesh	39	Patna	Bihar	1981-07-12	Male	NaN	18000
3	Rahul	25	Kolkata	West Bengal	1995-09-19	Male	36.5	40000
1	Rohit	23	New Delhi	Delhi	1997-09-19	Male	39.0	85000
5	Vivek	38	Gurugram	Haryana	1982-06-22	Male	38.9	35000

you can see that the ordering of records has changed now. Records are now listed in alphabetical order of Names. sort_values() has many other attributes which can be specified.

3.4 SUMMARIZING AND DESCRIPTIVE STATISTICS IN PANDAS

Pandas provides built-in functions to quickly summarize and analyze datasets.

1. describe()

- Provides summary statistics (count, mean, std, min, quartiles, max) for numeric columns.

df.describe()

2. Summary Functions (Column-wise)

2. Summary Functions (Column-wise)

Function	Description
`count()`	Number of non-missing values
`mean()`	Average/Mean
`std()`	Standard deviation
`min()`	Minimum value
`max()`	Maximum value
`sum()`	Sum of values
`median()`	Median value
`mode()`	Most frequent value(s)
`var()`	Variance

df['Age'].mean() # Mean of Age column

df.max() # Max of each column
value_counts()

Returns count of unique values in a Series.

df['Gender'].value_counts()

4. unique() and nunique()

- unique(): Returns unique values
- nunique(): Returns count of unique values

5. Correlation and Covariance

- corr(): Correlation between columns
- cov(): Covariance between columns

df.corr()

df.cov()

Task	Function
Basic stats summary	`describe()`
Count non-null	`count()`
Mean, median, mode	`mean()`, `median()`, `mode()`
Min, max, std, var	`min()`, `max()`, `std()`, `var()`
Unique values	`unique()`, `nunique()`
Frequency count	`value_counts()`
Correlation, Covariance	`corr()`, `cov()`
Multiple stats	`agg()`

3.5 DATA LOADING IN PANDAS

Pandas makes it easy to **load, read, and write** various data formats such as CSV, Excel, SQL, JSON, and more.

1. Loading CSV Files

df = pd.read_csv('file.csv') # Reads a CSV file into a DataFrame

Common options:

- sep=',': Specify delimiter
- header=0: Row to use as column names
- names=[...]: Provide custom column names
- index_col=0: Set column as index

2. Loading Excel Files

df = pd.read_excel('file.xlsx') # Requires `openpyxl` or `xlrd`

- **Options:**
  - sheet_name='Sheet1'
  - usecols='A:C'

3. Loading from a Dictionary or List

data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}

df = pd.DataFrame(data)

4. Loading JSON Data

df = pd.read_json('data.json') # Reads data from a JSON file

5. Reading from a SQL Database

import sqlite3

conn = sqlite3.connect('mydb.db')

df = pd.read_sql('SELECT * FROM table_name', conn)

Summary Table

Format	Function	Example
CSV	`read_csv()`	`pd.read_csv('data.csv')`
Excel	`read_excel()`	`pd.read_excel('file.xlsx')`
JSON	`read_json()`	`pd.read_json('data.json')`
SQL	`read_sql()`	`pd.read_sql(query, conn)`
Clipboard	`read_clipboard()`	`pd.read_clipboard()`
Dict	`DataFrame()`	`pd.DataFrame(dict)`

3.6 STORAGE AND FILE FORMATS

Pandas supports a wide range of file formats and storage options for reading and writing structured data efficiently. These include common formats like CSV and Excel, as well as optimized formats like Parquet and HDF5 for performance and scalability.

1. CSV (Comma-Separated Values)

Read CSV:

```
import pandas as pd
df = pd.read_csv('data.csv')
```

Write CSV:

```
df.to_csv('output.csv', index=False)
```

2. Excel Files (.xls, .xlsx)

Read Excel:

df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

Write Excel:

df.to_excel('output.xlsx', index=False)

Requires openpyxl or xlrd libraries.

3. JSON (JavaScript Object Notation)

Read JSON:

df = pd.read_json('data.json')

Write JSON:

df.to_json('output.json')

Good for nested structures; supports different orientations like 'records', 'split'.

4. HTML

Read HTML tables:

df_list = pd.read_html('https://example.com/tablepage.html')

Write HTML:

df.to_html('table.html')

read_html() returns a list of DataFrames if multiple tables exist.

5. SQL Databases

Read from SQL:

df = pd.read_sql('SELECT * FROM students', connection)

Write to SQL:

df.to_sql('students_copy', connection, if_exists='replace', index=False)

3.7 READING AND WRITING DATA IN TEXT FORMAT

Pandas supports **reading from and writing to** various file formats for data input and output. This makes it easy to load data from different sources and save processed data efficiently.

1. CSV (Comma-Separated Values)

Read CSV:

df = pd.read_csv('data.csv')

Write CSV:

df.to_csv('output.csv', index=False)

2. Excel Files (.xls, .xlsx)

Read Excel:

df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

Write Excel:

df.to_excel('output.xlsx', index=False)

3. JSON (JavaScript Object Notation)

Read JSON:

df = pd.read_json('data.json')

Write JSON:

df.to_json('output.json')

4. HDF5 (Hierarchical Data Format)

Efficient for large datasets.

Write to HDF5:

df.to_hdf('data.h5', key='df', mode='w')

Read HDF5:

df = pd.read_hdf('data.h5', key='df')

5. SQL Databases

Read from SQL:

import sqlite3

conn = sqlite3.connect('mydb.db')

df = pd.read_sql('SELECT * FROM table_name', conn)

Write to SQL:

df.to_sql('table_name', conn, index=False)

In this article, we will discuss how to read text files with pandas in Python. In Python, the Pandas module allows us to load DataFrames from external files and work on them. The dataset can be in different types of files.

Text File Used

Read Text Files with Pandas

Below are the methods by which we can read text files with Pandas:

- Using read_csv()
- Using read_table()

- Using read_fwf()

Read Text Files with Pandas Using read_csv()

We will read the text file with pandas using the read_csv() function. Along with the text file, we also pass separator as a single space (‘ ’) for the space character because, for text files, the space character will separate each field. There are three parameters we can pass to the read_csv() function.

Syntax:

***Syntax**: data=pandas.read_csv('filename.txt', sep=' ', header=None, names=["Column1", "Column2"])*

Parameters:

- ***filename.txt:** As the name suggests it is the name of the text file from which we want to read data.*
- ***sep**: It is a separator field. In the text file, we use the space character(' ') as the separator.*
- ***header:** This is an optional field. By default, it will take the first line of the text file as a header. If we use header=None then it will create the header.*
- ***names:** We can assign column names while importing the text file by using the names argument.*

Example 1

In this example, we are using read_csv() function to read the csv file.

```
# Read Text Files with Pandas using read_csv()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame
df = pd.read_csv("gfg.txt", sep=" ")

# display DataFrame
print(df)
```

Output:

```
  Batsman  Bolwer
0   Rohit  Bumrah
1   Virat   Siraj
2   Rahul   Shami
3   Dhoni  Ashwin
4   Raina  Jadeja
```

Example 2

In this example, we will make the header filed equal to None. This will create a default header in the output. And take the first line of the text file as data entry. The created header name will be a number starting from 0.

```
# Read Text Files with Pandas using read_csv()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame and
# create header
df = pd.read_csv("gfg.txt", sep=" ", header=None)

# display DataFrame
print(df)
```

Output:

```
         0       1
0  Batsman  Bolwer
1    Rohit  Bumrah
2    Virat   Siraj
3    Rahul   Shami
4    Dhoni  Ashwin
5    Raina  Jadeja
```

Example 3:

In the above output, we can see it creates a header starting from number 0. But we can also give names to the header. In this example, we will see how to create a header with a name using pandas.

```
# Read Text Files with Pandas using read_csv()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame and create
# header with names
df = pd.read_csv("gfg.txt", sep=" ", header=None,
                 names=["Team1", "Team2"])

# display DataFrame
print(df)
```

Output:

```
     Team1   Team2
0  Batsman  Bolwer
1    Rohit  Bumrah
2    Virat   Siraj
3    Rahul   Shami
4    Dhoni  Ashwin
5    Raina  Jadeja
```

Read Text Files with Pandas Using read_table()

We can read data from a text file using read_table() in pandas. This function reads a general delimited file to a DataFrame object. This function is essentially the same as the read_csv() function but with the delimiter = '\t', instead of a comma by default. We will read data with the read_table function making separator equal to a single space(' ').

Syntax: *data=pandas.read_table('filename.txt', delimiter = ' ')*

Parameters:

- ***filename.txt:*** *As the name suggests it is the name of the text file from which we want to read data.*

Example: In this example, we are using read_table() function to read the table.

```
# Read Text Files with Pandas using read_table()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame
df = pd.read_table("gfg.txt", delimiter=" ")

# display DataFrame
print(df)
```

Output:

```
  Batsman  Bolwer
0   Rohit  Bumrah
1   Virat   Siraj
2   Rahul   Shami
3   Dhoni  Ashwin
4   Raina  Jadeja
```

Read Text Files with Pandas Using read_fwf()

The fwf in the read_fwf() function stands for fixed-width lines. We can use this function to load DataFrames from files. This function also supports text files. We will read data from the text files using the read_fwf() function with pandas. It also supports optionally iterating or breaking the file into chunks. Since the columns in the text file were separated with a fixed width, this read_fwf() read the contents effectively into separate columns.

Syntax: *data=pandas.read_fwf('filename.txt')*

Parameters:

- ***filename.txt:*** *As the name suggests it is the name of the text file from which we want to read data.*

Example: In this example, we are using read_fwf to read the data.

```
# Read Text Files with Pandas using read_fwf()

# importing pandas
import pandas as pd

# read text file into pandas DataFrame
df = pd.read_fwf("gfg.txt")

# display DataFrame
print(df)
```

Output:

```
   Batsman Bolwer
0   Rohit Bumrah
1    Virat Siraj
2    Rahul Shami
3   Dhoni Ashwin
4   Raina Jadeja
```

Writing a File Using Pandas

#Creating a Sample DataFrame

data = pd.DataFrame({

'id': [1, 2, 3, 4, 5, 6, 7],

'age': [27, 32, 23, 41, 37, 31, 49],

'gender': ['M', 'F', 'F', 'M', 'M', 'M', 'F'],

'occupation': ['Salesman', 'Doctor', 'Manager', 'Teacher', 'Mechanic', 'Lawyer', 'Nurse']

})

Data

	id	age	gender	occupation
0	1	27	M	Salesman
1	2	32	F	Doctor
2	3	23	F	Manager
3	4	41	M	Teacher
4	5	37	M	Mechanic
5	6	31	M	Lawyer
6	7	49	F	Nurse

Save the DataFrame we created above as a CSV file using pandas **.to_csv()** function, as shown:

#Writing to CSV file

```
data.to_csv('data.csv')
```

We can also save the DataFrame as an Excel file using pandas **.to_excel()** function, as shown:

#Writing to Excel file

```
data.to_excel('data2.xlsx')
```

Save the DataFrame we created above as a Text file using the same function that we use for CSV files:

#Writing to Text file

```
data.to_csv('data3.txt')
```

There are various other file formats that you can write your data to. For example:

- **.to_json()**
- **.to_html()**
- **.to_sql()**

Take note that this isn't an exhaustive list. There are more formats you can write to, but they are out of the scope of this article.

3.8 WEBSCRAPING USING PANDAS

Web scraping refers to the process of extracting data from websites using automated tools and scripts. Web scraping can be used for a variety of purposes, such as market research, competitor analysis, and data analysis.

Pandas is a popular data analysis library in Python that provides powerful tools for working with structured data. In this article, we will explore how to use Pandas for web scraping and how it can make the process easier and more efficient.

The Pandas read_html() Function

One of the key features of Pandas for web scraping is the read_html() function. This function allows you to read HTML tables from web pages and convert them into Pandas DataFrames. The read_html() function takes a URL as input and returns a list of all HTML tables found on the page.

Here's an example of how to use read_html() to scrape a table from a web page:

```
import pandas as pd
import matplotlib.pyplot as plt

# Wikipedia page for total wealth data
url = 'https://en.wikipedia.org/wiki/List_of_countries_by_total_wealth'

# read HTML tables from URL
tables = pd.read_html(url)

# extract the first table (which contains the wealth data)
wealth_table = tables[0]
```

In this example, we first import the Pandas library and specify the URL of the web page we want to scrape. We then call the read_html() function with the URL as input, which returns a list of all tables found on the page. We extract the first table from the list by indexing it with [0].

Data Cleaning and Manipulation with Pandas

Once you have scraped the data from a web page into a Pandas DataFrame, you can use the full power of Pandas to clean, manipulate, and analyze the data.

Here's an example of how to clean and manipulate data in a scraped DataFrame:

```
wealth_table["Total wealth (USD bn)"] = wealth_table['Total wealth (USD bn)'].replace("—",pd.NA)
# remove unnecessary columns
wealth_table = wealth_table[['Country (or area)', 'Total wealth (USD bn)']]
# remove rows with missing values
wealth_table = wealth_table.dropna()
top10 = wealth_table.head(10)
# plot a bar chart of the top 10 countries by total wealth
plt.bar(top10['Country (or area)'], top10['Total wealth (USD bn)'])
plt.xticks(rotation=90)
plt.ylabel('Total wealth (USD bn)')
plt.title('Top 10 Countries by Total Wealth')
plt.show()
```

Note that the read_html() function may not work for all web pages, especially those with complex or dynamic HTML structures.

Web scraping with Pandas can be a powerful tool for extracting and analyzing data from web pages. The read_html() function provides an easy way to scrape HTML tables, and Pandas provides a wide range of tools for cleaning, manipulating, and analyzing the data. However, it's important to be mindful of the legal and ethical implications of web scraping, as some websites may prohibit or restrict scraping activities.

3.9 BINARY DATA FORMATS

Reading binary files means reading data that is stored in a binary format, which is not human-readable. Unlike text files, which store data as readable characters, binary files store data as raw bytes. Binary files store data as a sequence of bytes. Each byte can represent a wide range of values, from simple text characters to more complex data structures like images, videos and executable programs.

Different Modes for Binary Files in Python

When working with binary files in Python, there are specific modes we can use to open them:

- **'rb':** Read binary - Opens the file for reading in binary mode.
- **'wb':** Write binary - Opens the file for writing in binary mode.
- **'ab':** Append binary - Opens the file for appending in binary mode.

Opening a Binary File

To read a binary file, you need to use Python's built-in **open()** function, but with the mode '**rb**', which stands for read binary. The '**rb**' mode tells Python that you intend to read the file in binary format, and it will not try to decode the data into a string (as it would with text files).

file = open("file_name", "rb")

After opening the binary file, you can use different methods to read its content.

Using read()

The open() function is used to open files in Python. When dealing with binary files, we need to specify the mode as 'rb' (read binary) and then use read() to read the binary file.

```
f = open('example.bin', 'rb')

bin = f.read()
print(bin)

f.close()
```

Output:

```
b"b'\\x00\\nNotoSanSha\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x
00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\\r\nx00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x0
0\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00LWFNGWp1\\'"
```

Explanation: This code opens a binary file (**example.bin**) in read binary mode ('**rb**'). It reads the entire content of the file into the variable **bin** as bytes using the **read()** method. After reading the content, it prints the binary data. Finally, it closes the file using **f.close()** to release system resources.

Using readlines()

By using **readlines()** method we can read all lines in a file. However, in binary mode, it returns a list of lines, each ending with a newline byte (b'\n').

```
with open('example.bin', 'rb') as f:
    lines = f.readlines()
    for i in lines:
        print(i)
```

Output:

```
b"b'\\x00\\nNotoSanSha\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00
00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\\r\n"
b'x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\
\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00LWFNGWp1\\'
```

Explanation:

- The code opens a binary file (example.bin) in read-binary mode ('rb').
- readlines() reads all lines from the file into a list. Each item in the list is a byte object, representing a line in the binary file.
- The for loop iterates over each line, printing it as it goes.

Reading Binary File in Chunks

Reading a binary file in chunks is useful when dealing with large files that cannot be read into memory all at once. This uses **read(size) method** which reads up to size bytes from the file. If the size is not specified, it reads until the end of the file.

```
size = 1024

with open('example.bin', 'rb') as f:
    while True:
        chunk = f.read(size)
        if not chunk:
            break

        print(chunk)
```

Output:

```
b"b'\\x00\\nNotoSanSha\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x
00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\\r\nx00
\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00
\\x00\\x00\\x00\\x00\\x00\\x00\\x00LWFNGWp1\\'"
```

Explanation:

- The file example.bin is opened in read-binary mode ('rb').
- The code reads the file in chunks of 1024 bytes using f.read(size).
- The while True loop continues until the file is fully read, breaking when no more data is available (f.read(size) returns an empty chunk). Each chunk is printed to the console.

Python provides a module named pickle which help us to read and write binary file in python. Remember : Before writing to binary file the structure (list or dictionary) needs to be converted in binary format. This process of conversion is called Pickling. Reverse process Unpickling happen during reading binary file which converts the binary format back to readable form.

Q1. Write a function bwrite() to write list of five numbers in a binary file?

```
test.py - C:/Users/user/AppData/Local/Programs/Python/Python38-32/test.py (3.8.5)
File  Edit  Format  Run  Options  Window  Help
def bwrite():
    import pickle
    f = open("data.dat", 'wb')
    d = [1,2,3,4,5]
    pickle.dump(d,f)
    f.close()

bwrite()
```

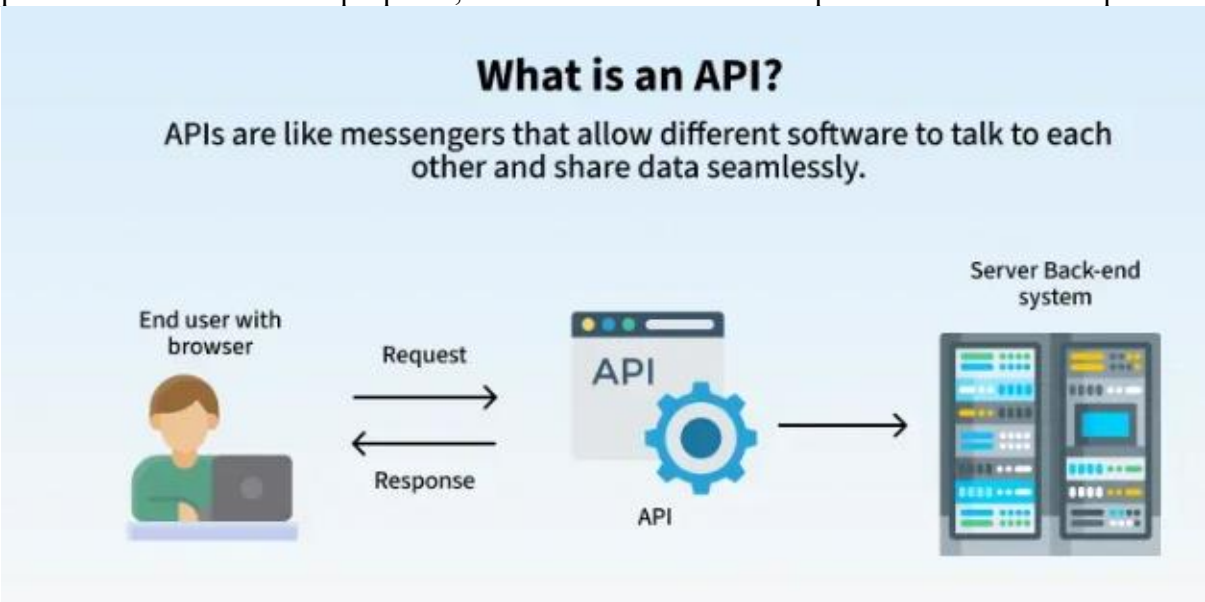
Line wise explanation of the above code :
1. Line one is just to define our function.
2. In line 2 we are importing a module pickle which help to read and write data to binary file.
3. In line number
3, we are opening a file named “data.dat” in writing mode and ‘f’ is our file object/handle.
4. Line number 4, we are declaring a list which needs to be write in a file.
5. This line is doing the main role of writing in a file. dump function of pickle module is used to write list in binary file.
6. This line is simply closing the file.
7. Last line is to calling function bwrite()

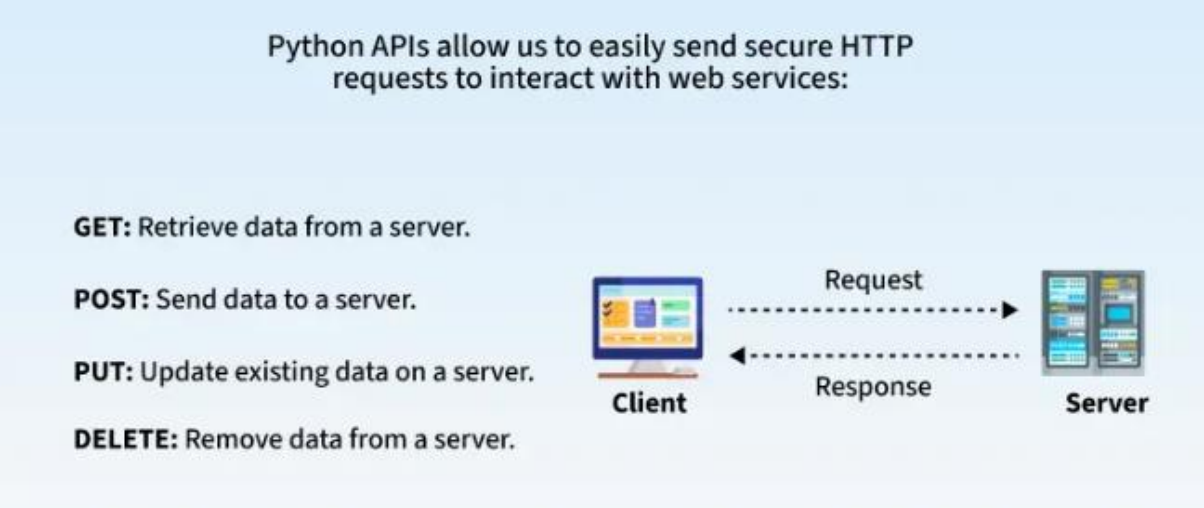
Output of above program

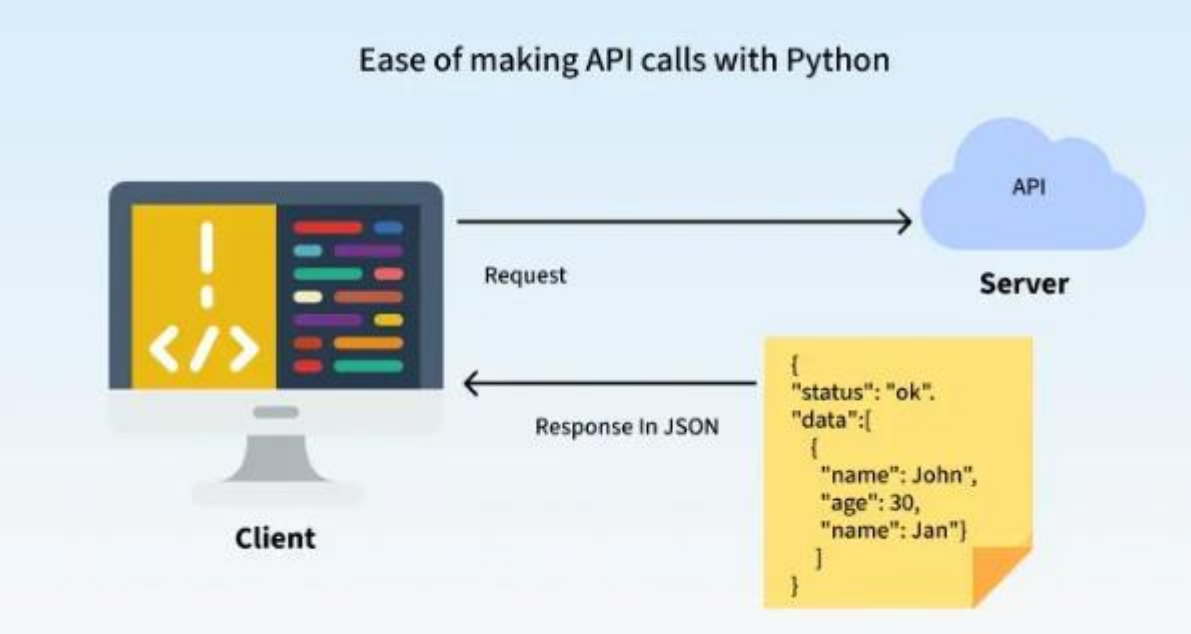
```
*data - Notepad
File  Edit  Format  View  Help
€□•□          ]”(K□K K□K□K□e.
```

3.10 INTERACTING WITH WEB APIS

Python API is used to retrieve data from various sources. Also, we will cover all concepts related to Python API from basic to advanced. Various websites provide weather data, Twitter provides data for research purposes, and stock market websites provide data for share prices.







API stands for "**Application Programming Interface.**" In simple terms, it's a set of rules and protocols that allow how different software applications can communicate and interact with each other. APIs define the methods and data formats that applications can use to request and exchange information. To retrieve data from a web server, a client application initiates a request, and the server responds with the requested data.
APIs act as bridges that enable the smooth exchange of data and functionality, enhancing interoperability across various applications.

Making API Requests in Python

In order to work with API some tools are required such as requests so we need to first install them in our system.
Command to install '**requests':**

pip install requests

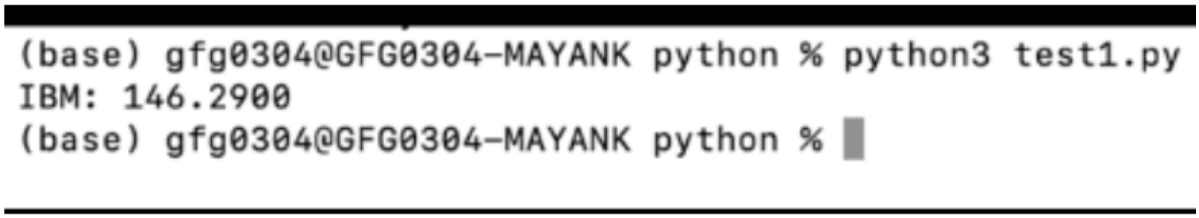
Once we have installed it, we need to import it in our code to use it.
Command to import **'requests':**

Let us understand the working of API with examples. First let us take a simple example.
Example 1: Fetching Live Stock Price Using Alpha Vantage API

This example retrieves the latest opening price for IBM stock at a 5-minute interval. Here we make use of 'requests' to make a call and it is checked with the help of status code that whether our request was successful or not.Then the response is converted to python dictionary and the respected data is stored .

```
import requests

def get_stock_data():
    url = "https://www.alphavantage.co/query?
function=TIME_SERIES_INTRADAY&symbol=IBM&interval=5min&outputsize=full&apikey=demo"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()
        last_refreshed = data["Meta Data"]["3. Last Refreshed"]
        price = data["Time Series (5min)"][last_refreshed]["1. open"]
        return price
    else:
        return None

price = get_stock_data()
symbol = "IBM"
if price is not None:
    print(f"{symbol}: {price}")
else:
    print("Failed to retrieve data.")
```

Output:



Explanation:

- Sends a GET request to Alpha Vantage API.
- Checks if the request was successful (status_code == 200).
- Parses JSON response to extract latest opening stock price.
- Prints the price or error message.

Understanding API Status Codes
Status codes tell us how the server handled our request:

- 200 OK: Request successful, data returned.
- 201 Created: New resource created.
- 204 No Content: Success but no data returned.
- 400 Bad Request: Invalid request.
- 401 Unauthorized: Missing or invalid API key.
- 500 Internal Server Error: Server encountered an error.

3.11 INTERACTING WITH DATABASES

Performing various operations on data saved in SQL might lead to performing very complex queries that are not easy to write. So to make this task easier it is often useful to do the job using pandas which are specially built for data preprocessing and is more simple and user-friendly than SQL. There might be cases when sometimes the data is stored in SQL and we want to fetch that data from SQL in python and then perform operations using pandas. So let's see how we can interact with SQL databases using pandas. This is the database we are going to work with diabetes_data.

Note: Assuming that the data is stored in sqlite3

Reading the data

```
# import the libraries
import sqlite3
import pandas as pd

# create a connection
con = sqlite3.connect('Diabetes.db')

# read data from SQL to pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# show top 5 rows
data.head()
```

Output

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Basic operation

Slicing of rows We can perform slicing operations to get the desired number of rows from within a given range. With the help of slicing, we can perform various operations only on the specific subset of the data

```
# read the data from sql to pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# slicing the number of rows
df1 = data[10:15]
df1
```

Output

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
10	4	110	92	0	0	37.6	0.191	30	0
11	10	168	74	0	0	38.0	0.537	34	1
12	10	139	80	0	0	27.1	1.441	57	0
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1

Selecting specific columns To select a particular column or to select number of columns from the dataframe for further processing of data.

```
# read the data from sql to
# pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# selecting specific columns.
df2 = data.loc[:, ['Glucose', 'BloodPressure']].head()
df2
```

Output:

	Glucose	BloodPressure
0	148	72
1	85	66
2	183	64
3	89	66
4	137	40

Summarize the data In order to get insights from data, we must have a statistical summary of data. To display a statistical summary of the data such as mean, median, mode, std etc. We perform the following operation

```
# read the data from sql
# to pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# summarize the data
data.describe()
```

Output:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Sort data with respect to a column For sorting the dataframe with respect to a given column values

```
# read the data from sql
# to pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# sort data with respect
# to particular column.
data.sort_values(by ='Age').head()
```

Output:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
255	1	113	64	35	0	33.6	0.543	21	1
60	2	84	0	0	0	0.0	0.304	21	0
102	0	125	96	0	0	22.5	0.262	21	0
182	1	0	74	20	23	27.7	0.299	21	0
623	0	94	70	27	115	43.5	0.347	21	0

Display mean of each column To Display the mean of every column of the dataframe.

```
# read the data from sql
# to pandas dataframe.
data = pd.read_sql_query('Select * from Diabetes;', con)

# count number of rows and columns
data.mean()
```

Output:

```
Pregnancies                  3.845052
Glucose                    120.894531
BloodPressure               69.105469
SkinThickness               20.536458
Insulin                     79.799479
BMI                         31.992578
DiabetesPedigreeFunction     0.471876
Age                         33.240885
Outcome                      0.348958
dtype: float64
```

3.12 DATA CLEANING AND PREPARATION

This includes handling missing data, correcting data types, removing duplicates, etc.

HANDLING MISSING DATA

Missing data can be detected and handled using isnull(), dropna(), fillna(), etc.

```
data = {
    'Name': ['John', 'Anna', 'Tom', None],
    'Score': [95, None, 88, 70]
}
df = pd.DataFrame(data)

# Check missing values
print(df.isnull())

# Drop rows with any missing values
df_dropped = df.dropna()

# Fill missing values
df_filled = df.fillna({'Name': 'Unknown', 'Score': df['Score'].mean()})

print(df_filled)
```

DATA TRANSFORMATION

Includes operations like renaming columns, changing data types, and applying functions to columns.

```
df = pd.DataFrame({
    'Name': ['John', 'Anna', 'Tom'],
    'Marks': ['90', '85', '78']
})

# Convert column to numeric
df['Marks'] = pd.to_numeric(df['Marks'])

# Add a new column with grade
df['Grade'] = df['Marks'].apply(lambda x: 'A' if x >= 85 else 'B')

print(df)
```

STRING MANIPULATION IN PANDAS

```
df = pd.DataFrame({
    'Name': ['john doe', 'ANNA SMITH', 'Tom Hanks']
})

# Capitalize names properly
df['Name_Clean'] = df['Name'].str.title()

# Extract first name
df['First_Name'] = df['Name'].str.split().str[0]

# Check if name contains 'Smith'
df['Has_Smith'] = df['Name'].str.contains('Smith', case=False)

print(df)
```

Unit - IV

Data Visualization Tools in Python- Introduction to Matplotlib, Basic plots Using matplotlib, Specialized Visualization Tools using Matplotlib, Advanced Visualization Tools using Matplotlib- Waffle Charts, Word Clouds.

Data visualization is the graphical representation of information and data. In Python, a wide range of libraries and tools are available to create both basic plots and advanced interactive visualizations. These tools help explore, analyze, and communicate data insights effectively.

Why Use Data Visualization Tools?

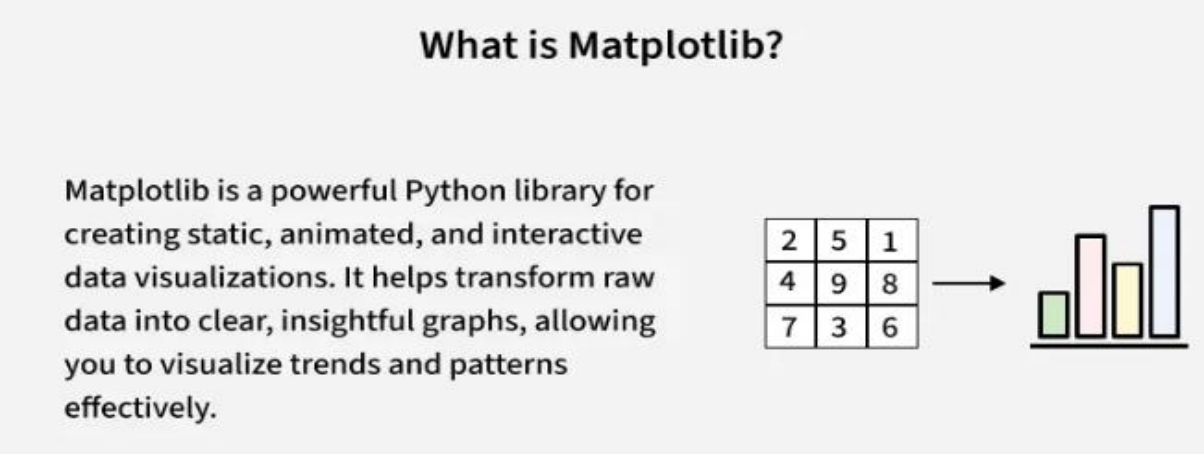
- To **explore** trends, patterns, and outliers in large datasets.
- To **summarize** complex data in a visually appealing way.
- To **communicate** results clearly to a non-technical audience.
- To support **decision-making** through intuitive graphics.

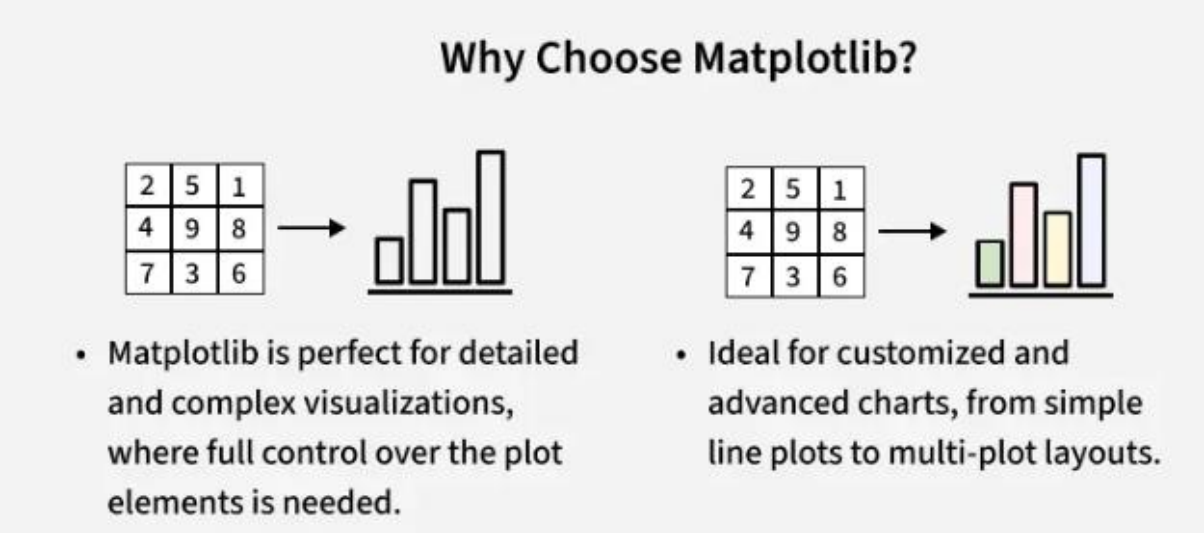
Data Visualization Tools in Python

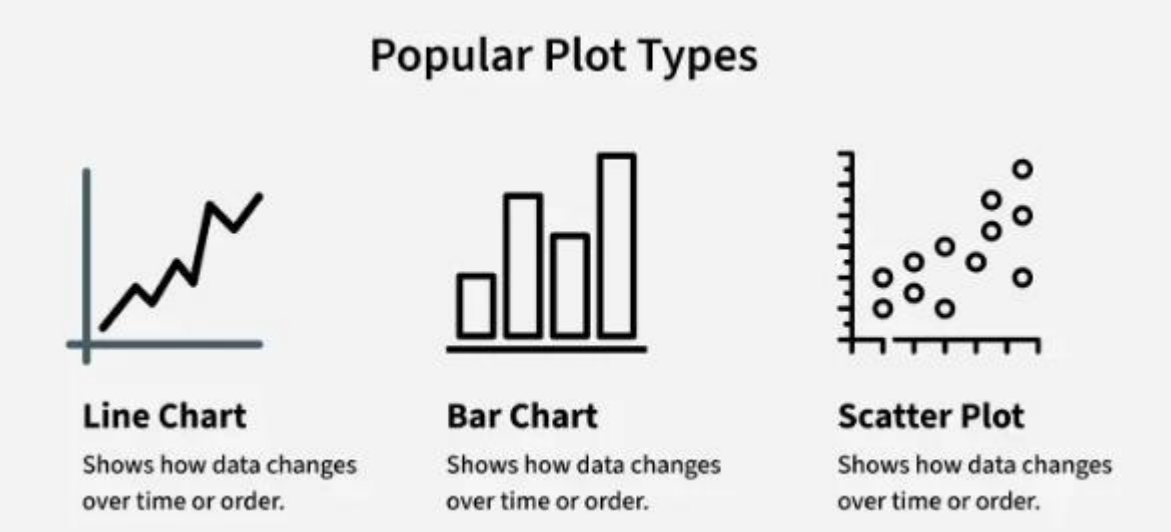
Tool / Library	Description	Key Features	Suitable For
Matplotlib	Foundation library for 2D plots in Python	Line, bar, scatter, pie, histogram, fully customizable	General plotting, publication-quality
Seaborn	High-level API based on Matplotlib	Built-in themes, statistical plots (box, violin, heatmaps)	Statistical visualization
Plotly	Interactive web-based visualizations	Hover tooltips, zoom, export to HTML	Dashboards, web apps
Geopandas	Geospatial plotting based on Matplotlib	Plot maps, shapefiles, geographic data	GIS, spatial analysis
WordCloud	Creates word clouds from text data	Frequency-based text visualization	NLP, textual data

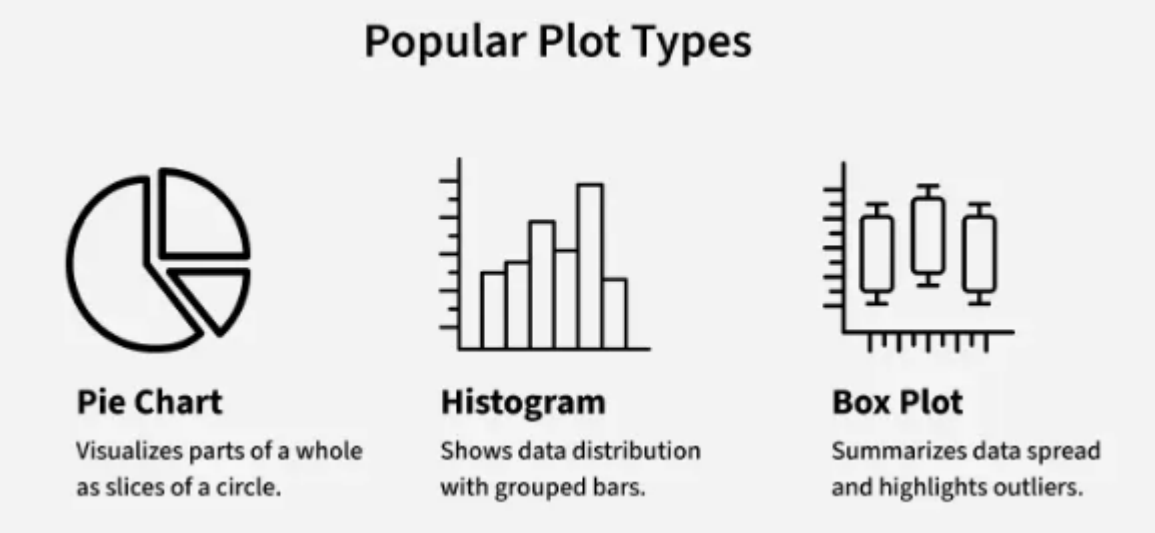
4.1 INTRODUCTION TO MATPLOTLIB

Matplotlib is a widely-used Python library used for creating static, animated and interactive data visualizations. It is built on the top of **NumPy** and it can easily handles large datasets for creating various types of plots such as line charts, bar charts, scatter plots, etc. These visualizations help us to understand data better by presenting it clearly through graphs and charts. In this article, we will see how to create different types of plots and customize them in matplotlib.









Installing Matplotlib for Data Visualization

To install Matplotlib, we use the pip command.

To install Matplotlib type below command in the terminal:

pip install matplotlib

If we are working on a Jupyter Notebook, we can install Matplotlib directly inside a notebook cell by running:

!pip install matplotlib

Visualizing Data with Pyplot using Matplotlib

Matplotlib provides a module called pyplot which offers a **MATLAB-like interface** for creating plots and charts. It simplifies the process of generating various types of visualizations by providing a collection of functions that handle common plotting tasks. Let's explore some examples with simple code to understand how to use it effectively.

4.2 BASIC PLOTS USING MATPLOTLIB

1. Line Chart

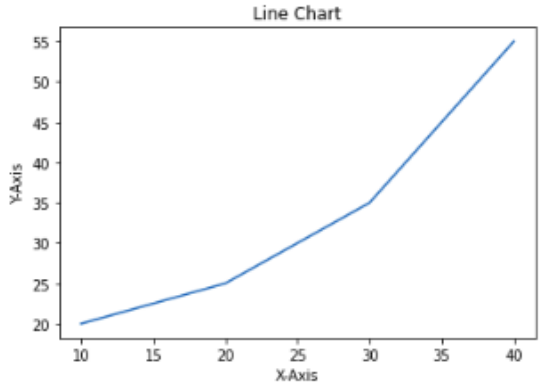
Line chart is one of the basic plots and can be created using the plot() function. It is used to represent a relationship between two data X and Y on a different axis.

Syntax:

matplotlib.pyplot.plot(x, y, color=None, linestyle='-', marker=None, linewidth=None, markersize=None)

```
import matplotlib.pyplot as plt

x = [10, 20, 30, 40]
y = [20, 25, 35, 55]

plt.plot(x, y)

plt.title("Line Chart")

plt.ylabel('Y-Axis')

plt.xlabel('X-Axis')
plt.show()
```

Output:



2. Bar Chart

A bar chart is a graph that represents the category of data with rectangular bars with lengths and heights which is proportional to the values which they represent. The bar plot can be plotted horizontally or vertically. It describes the comparisons between different categories and can be created using the ***bar()*** *method.*

In the below example we will using Pandas library for its implementation on tips dataset. It is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s and it contains 6 columns.

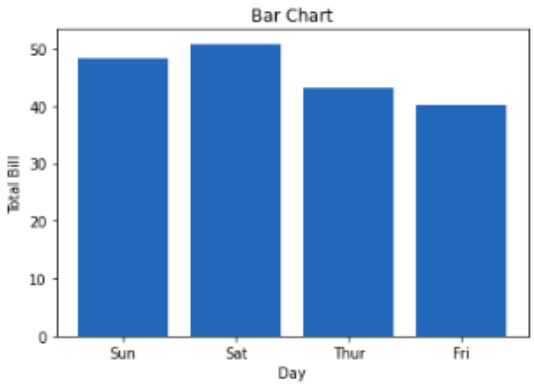
Syntax:

matplotlib.pyplot.bar(x, height, width=0.8, bottom=None, color=None, edgecolor=None, linewidth=None)

Example:

```
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('/content/tip.csv')

x = data['day']
y = data['total_bill']

plt.bar(x, y)

plt.title("Bar chart")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()
```

Output:



3. Histogram

*A **histogram** is used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The **hist()** function is used to find and create histogram of x.*

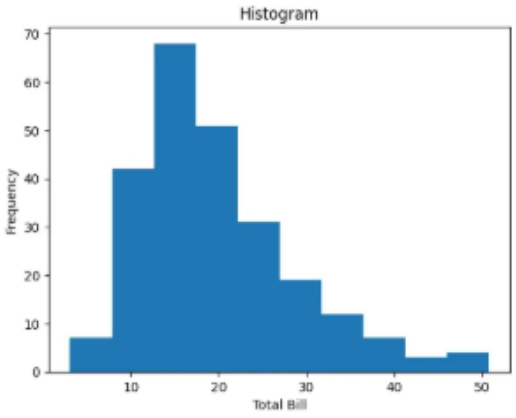
Syntax:

matplotlib.pyplot.hist(x, bins=None, range=None, density=False, color=None, edgecolor=None, alpha=None)

Example:

```
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('/content/tip.csv')

x = data['total_bill']

plt.hist(x)

plt.title("Histogram")

plt.ylabel('Frequency')

plt.xlabel('Total Bill')

plt.show()
```

Output:



Histogram

4. Scatter Plot

Scatter plots *are used to observe relationships between variables. The* ***scatter()*** *method in the matplotlib library is used to draw a scatter plot.*

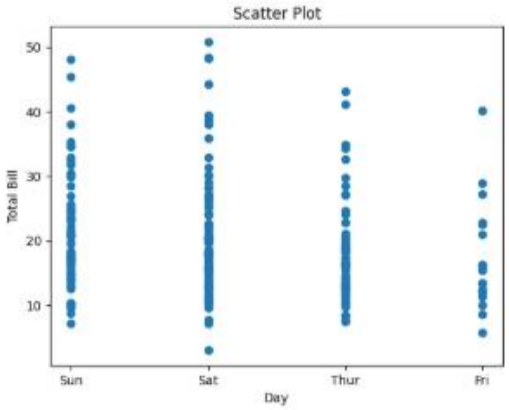
Syntax:

matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, linewidths=None, edgecolors=None, alpha=None)

Example:

```
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('/content/tip.csv')

x = data['day']
y = data['total_bill']

plt.scatter(x, y)

plt.title("Scatter Plot")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()
```

Output:



Scatter Plot

5. Pie Chart

Pie chart is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called ***wedges****. It can be created using the* ***pie()*** *method.*

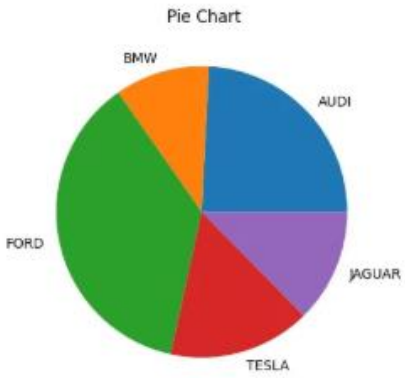
Syntax:

matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)

Example:

```
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('/content/tip.csv')

cars = ['AUDI', 'BMW', 'FORD',
        'TESLA', 'JAGUAR',]
data = [23, 10, 35, 15, 12]

plt.pie(data, labels=cars)

plt.title(" Pie Chart")

plt.show()
```

Output:



Pie Chart

6. Box Plot

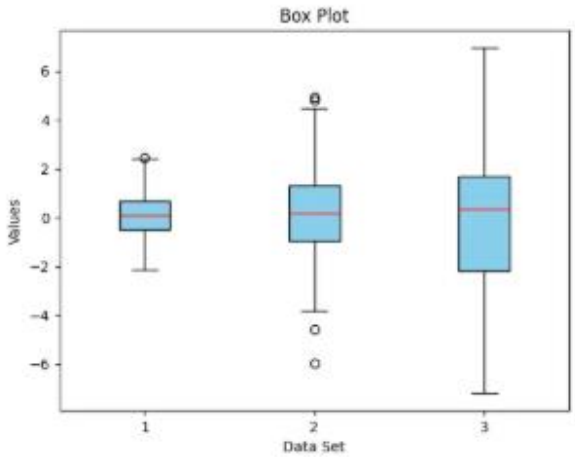
*A Box Plot is also known as a **Whisker Plot** which is a standardized way of displaying the distribution of data based on a five-number summary: **minimum, first quartile (Q1), median (Q2), third quartile (Q3) and maximum**. It can also show outliers.*

Syntax:

matplotlib.pyplot.boxplot(x, notch=False, vert=True, patch_artist=False, showmeans=False, showcaps=True, showbox=True)

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(10)
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

plt.boxplot(data, vert=True, patch_artist=True,
            boxprops=dict(facecolor='skyblue'),
            medianprops=dict(color='red'))

plt.xlabel('Data Set')
plt.ylabel('Values')
plt.title('Box Plot')
plt.show()
```

Output:



Box Plot

*The box shows the **interquartile range (IQR)** the line inside the box shows the median and the "**whiskers**" extend to the minimum and maximum values within **1.5 * IQR** from the first and third quartiles. Any points outside this range are considered outliers and are plotted as individual points.*

7. Heatmap

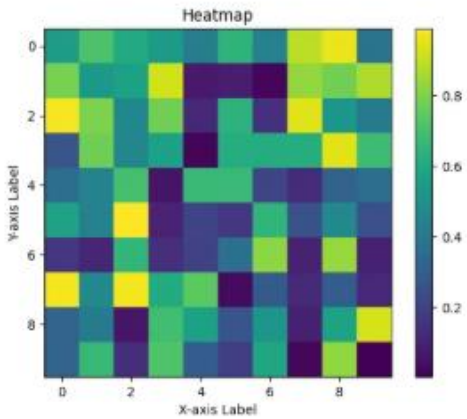
A Heatmap represents data in a matrix form where individual values are represented as colors. They are useful for visualizing the magnitude of multiple features in a two-dimensional surface and identifying patterns, correlations and concentrations.

Syntax:

matplotlib.pyplot.imshow(X, cmap=None, interpolation=None, aspect=None)

```python
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(0)
data = np.random.rand(10, 10)

plt.imshow(data, cmap='viridis', interpolation='nearest')

plt.colorbar()

plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Heatmap')

plt.show()
```

Output:



Heatmap

The color bar on the side provides a scale to interpret the colors, darker colors representing lower values and lighter colors representing higher values. This type of plot is used in fields like data analysis, bioinformatics and finance to visualize data correlations and distributions across a matrix.

4.3 SPECIALIZED VISUALIZATION TOOLS USING MATPLOTLIB

Specialized visualization tools in Matplotlib refer to advanced or non-standard plotting techniques that go **beyond basic line, bar, or scatter plots**. These tools are especially useful for domain-specific analysis, high-dimensional data, or customized presentation needs.

They allow you to create visualizations such as heatmaps, radar charts, 3D plots, polar plots, and more using Matplotlib's flexible plotting architecture.

Key Features of Specialized Visualization Tools in Matplotlib

1. **Customization**
   You can control every element of a plot—colors, axes, annotations, markers, and more.
2. **High-Dimensional Data Support**
   Create 3D plots, surface plots, and multivariate charts like radar plots.

3. **Domain-Specific Visuals**
   Useful in fields like finance, biology, engineering (e.g., polar plots for angles, heatmaps for genomics).
4. **Integration**
   Works seamlessly with NumPy, Pandas, and other Python libraries.

Examples of Specialized Visualizations

Chart Type	Description
Heatmap	Shows data values as color in a matrix layout.
Word Cloud	Visualizes word frequency.
Stacked Area Chart	Shows cumulative trends over time.
3D Surface Plot	Visualizes 3D surface data.
Stacked Area	Shows cumulative trends over time.
Quiver Plot	Quiver plots are used to display vector fields. Each arrow shows both direction and magnitude of the vector at a point. They are especially useful in physics, fluid dynamics, and wind flow analysis.

1.Heatmap

Shows data values as color in a matrix layout.

```
import matplotlib.pyplot as plt
import numpy as np

data = np.random.rand(6, 6)
plt.imshow(data, cmap='viridis')
plt.colorbar()
plt.title("Heatmap")
plt.show()
```

2. Word Count

Visualizes word frequency.

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

text = "machine learning data science artificial intelligence python visualization"
wordcloud = WordCloud(background_color='white').generate(text)

plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud")
plt.show()
```

3. Stacked Area Chart

Shows cumulative trends over time.

```
x = range(5)
y1 = [1, 2, 3, 4, 5]
y2 = [2, 1, 2, 1, 2]
y3 = [1, 3, 1, 3, 1]

plt.stackplot(x, y1, y2, y3, labels=['A', 'B', 'C'])
plt.legend(loc='upper left')
plt.title("Stacked Area Chart")
plt.show()
```

4. 3D Surface Plot

Visualizes 3D surface data.

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

X = Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z = np.sin(np.sqrt(X**2 + Y**2))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
plt.title("3D Surface Plot")
plt.show()
```

5. Quiver Plot

Quiver plots are used to display vector fields. Each arrow shows both direction and magnitude of the vector at a point. They are especially useful in **physics**, **fluid dynamics**, and **wind flow analysis**.

```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.meshgrid(np.arange(-5, 5, 1), np.arange(-5, 5, 1))
U = -Y  # horizontal component (e.g., wind direction)
V = X   # vertical component (e.g., wind speed)

plt.quiver(X, Y, U, V, color='blue')
plt.title("Quiver Plot - Vector Field")
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)
plt.axis('equal')
plt.show()
```

4.4 ADVANCED VISUALIZATION TOOLS USING MATPLOTLIB- WAFFLE CHARTS, WORD CLOUDS.

A **waffle chart** is a grid-based visualization that shows **proportional data**. Each square in the grid represents a **percentage** or **count** of the total.

Library:

pywaffle (extends matplotlib)

```
# Install: pip install pywaffle
from pywaffle import Waffle
import matplotlib.pyplot as plt

data = {'Category A': 40, 'Category B': 30, 'Category C': 20, 'Category D': 10}

fig = plt.figure(
    FigureClass=Waffle,
    rows=5,
    values=data,
    colors=["#4CAF50", "#2196F3", "#FFC107", "#F44336"],
    title={'label': 'Waffle Chart Example', 'loc': 'center'},
    legend={'loc': 'upper left', 'bbox_to_anchor': (1.1, 1)}
)
plt.show()
```

Use Case:

Comparing categories visually like pie charts, but in a **more structured format**.

2. **Word Cloud**

A **word cloud** is a visual representation of **text data** where the **size** of each word indicates its **frequency** or **importance**.

Library:

Wordcloud

Displayed using matplotlib

```python
from wordcloud import WordCloud
import matplotlib.pyplot as plt

text = "machine learning data science visualization python statistics pandas numpy"

wordcloud = WordCloud(
    width=600,
    height=400,
    background_color='white',
    colormap='tab10'
).generate(text)

plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud Example")
plt.show()
```

Use Case:

Text mining, NLP, feedback analysis, keyword visualization.

📜 Summary Table

Visualization	Tool Used	Purpose	Matplotlib Support
Waffle Chart	`pywaffle`	Show proportion using squares	✅ Yes
Word Cloud	`wordcloud`	Display frequent words visually	✅ Yes

Unit - V

Introduction: Data Analysis, Excel Data analysis. Working with range names. Tables. Cleaning Data. Conditional formatting, Sorting, Advanced Filtering, Lookup functions, Pivot tables, Data Visualization, Data Validation. Understanding Analysis tool pack: Anova, correlation, covariance, moving average, descriptive statistics, exponential smoothing, fourier Analysis, Random number generation, sampling, ttest, f-test, and regression.

5.1 DATA ANALYSIS

Data analysis is the process of cleaning, changing, and processing raw data and extracting actionable, relevant information that helps businesses make informed decisions.

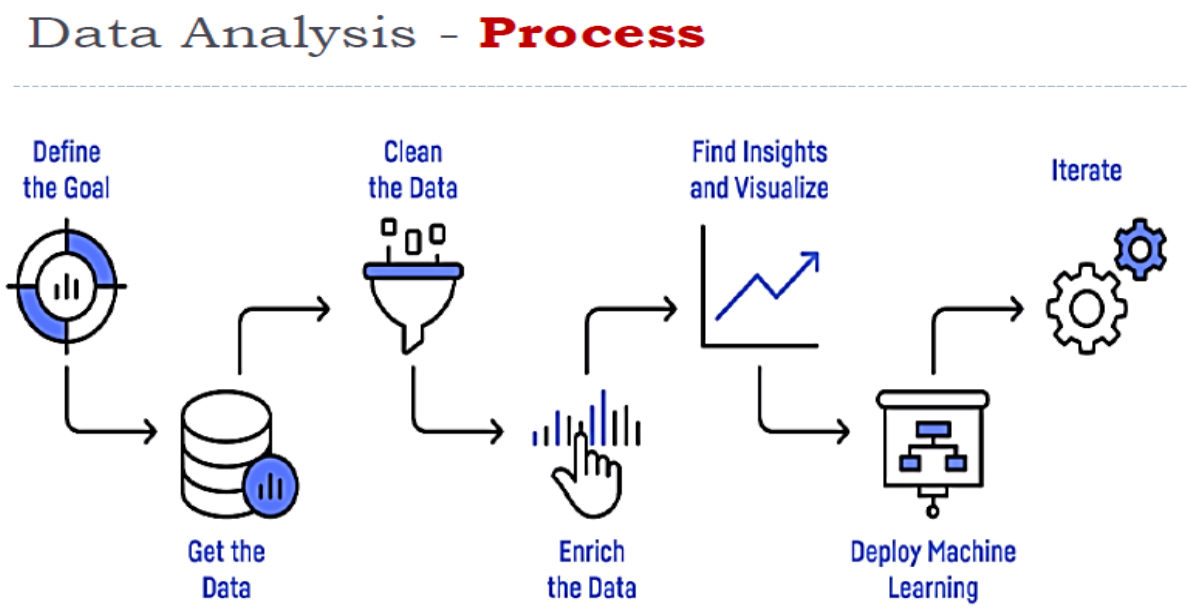
The procedure helps reduce the risks inherent in decision-making by providing useful insights and statistics, often presented in charts, images, tables, and graphs.

Data Analysis is the process of systematically applying statistical and/or logical techniques to describe and illustrate, condense and recap, and evaluate data.

An essential component of ensuring data integrity is the accurate and appropriate analysis of research findings. Data analysis plays a crucial role in processing big data into useful information.

Why is Data Analysis **Important**?

Better Customer Targeting
You Will Know Your Target Customers Better
Reduce Operational Costs
Better Problem-Solving Methods.
You Get More Accurate Data.



202

Data Analysis **Process.**

Data Requirement Gathering
Data Collection
Data Cleaning
Data Analysis
Data Interpretation
Data Visualization

Data Analysis **Process.**

Considerations/issues in data analysis

There are a number of issues that researchers should be cognizant of with respect to data analysis.

These include:

- Having the necessary skills to analyze
- Concurrently selecting data collection methods and appropriate analysis
- Drawing unbiased inference
- Inappropriate subgroup analysis
- Following acceptable norms for disciplines
- Determining **statistical significance**
- Lack of clearly defined and objective **outcome measurements**

5.2 EXCEL DATA ANALYSIS

Excel is one of the most powerful tools for **data analysis**, allowing you to process, manipulate, and visualize large datasets efficiently. Whether you're analyzing sales figures, financial reports, or any other type of data, knowing **how to perform data analysis in Excel** can help you make informed decisions quickly. In this guide, we will walk you through the essential steps to **analyze data in Excel**, including using built-in tools like pivot tables, charts, and formulas to extract meaningful insights. By the end of this article, you'll have a solid understanding of how to use Excel for data analysis and improve your decision-making process.

Preparing Data for Analysis

Data cleaning becomes an intuitive process with Excel's capabilities, allowing users to identify and rectify issues like missing values and duplicates. PivotTables, a hallmark feature, empower users to swiftly summarize and explore large datasets, providing dynamic insights through customizable cross-tabulations, making Data Analysis Excel an essential skill for professionals.

Preparing Your Dataset

Before getting into analysis, it's essential to clean and organize your dataset to ensure accuracy.
How to Clean Data in Excel

1. **Remove Duplicates:** Use **Data > Remove Duplicates** to eliminate redundancy.
2. **Use TRIM and CLEAN Functions:**
    - TRIM removes unnecessary spaces.
    - CLEAN removes non-printable characters.

3. Sort and Structure Data: Convert your dataset into an Excel Table (Insert > Table) for better organization.

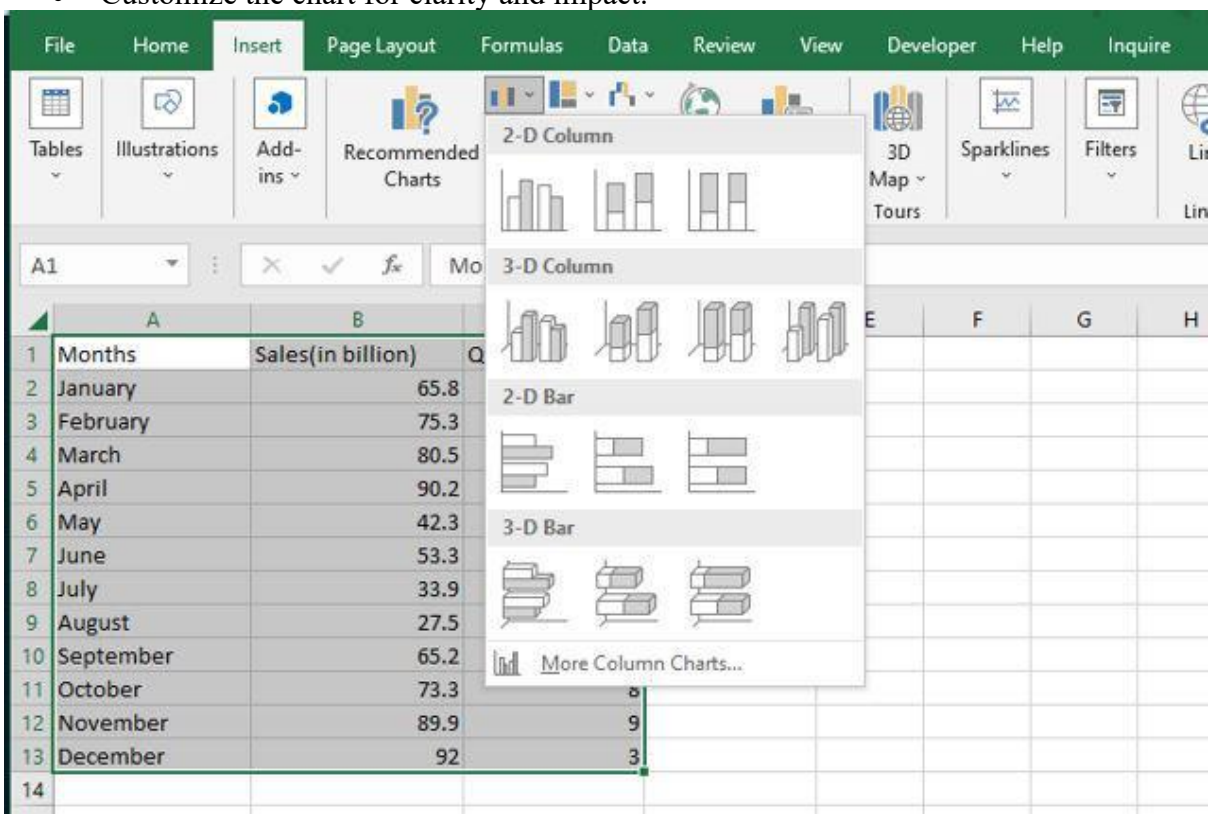
Basic Methods of Data Analysis in Excel

Excel offers several methods to analyze data effectively. Here are some key techniques:

1. Charts and Visualization

Any set of information may be graphically represented in a chart. A chart is a graphic representation of data that employs symbols to represent the data, such as bars in a bar chart or lines in a line chart. Data Analysis Excel offers several different chart types available for you to choose from, or you can use the Excel Recommended Charts option to look at charts specifically made for your data and choose one of those.
Charts make it easier to identify trends and relationships in your data:

- Select your dataset and go to **Insert > Charts**.
- Choose from bar charts, line charts, or pie charts.
- Customize the chart for clarity and impact.



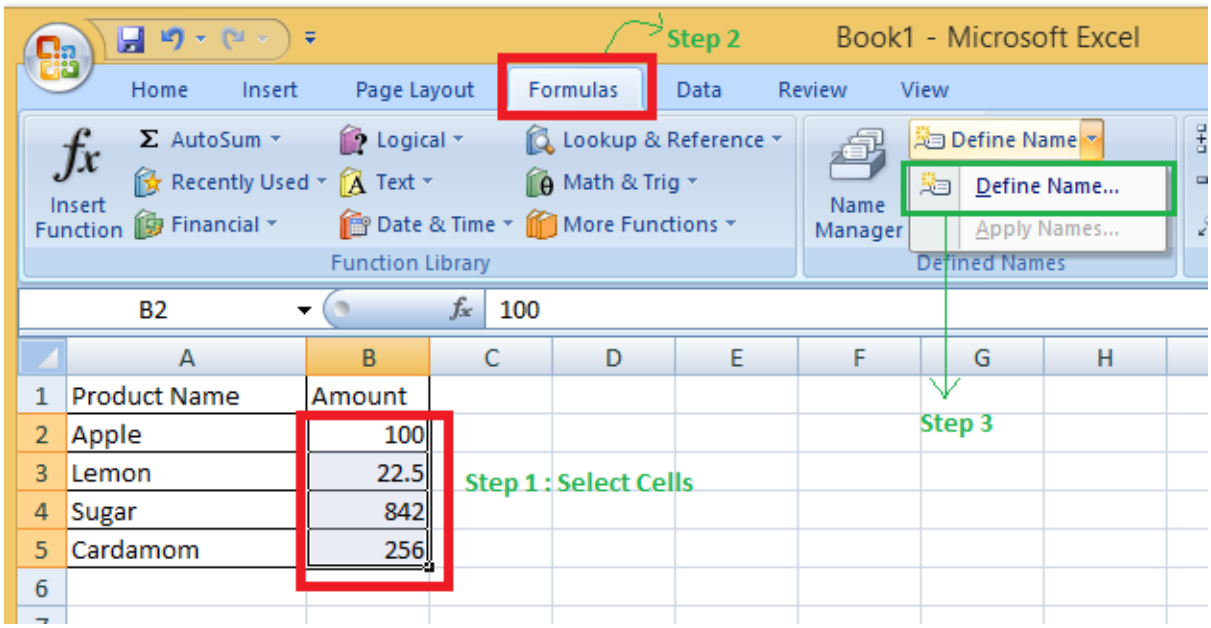
5.3 WORKING WITH RANGE NAMES

We can use the name for the cell Ranges instead of the cell reference (such as A1 or A1:A10). We can create a named range for a range of cells and use then use that name directly in the Excel formulas. When we have huge data sets, Excel-named ranges make it easy to refer (by directly using a name to that data set).

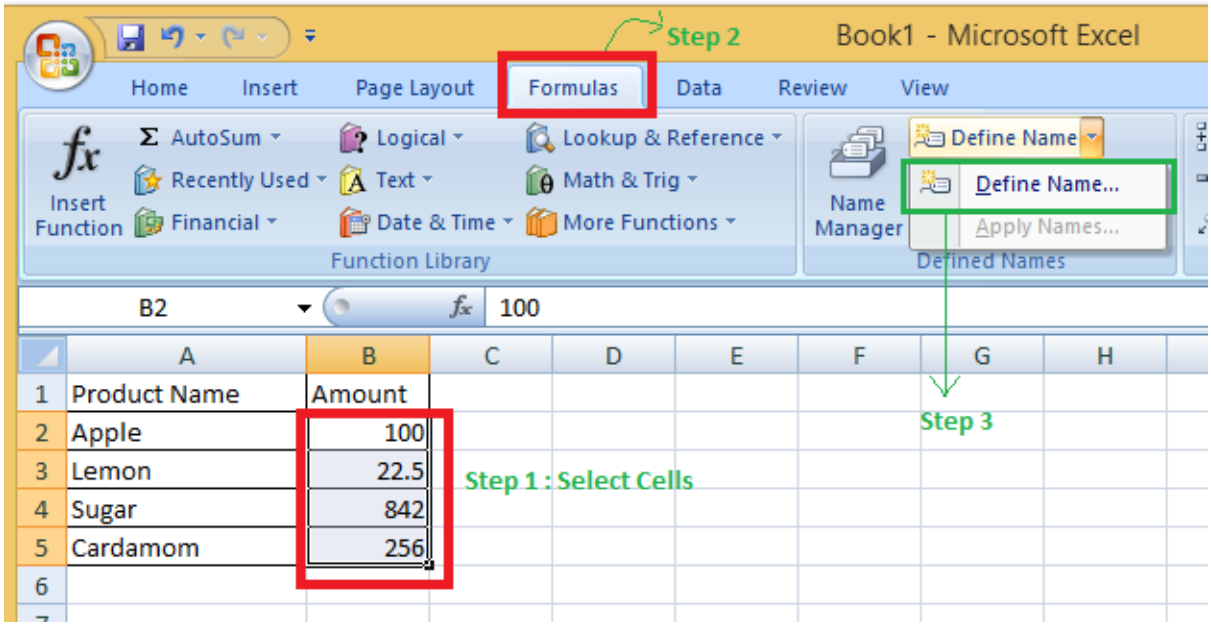
Creating an Excel Named Range :

There can be 3 ways to create named ranges in Excel :
Method 1: Using Define Name
Use the following steps to create named range using Define Name :

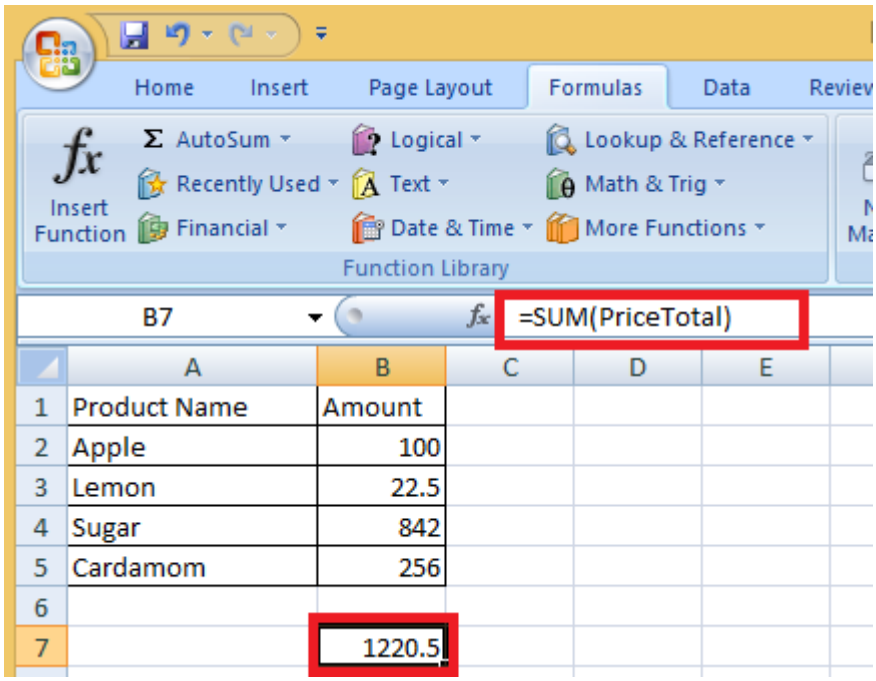
- Select the range B1:B5.
- Click on the Formulas tab.
- Then click on Define Name.



- Give a new Name(PriceTotal in our example) & click Ok. (You can see the range in the bottom refers to section, here absolute referencing is used, $ before the row number/ column letter locks the row/column).



- Now, the next thing is to see that how to use this named range in any of the Excel formulas. For example, if you want to get the sum of all numbers in the above name range then, can say simply write**: =SUM(PriceTota**l).



5.4 TABLES

You can create as many tables as you want in a spreadsheet.

To quickly create a table in Excel, do the following:

1. Select the cell or the range in the data.
2. Select **Home** > **Format as Table**.
3. Pick a table style.
4. In the **Format as Table** dialog box, select the checkbox next to **My table as headers** if you want the first row of the range to be the header row, and then click **OK**.

5.5 EXCEL DATA CLEANING TECHNIQUES

In today's data-driven world, having clean and reliable data is crucial for making informed business decisions. In this article, you will learn different techniques to clean data in Excel that will transform your excel spreadsheets from chaotic to organized. Whether you're a beginner or an experienced user, mastering these techniques can significantly enhance your data analysis skills.

We'll explore powerful tools such as Power Query to automate data cleaning tasks, conditional formatting to highlight inconsistencies, and other essential data cleaning tools that can streamline your workflow. Say goodbye to messy data and hello to accurate insights as we dive into the world of Excel data cleaning!

Excel provides some indispensable Data-Cleaning techniques to do data cleaning easily. The most widely used techniques are :

1. Remove Duplicates

Duplicate entries can sneak into your data when copying and pasting from various sources. Excel simplifies the process of removing duplicates, saving you time and effort. Excel has a built-in function to remove duplicates, which can save you a lot of time and effort. To do this, follow these steps:

Step 1: Select the data range.

Step 2: Go to the **Data** tab.

Step 3: Click on **Remove Duplicates**.

Step 4: Choose the relevant columns and hit **OK**.

2. Standardize Formats

Inconsistent formatting can hinder data analysis. To standardize formats (such as currency, dates, and times), use Excel’s formatting tools. Here’s how:

Step 1: Select the data range.

Step 2: Right-click and choose **Format Cells**.

Step 3: Adjust the format settings as needed.

3. Clean Text Data

Text data often harbors errors like typos, extra spaces, and inconsistent capitalization. Excel offers handy functions for cleaning text data:

- **TRIM**: Removes leading and trailing spaces.
- **CLEAN**: Eliminates non-printable characters.

- **PROPER**: Capitalizes the first letter of each word.

4. Fill Missing Values

Missing values can plague your data. Excel's data analysis tools come to the rescue:

- Calculate the average or median of surrounding data.
- Fill in missing values accordingly.

5. Data Validation

Data validation can help to prevent errors from being entered into your data in the first place. You can use data validation to specify the type of data that can be entered into a cell, as well as the range of valid values.

6. Conditional Formatting

Highlight errors or anomalies in your data using conditional formatting. For instance, you can:

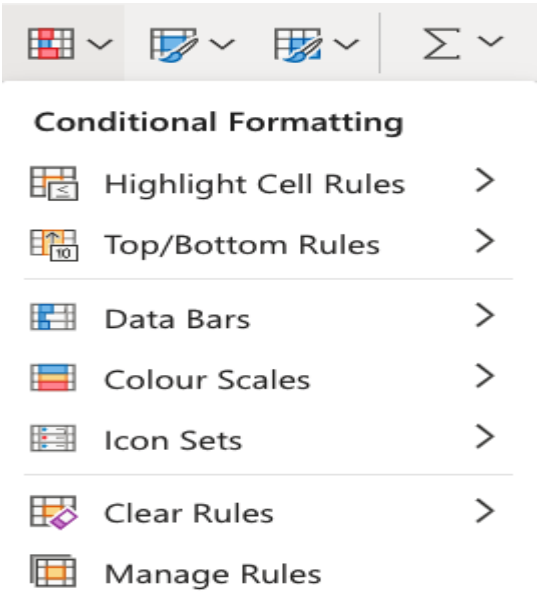
- Highlight blank cells.
- Identify invalid characters.

5.6 CONDITIONAL FORMATTING

Conditional formatting is used to change the appearance of cells in a range based on your specified **conditions**.

The conditions are rules based on specified numerical values or matching text.

The browser version of Excel provides a number of built-in conditions and appearances:



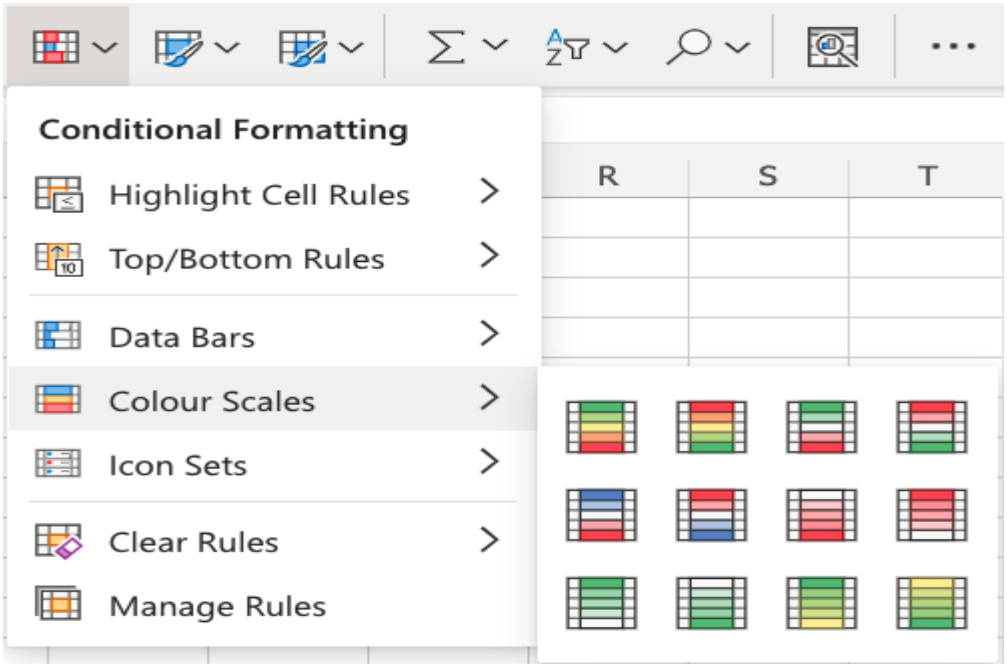
Conditional Formatting Example

Conditional formatting, step by step:

1. Select the range of Speed values C2:C9

	A	B	C	D
1	Name	Type 1	Speed	
2	Bulbasaur	Grass	45	
3	Ivysaur	Grass	60	
4	Venusaur	Grass	80	
5	Charmander	Fire	65	
6	Charmeleon	Fire	80	
7	Charizard	Fire	100	
8	Squirtle	Water	43	
9	Wartortle	Water	58	
10				

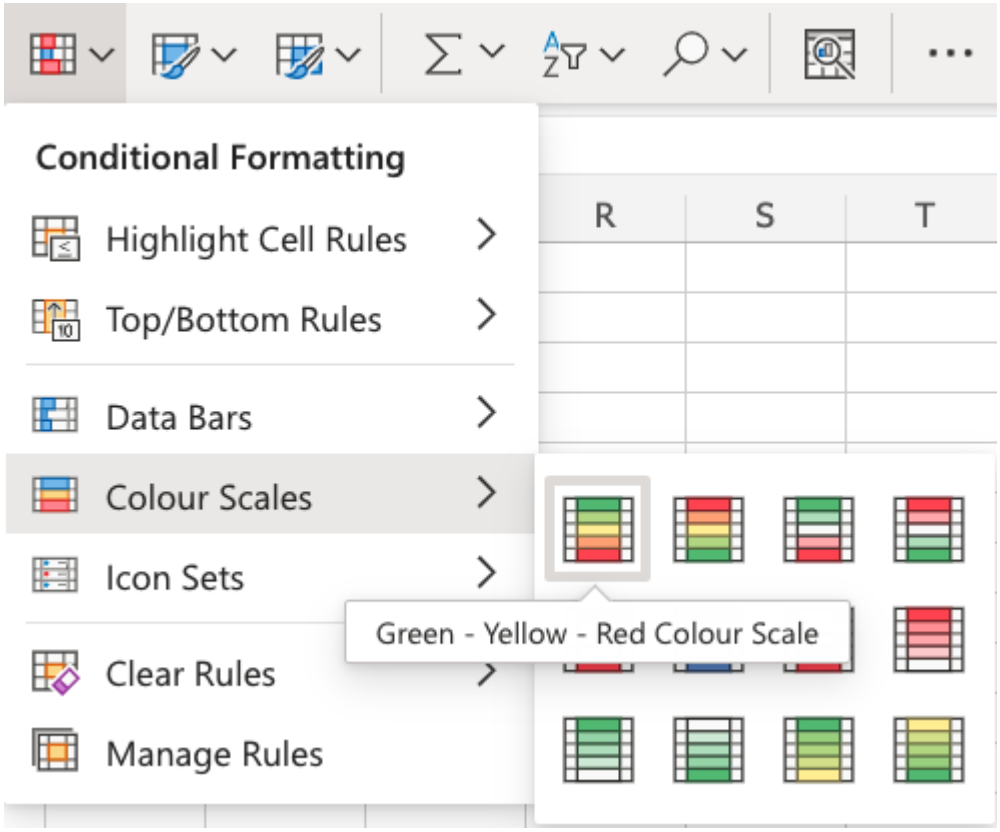
2. Click on the Conditional Formatting icon  in the ribbon, from the **Home** menu
3. Select **Color Scales** from the drop-down menu



There are 12 Color Scale options with different color variations.

The color on the top of the icon  will apply to the highest values.

4. Click on the "Green - Yellow - Red Colour Scale" icon



Now, the Speed value cells will have a colored background highlighting:

	A	B	C	D
1	Name	Type 1	Speed	
2	Bulbasaur	Grass	45	
3	Ivysaur	Grass	60	
4	Venusaur	Grass	80	
5	Charmander	Fire	65	
6	Charmeleon	Fire	80	
7	Charizard	Fire	100	
8	Squirtle	Water	43	
9	Wartortle	Water	58	
10				

Dark green is used for the highest values, and dark red for the lowest values.

Charizard has the highest Speed value (100) and Squirtle has the lowest Speed value (43).

All the cells in the range gradually change color from green, yellow, orange, then red.

EXCEL SORTING

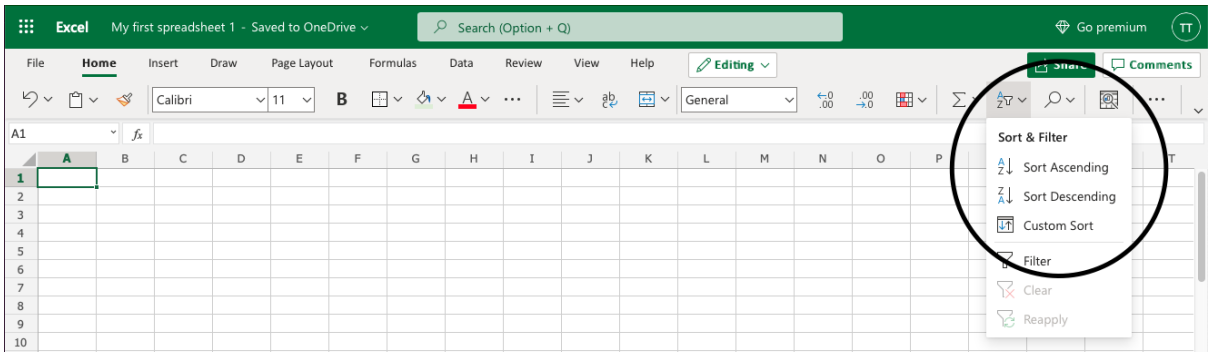
Ranges can be sorted using the **Sort Ascending** and **Sort Descending** commands.

Sort Ascending: from smallest to largest.

Sort Descending: from largest to smallest.

The sort commands work for text too, using A-Z order.

The commands are found in the Ribbon under the **Sort & Filter** menu ()



Example Sort (text)

Sort the Pokemons in the range **A2:A21** by their **Name**, ascending from smallest to largest (A-Z).

1. Select **A2:A21**
2. Open the **Sort & Filter** menu
3. Click **Sort Ascending**

Note: A1 is not included as it is the header for the column. This is the row that is dedicated to the filter. Including it will blend it with the rest.

	A	B	C
1	Name		
2	Bulbasaur		
3	Ivysaur		
4	Venusaur		
5	Charmander		
6	Charmeleon		
7	Charizard		
8	Squirtle		
9	Wartortle		
10	Blastoise		
11	Caterpie		
12	Metapod		
13	Butterfree		
14	Weedle		
15	Kakuna		
16	Beedrill		
17	Pidgey		
18	Pidgeotto		
19	Pidgeot		
20	Rattata		
21	Raticate		

ADVANCED FILTER IN EXCEL

This example teaches you how to apply an advanced filter in Excel to only display records that meet complex criteria.

When you use the Advanced Filter, you need to enter the criteria on the worksheet. Create a Criteria range (blue borders in the image below for illustration only) above your data set. Use the same column headers. Be sure there's at least one blank row between your Criteria range and data set.

And Criteria

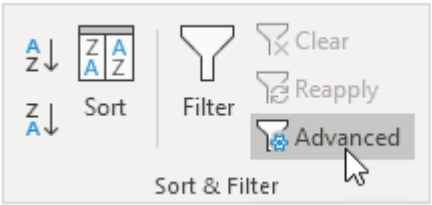
To display the sales in the USA and in Qtr 4, execute the following steps.

1. Enter the criteria shown below on the worksheet.

	A	B	C	D	E
1	**Last Name**	**Sales**	**Country**	**Quarter**	
2			USA	Qtr 4	
3					
4					
5	**Last Name**	**Sales**	**Country**	**Quarter**	
6	Smith	$16,753.00	UK	Qtr 3	
7	Johnson	$14,808.00	USA	Qtr 4	
8	Williams	$10,644.00	UK	Qtr 2	
9	Jones	$1,390.00	USA	Qtr 3	
10	Brown	$4,865.00	USA	Qtr 4	
11	Williams	$12,438.00	UK	Qtr 1	
12	Johnson	$9,339.00	UK	Qtr 2	
13	Smith	$18,919.00	USA	Qtr 3	
14	Jones	$9,213.00	USA	Qtr 4	
15	Jones	$7,433.00	UK	Qtr 1	
16	Brown	$3,255.00	USA	Qtr 2	
17	Williams	$14,867.00	USA	Qtr 3	
18	Williams	$19,302.00	UK	Qtr 4	
19	Smith	$9,698.00	USA	Qtr 1	
20					

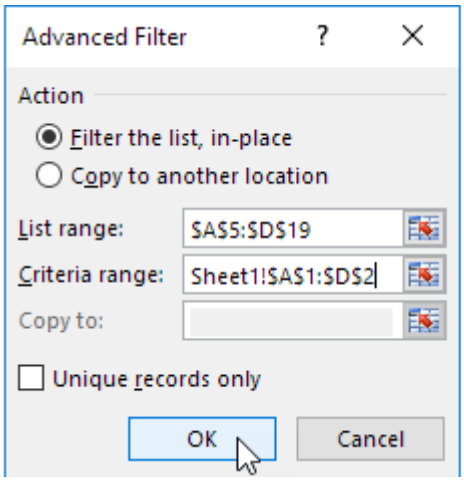
2. Click any single cell inside the data set.

3. On the Data tab, in the Sort & Filter group, click Advanced.



4. Click in the Criteria range box and select the range A1:D2 (blue).

5. Click OK.



Notice the options to copy your filtered data set to another location and display unique records only (if your data set contains duplicates).

Result:

	A	B	C	D	E
1	**Last Name**	**Sales**	**Country**	**Quarter**	
2			USA	Qtr 4	
3					
4					
5	**Last Name**	**Sales**	**Country**	**Quarter**	
7	Johnson	$14,808.00	USA	Qtr 4	
10	Brown	$4,865.00	USA	Qtr 4	
14	Jones	$9,213.00	USA	Qtr 4	
20					

No rocket science so far. We can achieve the same result with the normal filter. We need the Advanced Filter for Or criteria.

LOOKUP FUNCTIONS

Excel's LOOKUP functions look for and get data from tables according to particular criteria. They are crucial instruments for data analysis. Furthermore, they make it possible for users to locate and get data from huge databases quickly.

Top LOOKUP Functions in Excel

CustomerID	CustomerName	City	Country
C001	John Doe	Los Angeles	USA
C002	Aarav Patel	Mumbai	India
C003	Michael Lee	Sydney	Australia
C004	Ayşe Yılmaz	Istanbul	Turkey
C005	David Brown	London	Britain
C006	Marie Dubois	Paris	France
C007	Dinesh Perera	Colombo	Sri Lanka
C008	Rina Wijaya	Jakarta	Indonesia
C009	Carlos Rodriguez	Mexico City	Mexico
C010	Kwame Mensah	Accra	Ghana

1. LOOKUP Function

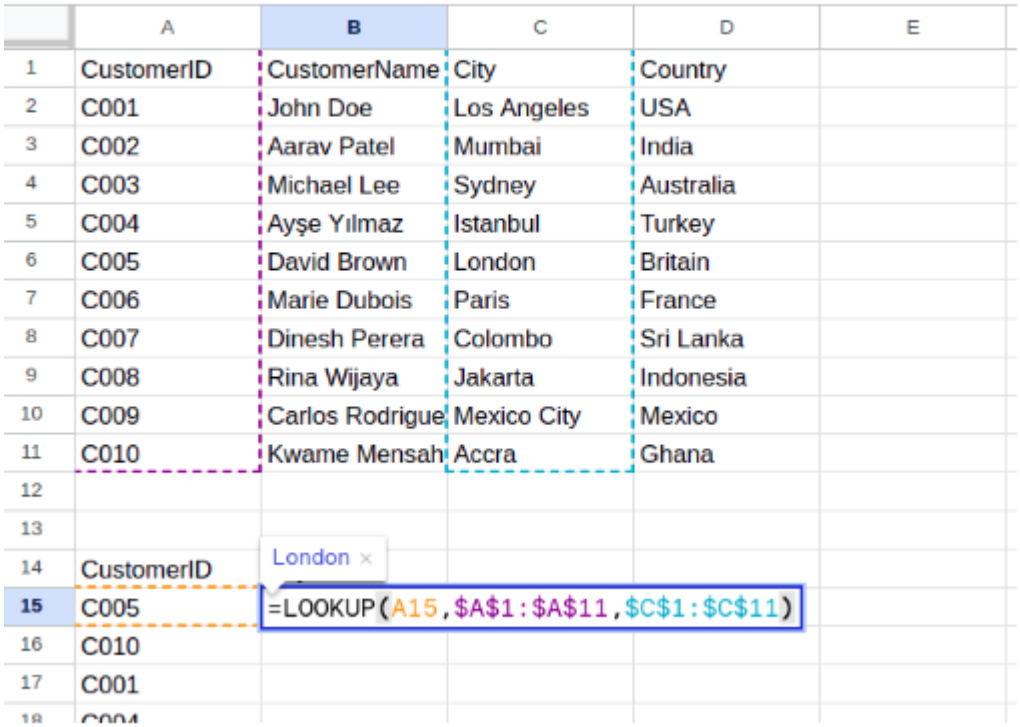
Excel's LOOKUP function is useful for finding a value inside a range or array. There are two types of it: array and vector.

Syntax:

=LOOKUP(lookup_value, lookup_vector, [result_vector])

Example: The LOOKUP function may locate the name linked to a certain ID. If you have a list of employee IDs in column A and their corresponding names in column B, the result will be the ID that you are looking for.

```
=LOOKUP(A15,$A$1:$A$11,$C$1:$C$11)
```



	A	B	C	D	E
1	CustomerID	CustomerName	City	Country	
2	C001	John Doe	Los Angeles	USA	
3	C002	Aarav Patel	Mumbai	India	
4	C003	Michael Lee	Sydney	Australia	
5	C004	Ayşe Yılmaz	Istanbul	Turkey	
6	C005	David Brown	London	Britain	
7	C006	Marie Dubois	Paris	France	
8	C007	Dinesh Perera	Colombo	Sri Lanka	
9	C008	Rina Wijaya	Jakarta	Indonesia	
10	C009	Carlos Rodrigue	Mexico City	Mexico	
11	C010	Kwame Mensah	Accra	Ghana	
12					
13					
14	CustomerID	London			
15	C005	=LOOKUP(A15,$A$1:$A$11,$C$1:$C$11)			
16	C010				
17	C001				

2. VLOOKUP Function

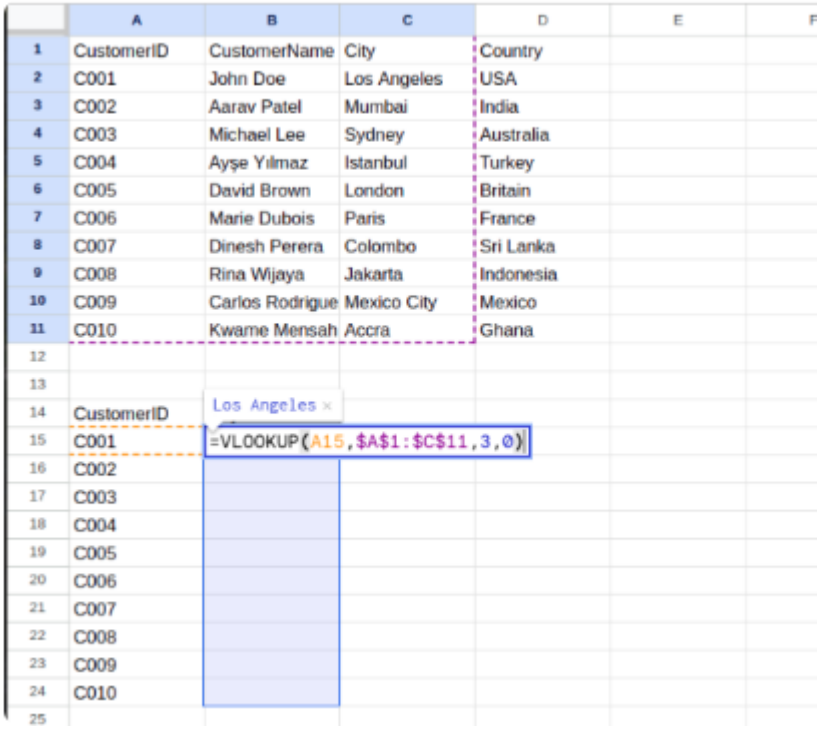
Excel's most frequently used function is the VLOOKUP function. It retrieves a value from another column in the same row after looking for a value in the first column of a range.

Syntax

=VLOOKUP(lookup_value, table_array, col_index_num, [range_lookup])

Example: To find the City of a customer in the Sales table using the CustomerID:

=VLOOKUP(A15,$A$1:$C$11,3,0)



3. HLOOKUP Function

The HLOOKUP function is another LOOKUP function in Excel. After searching the top row of a range for a given value, it retrieves a value from a different row in the same column.

=HLOOKUP(lookup_value, table_array, row_index_num, [range_lookup])

Example: Assuming the Customers table is transposed horizontally (customer data in rows):

```
=HLOOKUP(F8,$F$1:$P$4,3,0)                                    Copy Code
```

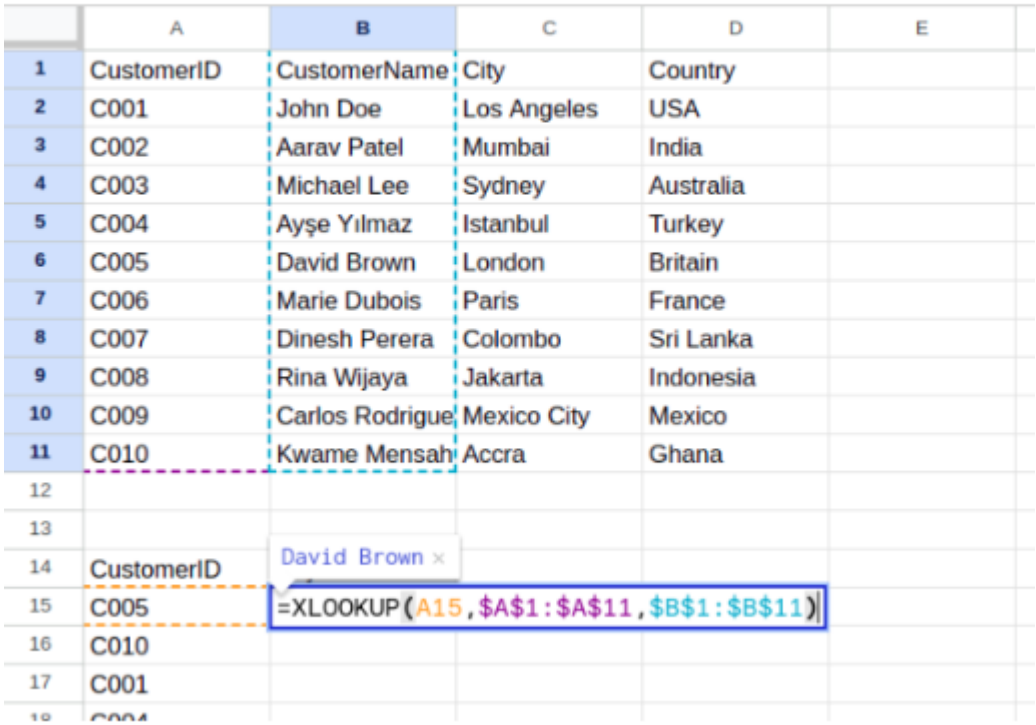
4. XLOOKUP Function

XLOOKUP was created to replace VLOOKUP and HLOOKUP. It has improved features, and it is more powerful and versatile.

=XLOOKUP(lookup_value, lookup_array, return_array, [if_not_found], [match_mode], [search_mode])

Example: To find the city of an customer's Name using their ID from a table where IDs are in column A and Name in column B:

```
=XLOOKUP(A15,$A$1:$A$11,$B$1:$B$11)
```



	A	B	C	D	E
1	CustomerID	CustomerName	City	Country	
2	C001	John Doe	Los Angeles	USA	
3	C002	Aarav Patel	Mumbai	India	
4	C003	Michael Lee	Sydney	Australia	
5	C004	Ayşe Yılmaz	Istanbul	Turkey	
6	C005	David Brown	London	Britain	
7	C006	Marie Dubois	Paris	France	
8	C007	Dinesh Perera	Colombo	Sri Lanka	
9	C008	Rina Wijaya	Jakarta	Indonesia	
10	C009	Carlos Rodrigue	Mexico City	Mexico	
11	C010	Kwame Mensah	Accra	Ghana	
12					
13					
14	CustomerID	David Brown ×			
15	C005	=XLOOKUP(A15,$A$1:$A$11,$B$1:$B$11)			
16	C010				
17	C001				

Please read the book for further topics

Data Analysis with Excel : Manisa Nigam

Mastering Advanced Excel : Ritu Arora