PROFESSIONAL ELECTIVE COURSES: VERTICALS
VERTICAL 1: DATA SCIENCE
CCS346 EXPLORATORY DATA ANALYSIS
UNIT I - EXPLORATORY DATA ANALYSIS

EDA fundamentals – Understanding data science – Significance of EDA – Making sense of data – Comparing EDA with classical and Bayesian analysis – Software tools for EDA - Visual Aids for EDA- Data transformation techniques-merging database, reshaping and pivoting, Transformation techniques.

Data science

Data science is an interdisciplinary field that combines scientific methods, processes, algorithms, and systems to extract insights and knowledge from structured and unstructured data. It involves applying techniques from various fields such as statistics, mathematics, computer science, and domain expertise to analyze and interpret data in order to solve complex problems, make predictions, and drive decision-making.

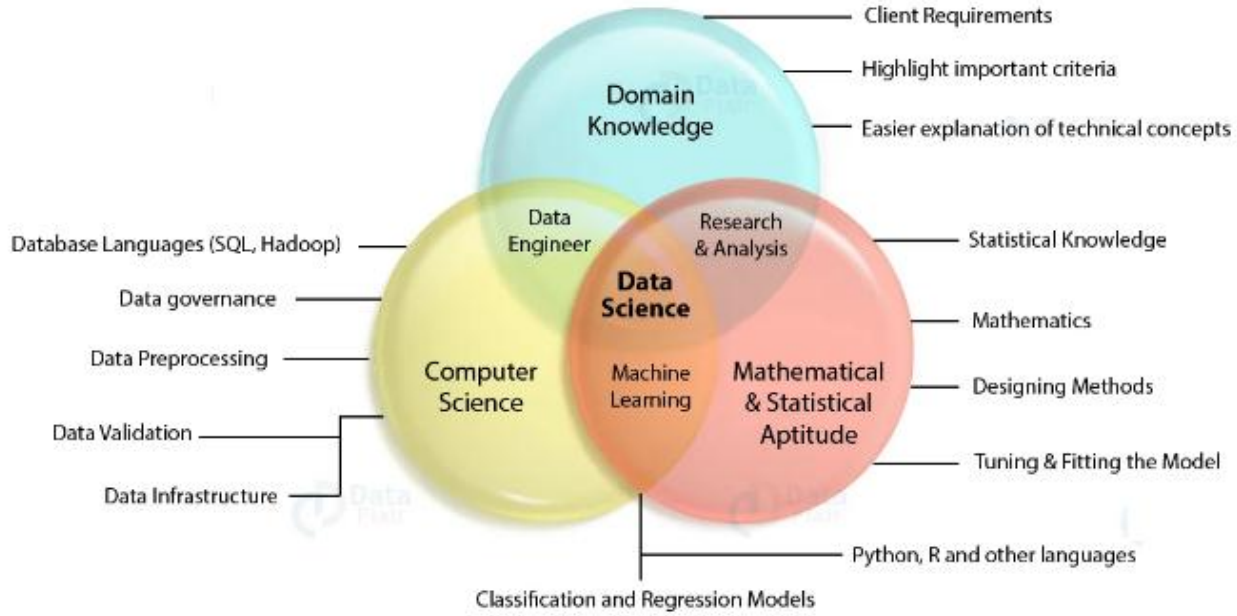


Fig. Data Science

Data science encompasses a range of activities, including data collection, data cleaning and preprocessing, exploratory data analysis, feature engineering, modeling, and evaluation. It often involves working with large and diverse datasets, utilizing statistical and machine learning techniques, and leveraging computational tools and technologies to extract meaningful patterns, correlations, and insights from data.

Data scientists use a combination of analytical, programming, and problem-solving skills to formulate research questions, design experiments, develop models and algorithms, and interpret the results. They may work with structured data from

databases, spreadsheets, or relational data sources, as well as unstructured data from text, images, videos, or social media.

The ultimate goal of data science is to uncover valuable insights and knowledge that can drive data-driven decision-making, optimize processes, improve efficiency, and enable businesses or organizations to gain a competitive edge. Data science has applications in a wide range of domains, including business, finance, healthcare, marketing, social sciences, and more.

Understanding Data Science

Data science is an interdisciplinary field that combines various techniques, tools, and methodologies to extract insights and knowledge from data. It involves applying statistical analysis, machine learning, and computational methods to solve complex problems, make predictions, and drive decision-making.

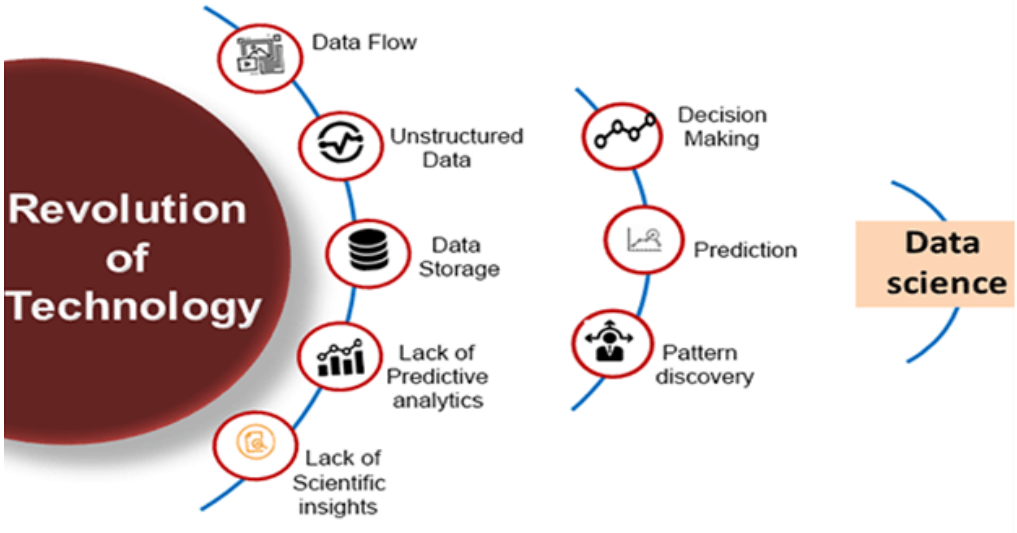


Fig. Need for Data Science

Data Requirements: Data requirements refer to the specific needs to address a particular problem or achieve specific objectives. It involves identifying the types of data required, their sources, formats, quality, and quantity.

Data Collection: Data collection is the process of gathering or acquiring data from various sources. This may include surveys, experiments, observations, web scraping, sensors, or accessing existing datasets. The data collected should align with the defined data requirements.

Data Processing: Data processing involves transforming raw data into a more usable and structured format. This step may include data integration, data aggregation, data transformation, and data normalization to prepare the data for further analysis.

Data Cleaning: Data cleaning, also known as data cleansing or data scrubbing, is the process of identifying and rectifying errors, inconsistencies, missing values, and outliers

in the dataset. It aims to improve data quality and ensure that the data is accurate and reliable for analysis.

EDA (Exploratory Data Analysis)

Exploratory Data Analysis is an approach in data analysis that focuses on summarizing, visualizing, and understanding the main characteristics of a dataset. EDA involves using statistical and visualization techniques to examine the data, identify patterns, uncover relationships between variables, detect anomalies, and gain insights that can inform further analysis or decision-making.

The main objectives of EDA are to:

- Understand the structure and nature of the data.
- Identify any missing values, outliers, or inconsistencies in the data.
- Discover patterns, trends, and relationships between variables.
- Extract meaningful insights and generate hypotheses for further analysis.
- Validate assumptions and check data quality.

EDA typically involves various techniques, such as data visualization (e.g., plots, charts, graphs), summary statistics, descriptive statistics, correlation analysis, distribution analysis, and data preprocessing. These techniques help data analysts and scientists gain a deeper understanding of the dataset and guide them in making informed decisions regarding data cleaning, feature engineering, modeling, or hypothesis testing.

Exploratory Data Analysis (EDA) is of great significance in data science and analysis. Here are some key reasons why EDA is crucial:

Understanding the Data: EDA helps in gaining a deep understanding of the dataset at hand. It allows data scientists to become familiar with the structure, contents, and characteristics of the data, including variables, their distributions, and relationships. This understanding is essential for making informed decisions throughout the data analysis process.

Data Quality Assessment: EDA helps identify and assess the quality of the data. It allows for the detection of missing values, outliers, inconsistencies, or errors in the dataset. By addressing data quality issues, EDA helps ensure that subsequent analyses and models are built on reliable and accurate data.

Feature Selection and Engineering: EDA aids in selecting relevant features or variables for analysis. By examining relationships and correlations between variables, EDA can guide the identification of important predictors or features that significantly contribute to the desired outcome. EDA can also inspire the creation of new derived features or transformations that improve model performance.

Uncovering Patterns and Insights: EDA enables the discovery of patterns, trends, and relationships within the data. By using visualization techniques and summary statistics, EDA helps uncover valuable insights and potential associations between variables. These insights can drive further analysis, hypothesis generation, or the formulation of research questions.

Hypothesis Generation and Testing: EDA plays a crucial role in generating hypotheses for further investigation. By exploring the data, researchers can identify potential relationships or patterns and formulate hypotheses to test formally. EDA can also provide evidence or insights to support or refute existing hypotheses.

Decision-Making Support: EDA assists in making data-driven decisions. By visualizing and summarizing the data, EDA provides insights that can inform strategic and tactical decisions. It helps stakeholders understand the implications of the data and facilitates evidence-based decision-making.

Data Visualization and Communication: EDA utilizes various data visualization techniques to present the findings in a clear and understandable manner. Visualizations enable effective communication of complex information, making it easier for stakeholders to grasp the insights derived from the data.

Steps involved in Exploratory Data Analysis (EDA)

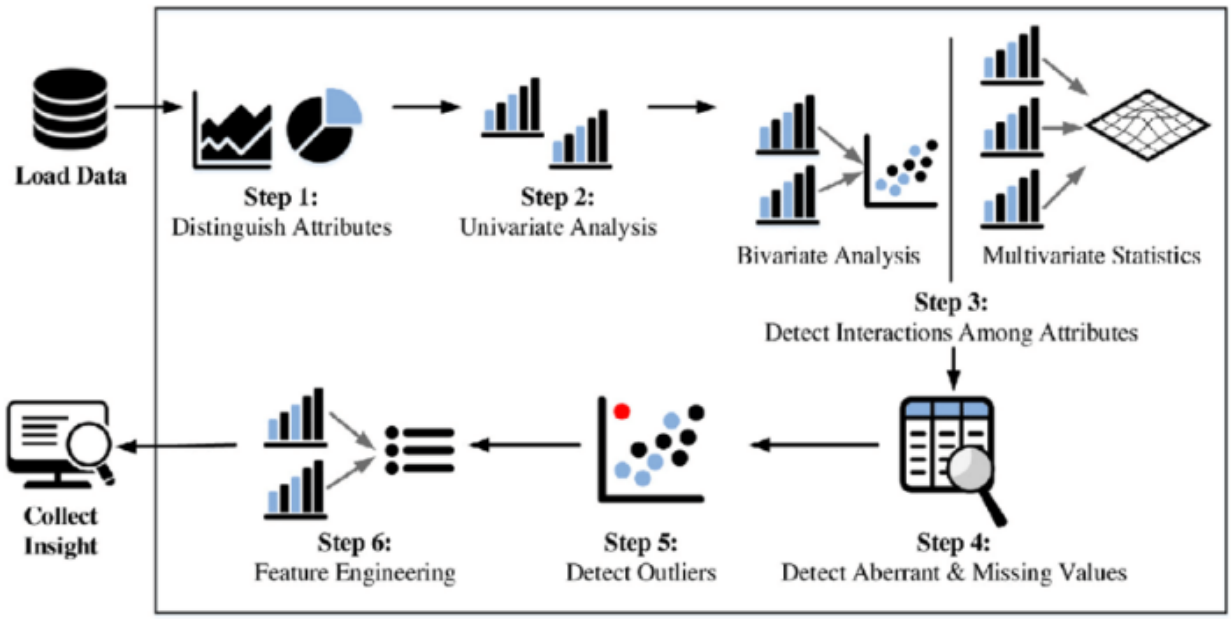
It can vary depending on the specific dataset and objectives. However, here is a general outline of the key steps in EDA:

Understand the Data: Start by getting familiar with the dataset, its structure, and the variables it contains. Understand the data types (e.g., numerical, categorical) and the meaning of each variable.

Data Cleaning: Clean the dataset by handling missing values, outliers, and inconsistencies. Identify and handle missing data appropriately (e.g., imputation, deletion) based on the context and data quality requirements. Treat outliers and inconsistent values by either correcting or removing them if necessary.

Handle Data Transformations: Explore and apply necessary data transformations such as scaling, normalization, or logarithmic transformations to make the data suitable for analysis. This step may be required to meet assumptions of certain statistical methods or to improve the interpretability of the data.

Summary Statistics: Compute and analyze summary statistics for each variable. This includes measures such as mean, median, mode, standard deviation, range, quartiles, and other descriptive statistics. Summary statistics provide an initial understanding of the data distribution and basic insights.



Data Visualization: Utilize various visualization techniques to explore the data visually. Create histograms, scatter plots, box plots, bar charts, heatmaps, or other relevant visualizations to understand the patterns, distributions, and relationships between variables. Visualizations can reveal insights that may not be apparent from summary statistics alone.

Identify Relationships and Correlations: Analyze the relationships between variables using correlation analysis or other statistical techniques. Identify variables that are highly correlated or exhibit strong associations. Correlation matrices, scatter plots, or heatmaps can be useful for visualizing relationships between numerical variables. Cross-tabulations or stacked bar charts can be used for categorical variables.

Exploring Time Series Data: If the dataset involves time series data, analyze trends, seasonality, and other temporal patterns. Use line plots, time series decomposition, autocorrelation plots, or other relevant techniques to explore the temporal behavior of the data.

Feature Engineering: Based on the insights gained from EDA, consider creating new derived features or transformations that may enhance the predictive power or interpretability of the data. This can involve mathematical operations, combinations of variables, or domain-specific transformations.

Iterative Analysis: EDA is often an iterative process. Repeat the above steps as needed, diving deeper into specific variables or subsets of the data based on emerging patterns or research questions. Refine the analysis based on new insights or feedback from stakeholders.

Documentation and Reporting: Document the findings, insights, and visualizations generated during the EDA process. Prepare a report or presentation that effectively communicates the key findings, patterns, and relationships discovered. Include visualizations, summary statistics, and any relevant observations to support the conclusions.

Remember, the steps and techniques used in EDA can be flexible and iterative, tailored to the specific dataset and research objectives. The goal is to gain a comprehensive understanding of the data, identify patterns, and generate hypotheses for further analysis.

Define Data

In data science, "data" refers to the raw, unprocessed, and often vast quantities of information that is collected or generated from various sources. It can exist in different formats, such as structured data (organized and well-defined), semi-structured data (partially organized), or unstructured data (lacks a predefined structure).

Data serves as the foundation for data science activities and analysis. It can include numbers, text, images, audio, video, sensor readings, transaction records, social media posts, and much more. Data can be generated from diverse sources, including databases, spreadsheets, web scraping, sensors, surveys, or online platforms.

Categorization of Data

In data science, data is typically categorized into two main types:

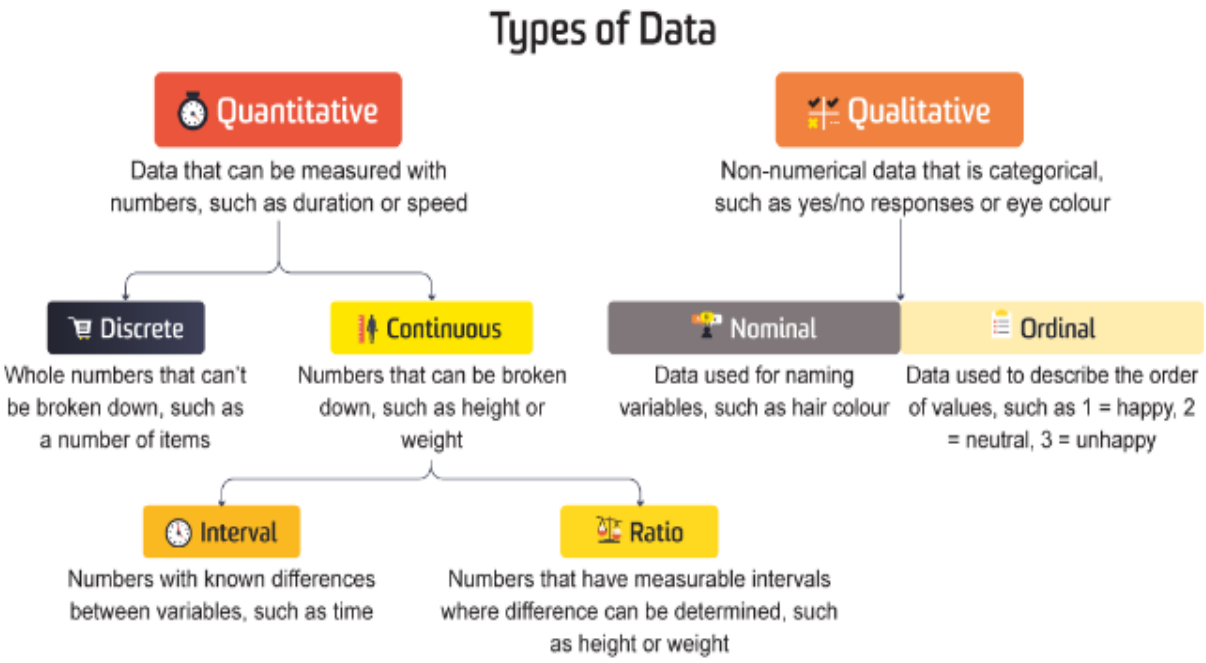
Quantitative Data: Also known as numerical or structured data, quantitative data represents measurable quantities or variables. It includes attributes such as age, temperature, sales figures, or stock prices. Quantitative data is typically analyzed using statistical techniques and mathematical models.

Examples of quantitative data:

- Scores of tests and exams e.g. 74, 67, 98, etc.
- The weight of a person.
- The temperature in a roo

Qualitative Data: Also known as categorical or unstructured data, qualitative data represents non-numeric attributes or characteristics. It includes categories such as gender, color, occupation, or sentiment. Qualitative data is often analyzed using techniques such as text mining, sentiment analysis, or topic modeling.
Examples of qualitative data:

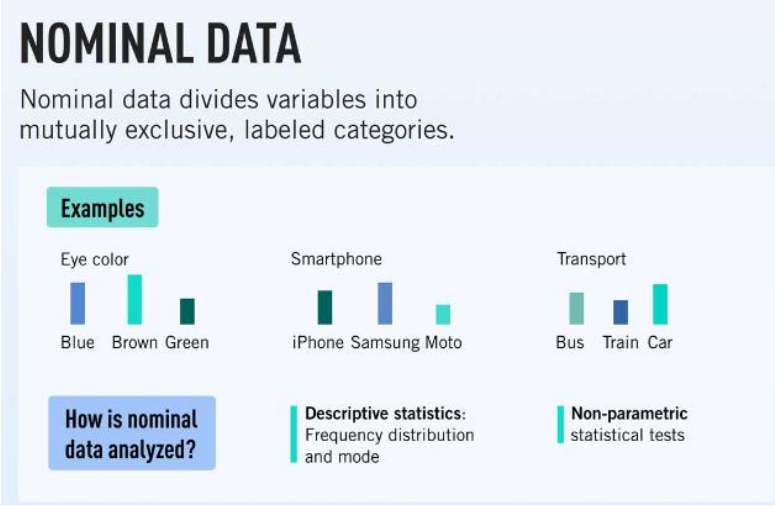
- Colors e.g. the color of the sea
- Popular holiday destinations such as Switzerland, New Zealand, South Africa, etc.
- Ethnicity such as American Indian, Asian, etc.



Qualitative Data Types

Nominal Data

This data type is used just for labeling variables, without having any quantitative value. It just names a thing without applying for any particular order.

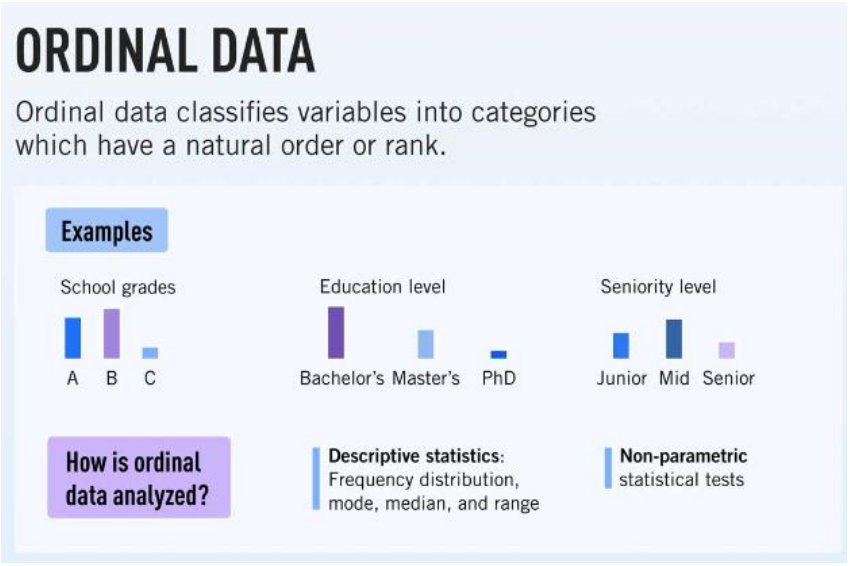


Examples of Nominal Data:

- Gender (Women, Men)
- Hair color (Blonde, Brown, Brunette, Red, etc.)
- Marital status (Married, Single, Widowed)

Ordinal Data

Ordinal data is qualitative data for which their values have some kind of relative position. These kinds of data can be considered "in-between" qualitative and quantitative data. The ordinal data only shows the sequences and cannot use for statistical analysis. Compared to nominal data, ordinal data have some kind of order that is not present in nominal data.



Examples of Ordinal Data:

- When companies ask for feedback, experience, or satisfaction on a scale of 1 to 10
- Letter grades in the exam (A, B, C, D, etc.)
- Ranking of people in a competition (First, Second, Third, etc.)
- Economic Status (High, Medium, and Low)

- Education Level (Higher, Secondary, Primary)

Quantitative Data Types

Discrete Data

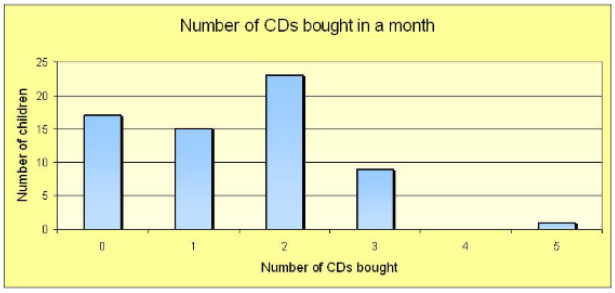
The term discrete means distinct or separate. The discrete data contain the values that fall under integers or whole numbers. These data can't be broken into decimal or fraction values.The discrete data are countable and have finite values; their subdivision is not possible. These data are represented mainly by a bar graph, number line, or frequency table.

Examples of discrete data:

- Number of students in a class.
- Number of workers in a company.
- Number of test questions you answered correctly.

Bar charts can be used to display discrete numerical data.

For example, this bar graph shows the number of CDs bought by a group of children i a given month.

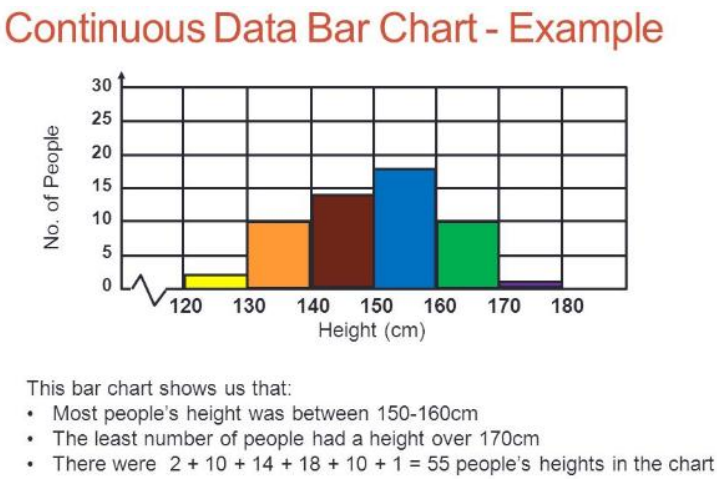


Continuous Data

Continuous data are in the form of fractional numbers. It can be the version of an android phone, the height of a person, the length of an object, etc. Continuous data represents information that can be divided into smaller levels. The continuous variable can take any value within a range.

Examples of continuous data:

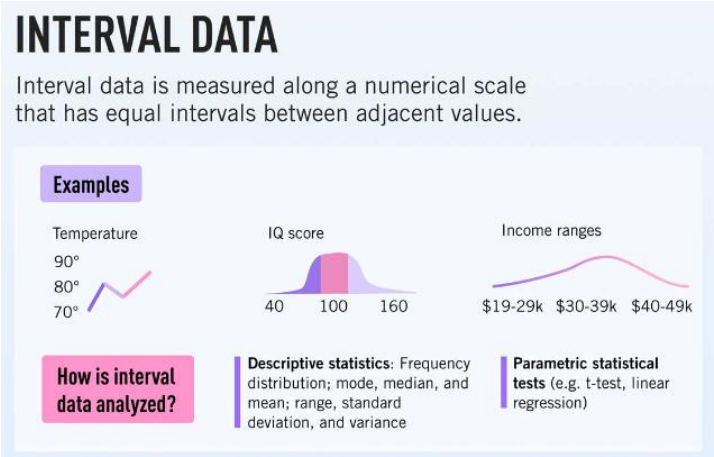
- Height of a person - 62.04762 inches, 79.948376 inches
- Speed of a vehicle
- "Time-taken" to finish the work
- Wi-Fi Frequency
- Market share price



Interval Data

The interval level is a numerical level of measurement which, like the ordinal scale, places variables in order. The interval scale has a known and equal distance between each value on the scale (imagine the points on a thermometer). Unlike the ratio scale (the fourth level of measurement), interval data has no true zero; in other words, a value of zero on an interval scale does not mean the variable is absent.

A temperature of zero degrees Fahrenheit doesn't mean there is "no temperature" to be measured—rather, it signifies a very low or cold temperature.



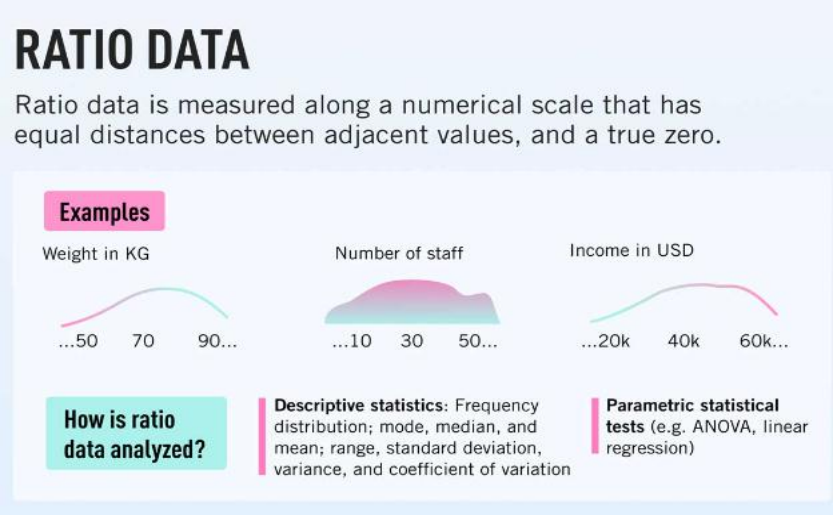
Examples of Interval data:

- Temperature (°C or F, but not Kelvin)
- Dates (1055, 1297, 1976, etc.)
- Time Gap on a 12-hour clock (6 am, 6 pm)
- IQ score
- Income categorized as ranges ($30-39k, $40-49k, $50-59k, and so on)

Ratio Data

The fourth and final level of measurement is the ratio level. Just like the interval scale, the ratio scale is a quantitative level of measurement with equal intervals between each point. Difference between Interval scale and Ratio scale is that it has a true zero. That is, a value of zero on a ratio scale means that the variable you're measuring is absent.

Population is a good example of ratio data. If you have a population count of zero people, this means there are no people!

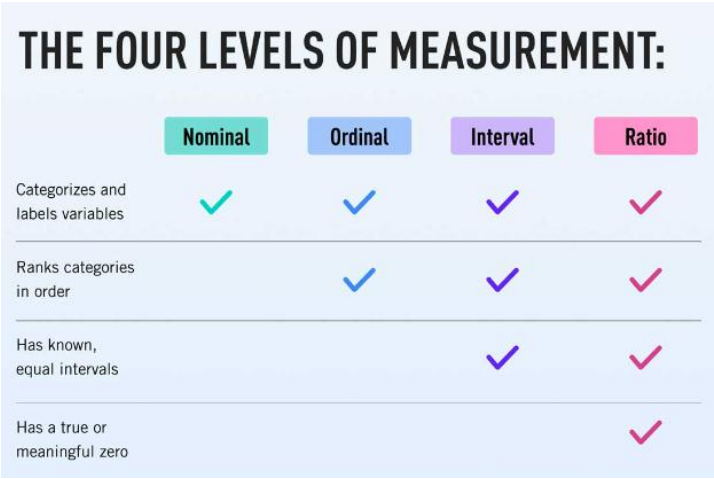


Example of Ratio data:

- Weight in grams (continuous)
- Number of employees at a company (discrete)
- Speed in miles per hour (continuous)
- Length in centimeters (continuous)
- Age in years (continuous)
- Income in dollars (continuous)

Measurement scales

Measurement scales, also known as data scales or levels of measurement, define the properties and characteristics of the data collected or measured. There are four commonly recognized measurement scales:



Nominal Scale:

- The nominal scale is the lowest level of measurement. It represents data that can be categorized into distinct and mutually exclusive groups or categories. The categories in a nominal scale have no inherent order or ranking.
- Examples of nominal scale data include gender (male/female), eye color (blue/green/brown), or types of cars (sedan/SUV/hatchback).
- Nominal data can be represented using labels or codes.

Ordinal Scale:

- The ordinal scale represents data with categories that have a natural order or ranking. In addition to the properties of the nominal scale, ordinal data allows for the relative positioning or hierarchy between the categories. However, the intervals between the categories may not be equal.
- Examples of ordinal scale data include rating scales (e.g., 1-5 scale indicating satisfaction levels), education levels (e.g., high school, bachelor's, master's), or performance rankings (first, second, third place). Ordinal data can be represented using labels, codes, or numerical rankings.

Interval Scale:

- The interval scale represents data with categories that have equal intervals between the values. In addition to the properties of the ordinal scale, interval data allows for meaningful comparisons of the intervals between the categories. However, it does not have a true zero point.
- Examples of interval scale data include calendar dates, temperature measured in Celsius or Fahrenheit, or years. Interval data allows for mathematical operations such as addition and subtraction but does not support meaningful multiplication or division.

Ratio Scale:

- The ratio scale is the highest level of measurement. It represents data with categories that have equal intervals between the values and possess a true zero point. In addition to the properties of the interval scale, ratio data allows for all mathematical operations and meaningful ratios.
- Examples of ratio scale data include weight, length, time duration, or count. Ratio scale data provides a complete and meaningful representation of the data.

Understanding the measurement scale of the data is important for selecting appropriate statistical techniques, visualization methods, and modeling approaches. Different scales require different levels of analysis and interpretation.

Provides:	Nominal	Ordinal	Interval	Ratio
The "order" of values is known		✔	✔	✔
"Counts," aka "Frequency of Distribution"	✔	✔	✔	✔
Mode	✔	✔	✔	✔
Median		✔	✔	✔
Mean			✔	✔
Can quantify the difference between each value			✔	✔
Can add or subtract values			✔	✔
Can multiple and divide values				✔
Has "true zero"				✔

Comparing EDA with classical and Bayesian analysis

There are several approaches to data analysis.

For classical analysis, the sequence is

Problem => Data => Model => Analysis => Conclusions

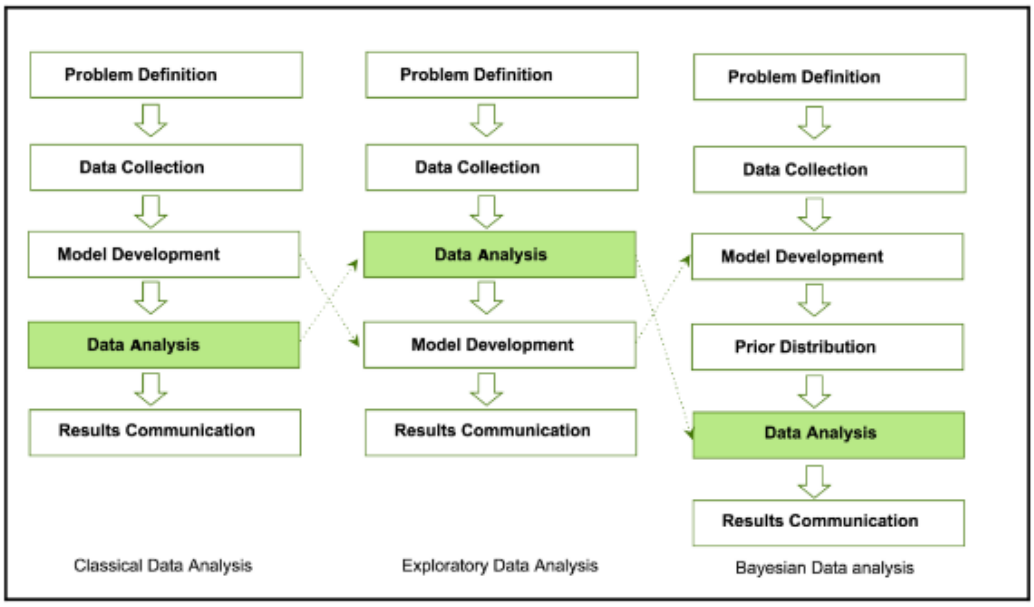
For EDA, the sequence is

Problem => Data => Analysis => Model => Conclusions

For Bayesian, the sequence is

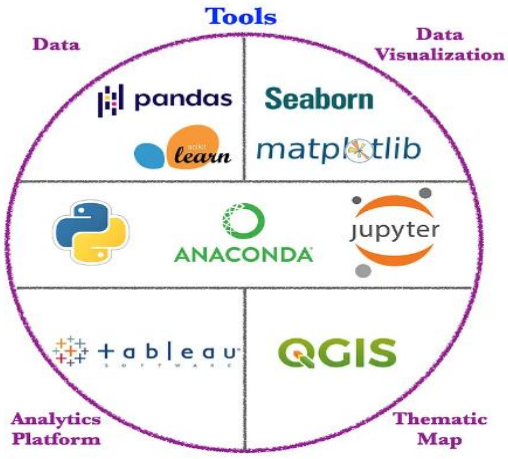
Problem => Data => Model => Prior Distribution => Analysis => Conclusions

- Classical data analysis:
  - For the classical data analysis approach, the problem definition and data collection step are followed by model development, which is followed by analysis and result communication.
- Exploratory data analysis approach:
  - For the EDA approach, it follows the same approach as classical data analysis except the model imposition and the data analysis steps are swapped. The main focus is on the data, its structure, outliers, models, and visualizations. Generally, in EDA, we do not impose any deterministic or probabilistic models on the data.
- Bayesian data analysis approach:
  - The Bayesian approach incorporates prior probability distribution knowledge into the analysis steps as shown in the following diagram



Software tools available for EDA

There are several software tools available for performing Exploratory Data Analysis (EDA). Here are some commonly used ones:



- *Python:* This is an open source programming language widely used in data analysis, data mining, and data science
- *R programming language:* R is an open source programming language that is widely utilized in statistical computation and graphical data analysis
- *Weka:* This is an open source data mining package that involves several EDA tools and algorithms
- *Jupyter Notebook / JupyterLab*: Jupyter is an open-source web-based platform that supports multiple programming languages, including Python and R.
- *SPSS*: IBM SPSS Statistics is a comprehensive software package for statistical analysis. It provides a range of tools for data exploration, descriptive statistics, hypothesis testing, and advanced modeling techniques.

- *KNIME:* KNIME (Konstanz Information Miner) is an open-source data analytics platform that allows users to visually design data workflows.

Visual Aids for EDA

Two important goals of data scientists are:

- Extract knowledge from the data.
- Present the data to stakeholders.

Presenting results to stakeholders is very complex because the audiences have not enough technical knowledge. Hence, visual aids are very useful tools. We are going to learn different types of techniques that can be used in the visualization of data.

- Line chart
- Bar chart
- Scatter plot
- Area plot and stacked plot
- Pie chart
- Table chart
- Polar chart
- Histogram
- Lollipop chart
- Choosing the best chart
- Other libraries to explore

Matplotlib is a data visualization library in Python. The pyplot, a sublibrary of matplotlib, is a collection of functions that helps in creating a variety of charts.

Line chart

Line charts are used to represent the relation between two data X and Y on a different axis. Here we will see some of the examples of a line chart in Python :

Steps to Plot a Line Chart in Python using Matplotlib

Step 1: Install the Matplotlib package

```
pip install matplotlib
```

Step 2: Gather the data for the Line chart

Next, gather the data for your Line chart. For example, let's use the following data about two variables:

- Year
- Unemployment_rate

Here is the complete data:

Year	Unemployment_Rate
1920	9.8
1930	12
1940	8
1950	7.2
1960	6.9
1970	7
1980	6.5
1990	6.2
2000	5.5
2010	3.3

The ultimate goal is to depict the above data using a Line chart.

Step 3: Capture the data in Python
You can capture the above data in Python using the following two Lists:

```
year = [1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010]
unemployment_rate = [9.8, 12, 8, 7.2, 6.9, 7, 6.5, 6.2, 5.5, 3.3]
```

Step 4: Plot a Line chart in Python using Matplotlib
For the final step, you may use the template below in order to plot the Line chart in Python:

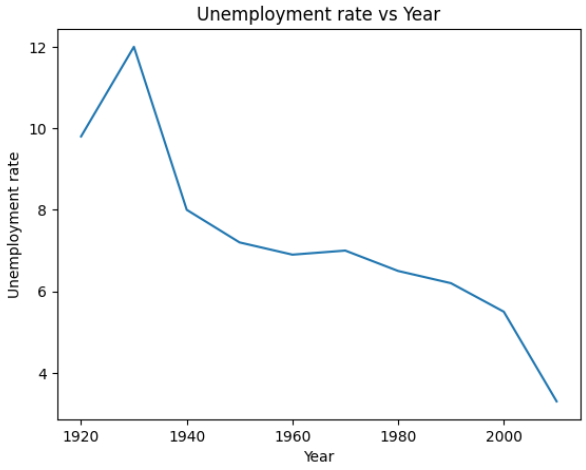
```
import matplotlib.pyplot as plt

x_axis = ['value_1', 'value_2', 'value_3', ...]
y_axis = ['value_1', 'value_2', 'value_3', ...]

plt.plot(x_axis, y_axis)
plt.title('title name')
plt.xlabel('x_axis name')
plt.ylabel('y_axis name')
plt.show()
```

Here is the code for our example:

```
/* Python program to create a Line chart */
import matplotlib.pyplot as plt

year = [1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010]
unemployment_rate = [9.8, 12, 8, 7.2, 6.9, 7, 6.5, 6.2, 5.5, 3.3]

plt.plot(year, unemployment_rate)
plt.title('Unemployment rate vs Year')
```

```
plt.xlabel('Year')
plt.ylabel('Unemployment rate')
plt.show()
```

Output:

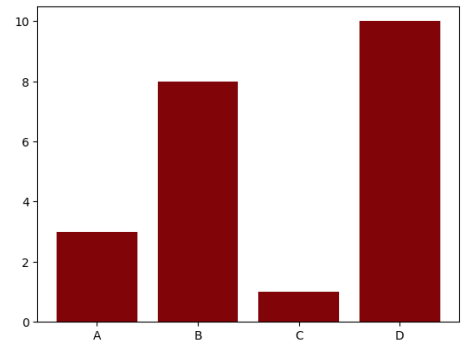


Bar charts

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories.

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y, color = 'maroon')
plt.show()
```

Output:



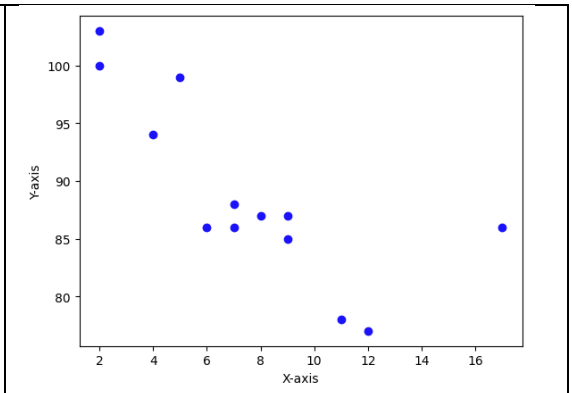
Scatter plots

Scatter plots are used to observe relationship between variables and uses dots to represent the relationship between them. The scatter() method in the matplotlib library is used to draw a scatter plot. Scatter plots are widely used to represent relation among variables and how change in one affects the other.

```
/* Python program to create scatter plots*/

import matplotlib.pyplot as plt


x =[5, 7, 8, 7, 2, 17, 2, 9,
```

Output:

```
    4, 11, 12, 9, 6]

y =[99, 86, 87, 88, 100, 86,
    103, 87, 94, 78, 77, 85, 86]

plt.scatter(x, y, c ="blue")

plt.xlabel("X-axis")
plt.ylabel("Y-axis")

# To show the plot
plt.show()
```



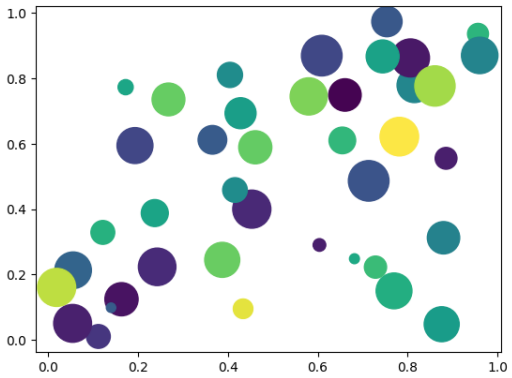
Bubble chart

Bubble plots are an improved version of the scatter plot. In a scatter plot, there are two dimensions x, and y. In a bubble plot, there are three dimensions x, y, and z, where the third dimension z denotes weight. Here, each data point on the graph is shown as a bubble. Each bubble can be illustrated with a different color, size, and appearance.

/* Python program to create Bubble plots*/

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.rand(40)
y = np.random.rand(40)
z = np.random.rand(40)
colors = np.random.rand(40)
# use the scatter function
plt.scatter(x, y, s=z*1000,c=colors)
plt.show()
```

Output:



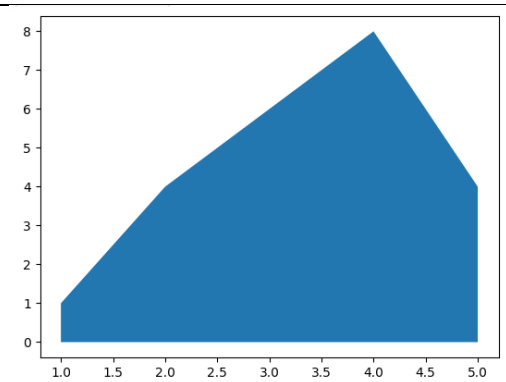
Area Chart

An area chart is really similar to a line chart, except that the area between the x axis and the line is filled in with color or shading. It represents the evolution of a numeric variable.

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

# Create data
x=range(1,6)
y=[1,4,6,8,4]
```

Output:

```
# Area plot
plt.fill_between(x, y)
plot.show()
```



Stacked Area Chart

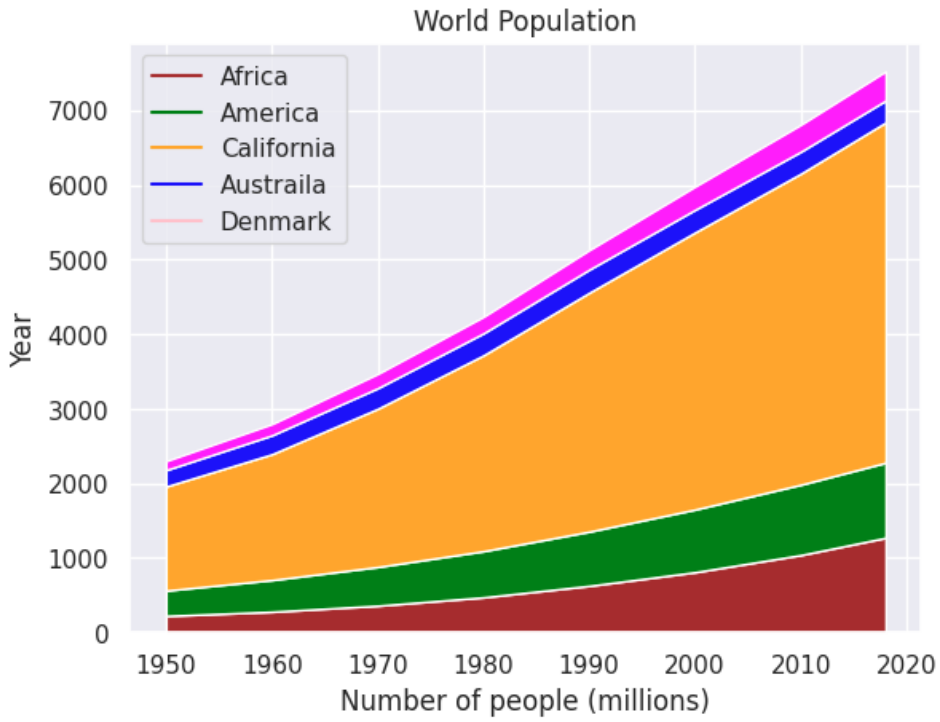
A stacked area chart is the extension of a basic area chart. It displays the evolution of the value of several groups on the same graphic. The values of each group are displayed on top of each other, what allows checking on the same figure the evolution of both the total of a numeric variable, and the importance of each group.

The stacked plot owes its name to the fact that it represents the area under a line plot and that several such plots can be stacked on top of one another, giving the feeling of a stack. The stacked plot can be useful when we want to visualize the cumulative effect of multiple variables being plotted on the y axis.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

# Population of different countries in billions
Africa = [228, 284, 365, 477, 631, 814, 1044, 1275]
America = [340, 425, 519, 619, 727, 840, 943, 1006]
California = [1394, 1686, 2120, 2625, 3202, 3714, 4169, 4560]
Austraila = [220, 253, 276, 295, 310, 303, 294, 293]
Denmark = [120, 150, 190, 220, 260, 310, 360, 390]

year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2018]

# Create placeholders for plot and add required color
plt.plot([],[], color='brown', label='Africa')
plt.plot([],[], color='green', label='America')
plt.plot([],[], color='orange', label='California')
plt.plot([],[], color='blue', label='Austraila')
plt.plot([],[], color='pink', label='Denmark')

# Add stacks to the plot
plt.stackplot(year, Africa, America, California, Austraila,
Denmark, colors=['brown', 'green', 'orange', 'blue',
```

```
'magenta'])
plt.legend()

# Add Labels
plt.legend(loc='upper left')
plt.title('World Population')
plt.xlabel('Number of people (millions)')
plt.ylabel('Year')

# Display on the screen
plt.show()
```

Output:

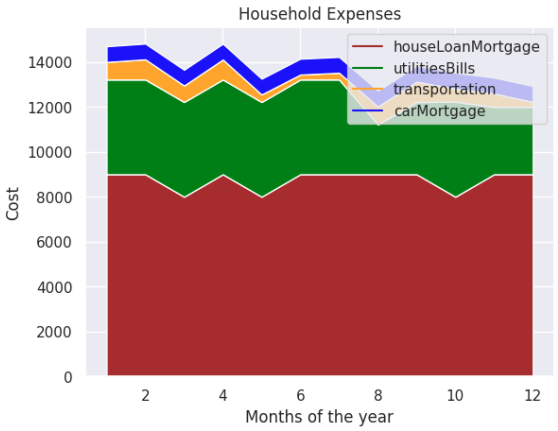


Python program to display area and stacked chart:

```
# House loan Mortgage cost per month for a year
houseLoanMortgage = [9000, 9000, 8000, 9000, 8000,
                     9000, 9000, 9000,  9000, 8000, 9000, 9000]
# Utilities Bills for a year
utilitiesBills = [4218, 4218, 4218, 4218, 4218,
                  4218, 4219, 2218, 3218, 4233, 3000, 3000]
# Transportation bill for a year
transportation = [782, 900, 732, 892, 334, 222,
                  300, 800, 900, 582, 596, 222]
# Car mortgage cost for one year
```

```
carMortgage = [700, 701, 702, 703, 704,
               705, 706, 707, 708, 709, 710, 711]

import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

months= [x for x in range(1,13)]

# Create placeholders for plot and add required color
plt.plot([],[], color='brown', label='houseLoanMortgage')
plt.plot([],[], color='green', label='utilitiesBills')
plt.plot([],[], color='orange', label='transportation')
plt.plot([],[], color='blue', label='carMortgage')

# Add stacks to the plot
plt.stackplot(months, houseLoanMortgage, utilitiesBills,
transportation, carMortgage, colors=['brown', 'green', 'orange',
'blue'])
plt.legend()

# Add Labels
plt.title('Household Expenses')
plt.xlabel('Months of the year')
plt.ylabel('Cost')

# Display on the screen
plt.show()
```

Output:



Pie Chart

A Pie Chart is a circular statistical plot that can display only one series of data. The area of the chart is the total percentage of the given data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. The area of the wedge is determined by the length of the arc of the wedge. The area of a wedge represents the relative percentage of that part with respect to whole data. Pie

charts are commonly used in business presentations like sales, operations, survey results, resources, etc as they provide a quick summary.

```
import matplotlib.pyplot as plt

langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
plt.pie(students, labels=langs)
plt.title('Students taking different programming languages')
plt.axis('equal')
plt.show()
```

Output:

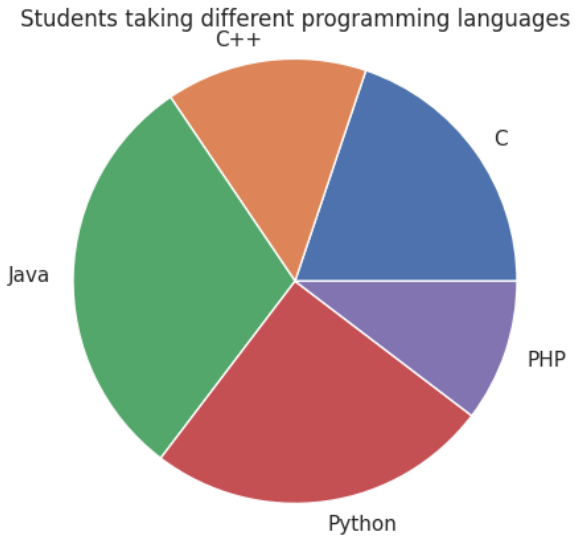


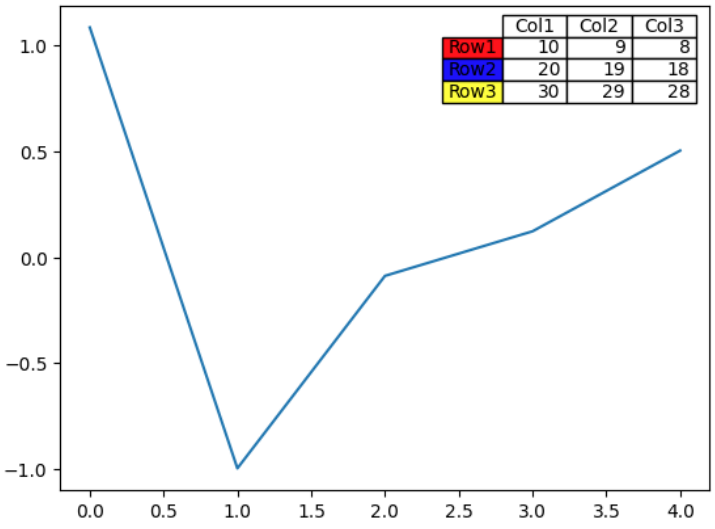
Table Chart

Matplotlib.pyplot.table() is a subpart of matplotlib library in which a table is generated using the plotted graph for analysis. This method makes analysis easier and more efficient as tables give a precise detail than graphs. The matplotlib.pyplot.table creates tables that often hang beneath stacked bar charts to provide readers insight into the data generated by the above graph.

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

plt.figure()
ax = plt.gca()
a = np.random.randn(5)
```

```
#defining the attributes
col_labels = ['Col1','Col2','Col3']
row_labels = ['Row1','Row2','Row3']
table_vals = [[10, 9, 8], [20, 19, 18], [30, 29, 28]]
row_colors = ['red', 'blue', 'yellow']
#plotting
my_table = plt.table(cellText=table_vals,
                     colWidths=[0.1] * 3,
                     rowLabels=row_labels,
                     colLabels=col_labels,
                     rowColours=row_colors,
                     loc='upper right')

plt.plot(a)
plt.show()
```

Output



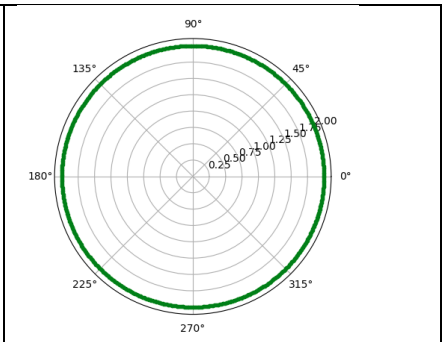
Polar chart or spider web plot

Matplotlib provides the module and functions to plot the coordinates on polar axes. A point in polar co-ordinates is represented as (r, theta). Here, r is its distance from the origin and theta is the angle at which r has to be measured from origin. Any mathematical function in the Cartesian coordinate system can also be plotted using the polar coordinates.

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

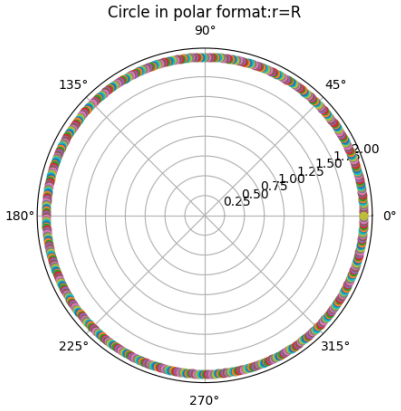
Output

```
plt.axes(projection = 'polar')
# setting the radius
r = 2
rads = np.arange(0, (2 * np.pi), 0.01)
# plotting the circle
for rad in rads:
    plt.polar(rad, r, 'g.')
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plot
plot.axes(projection='polar')
plot.title('Circle in polar format')
rads = np.arange(0, (2*np.pi), 0.01)
for radian in rads:
    plot.polar(radian,2,'o')
plot.show()
```

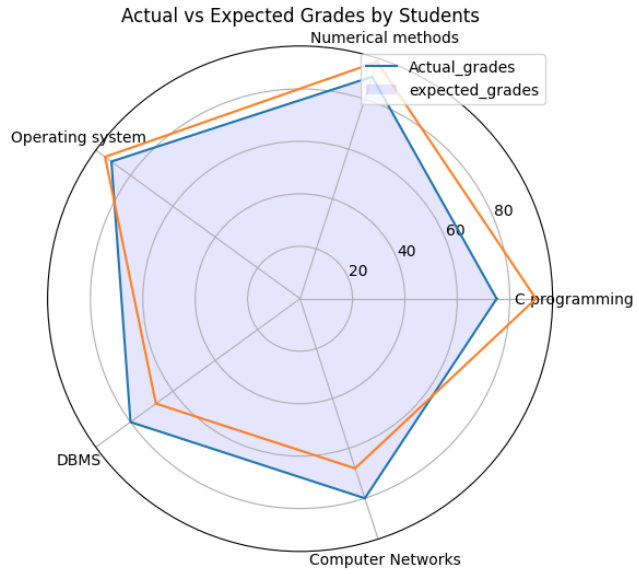
Output




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

subjects = ["C programming", "Numerical methods", "Operating
system",
            "DBMS", "Computer Networks"]

actual_grades = [75, 89, 89, 80, 80, 75]
expected_grades = [90, 95, 92, 68, 68, 90]

# Initializing the spiderplot by
# setting figure size and polar
# projection
plt.figure(figsize =(10, 6))
plt.subplot(polar = True)

theta = np.linspace(0, 2 * np.pi, len(actual_grades))

# Arranging the grid into number
# of sales into equal parts in
# degrees
lines,       labels       =       plt.thetagrids(range(0,       360,
int(360/len(subjects))),
                                                    (subjects))

# Plot actual sales graph
plt.plot(theta, actual_grades)
```

```
plt.fill(theta, actual_grades, 'b', alpha = 0.1)

# Plot expected sales graph
plt.plot(theta, expected_grades)

# Add legend and title for the plot
plt.legend(labels =('Actual_grades', 'expected_grades'),
           loc = 1)
plt.title("Actual vs Expected Grades by Students")

# Display the plot on the screen
plt.show()
```

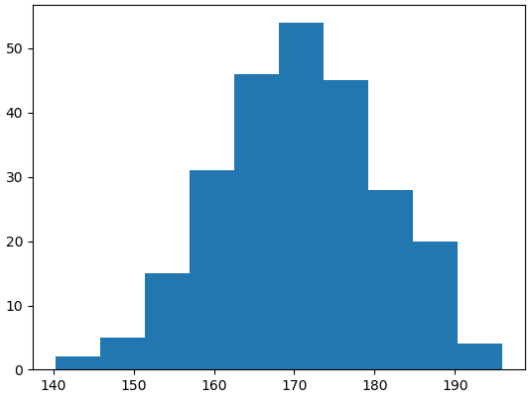
Output



Histogram

A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:

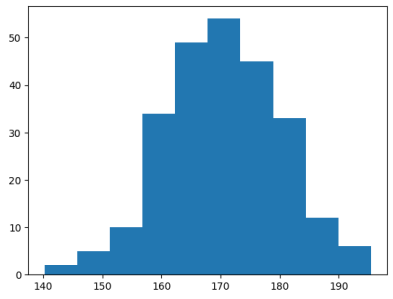


You can read from the histogram that there are approximately:

2 people from 140 to 145cm, 5 people from 145 to 150cm, 15 people from 151 to 156cm, 31 people from 157 to 162cm, 46 people from 163 to 168cm, 53 people from 168 to 173cm, 45 people from 173 to 178cm, 28 people from 179 to 184cm, 21 people from 185 to 190cm, 4 people from 190 to 195cm

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

x = np.random.normal(170, 10, 250)

plt.hist(x)
plt.show()
```

Output

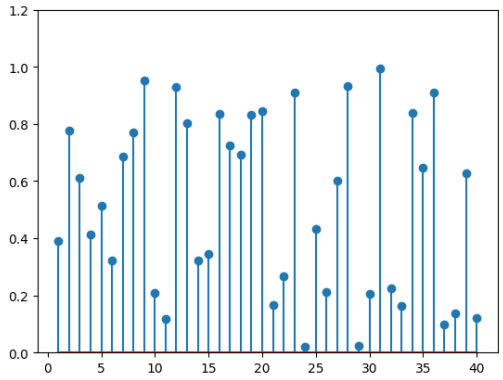


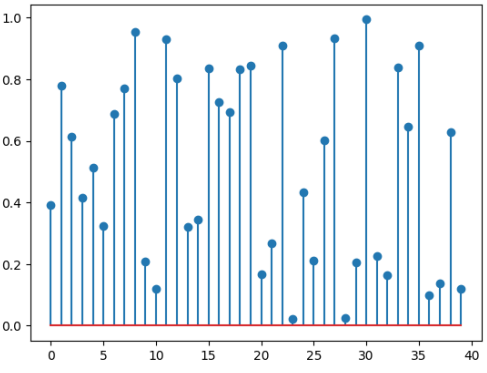
Lollipop plot

A basic lollipop plot can be created using the stem() function of matplotlib. This function takes x axis and y axis values as an argument. x values are optional; if you do not provide x values, it will automatically assign x positions.

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

# create data
x=range(1,41)
values=np.random.uniform(size=40)

# stem function
plt.stem(x, values)
plt.ylim(0, 1.2)
plt.show()

# stem function: If x is not provided, a sequence of numbers is created by python:
plt.stem(values)
plt.show()
```

Output





Data transformation

Data transformation is a set of techniques used to convert data from one format or structure to another format or structure. The following are some examples of transformation activities:

- Data deduplication involves the identification of duplicates and their removal.
- Key restructuring involves transforming any keys with built-in meanings to the generic keys.
- Data cleansing involves extracting words and deleting out-of-date, inaccurate, and incomplete information from the source language without extracting the meaning or information to enhance the accuracy of the source data.
- Data validation is a process of formulating rules or algorithms that help in validating different types of data against some known issues.
- Format revisioning involves converting from one format to another.
- Data derivation consists of creating a set of rules to generate more information from the data source.
- Data aggregation involves searching, extracting, summarizing, and preserving important information in different types of reporting systems.
- Data integration involves converting different data types and merging them into a common structure or schema.
- Data filtering involves identifying information relevant to any particular user.
- Data joining involves establishing a relationship between two or more tables.

Merging database-style dataframes

Combining Data in Pandas with append(), merge(), join(), and concat()

pandas concat(): Combining Data Across Rows or Columns

Concatenation is a bit different from the merging techniques that you saw above. With merging, you can expect the resulting dataset to have rows from the parent datasets mixed in together, often based on some commonality.

With concatenation, your datasets are just stitched together along an axis — either the row axis or column axis.

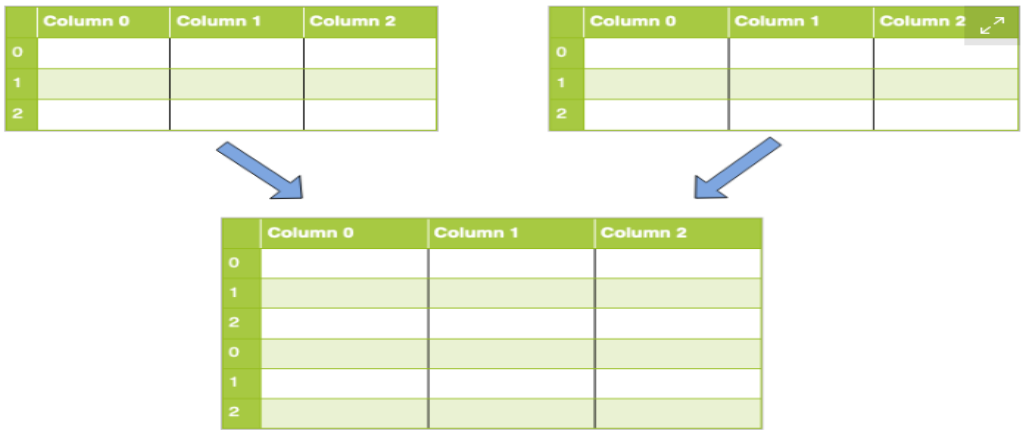
```
#Importing libraries
import pandas as pd
import numpy as np

df1 = pd.DataFrame(
    {
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    },
```

Output:

```
    index=[0, 1, 2, 3],
)
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
    },
    index=[4, 5, 6, 7],
)
df3 = pd.DataFrame(
    {
        "A": ["A8", "A9", "A10", "A11"],
        "B": ["B8", "B9", "B10", "B11"],
        "C": ["C8", "C9", "C10", "C11"],
        "D": ["D8", "D9", "D10", "D11"],
    },
    index=[8, 9, 10, 11],
)
frames = [df1, df2, df3]
result = pd.concat(frames)
print("\n", df1)
print("\n", df2)
print("\n", df3)
print("\n", result)
```

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3

    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7

     A    B    C    D
8   A8   B8   C8   D8
9   A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11


     A    B    C    D
0   A0   B0   C0   D0
1   A1   B1   C1   D1
2   A2   B2   C2   D2
3   A3   B3   C3   D3
4   A4   B4   C4   D4
5   A5   B5   C5   D5
6   A6   B6   C6   D6
7   A7   B7   C7   D7
8   A8   B8   C8   D8
9   A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11
```

Visually, a concatenation with no parameters along rows would look like this:




```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1"],
        "B": ["B0", "B1"],
        "C": ["C0", "C1"],
        "D": ["D0", "D1"],
    },index=[0, 1])
```

```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1"],
        "B": ["B0", "B1"],
        "C": ["C0", "C1"],
        "D": ["D0", "D1"],
    },index=[0, 1])
```

```
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5"],
        "B": ["B4", "B5"],
        "C": ["C4", "C5"],
        "D": ["D4", "D5"],
    },index=[0, 1])
frames = [df1, df2]
result = pd.concat(frames)
print("\n", df1)
print("\n", df2)
print("\n", result)
```

Output:

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

     A   B   C   D
0  A4  B4  C4  D4
1  A5  B5  C5  D5

     A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
0  A4  B4  C4  D4
1  A5  B5  C5  D5
```

result = pd.concat((frames), axis = "rows")

```
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5"],
        "B": ["B4", "B5"],
        "C": ["C4", "C5"],
        "D": ["D4", "D5"],
    },index=[0, 1])
frames = [df1, df2]
result = pd.concat((frames),
                   axis = "columns")
print("\n", df1)
print("\n", df2)
print("\n", result)
```

Output:

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

     A   B   C   D
0  A4  B4  C4  D4
1  A5  B5  C5  D5

     A   B   C   D   A   B   C   D
0  A0  B0  C0  D0  A4  B4  C4  D4
1  A1  B1  C1  D1  A5  B5  C5  D5
```

```
result = pd.concat(frames, keys=["x", "y"])
```

As you can see, the resulting object's index has a hierarchical index.

Output:

```
       A   B   C   D
x 0  A0  B0  C0  D0
  1  A1  B1  C1  D1
y 0  A4  B4  C4  D4
  1  A5  B5  C5  D5
```

In Pandas for a horizontal combination we have merge() and join(), whereas for vertical combination we can use concat() and append(). Merge and join perform similar tasks but internally they have some differences, similar to concat and append.

pandas merge():

Pandas provides various built-in functions for easily combining datasets. Among them, merge() is a high-performance in-memory operation very similar to relational databases like SQL. You can use merge() any time when you want to do database-like join operations.

- The simplest call without any key column
- Specifying key columns using **on**
- Merging using left_on and right_on
- Various forms of joins: inner, left, right and outer

Syntax:

- # This join brings together the entire DataFame
  df.merge(df2)

- # This join only brings together a subset of columns
- # 'col1' is my key in this case

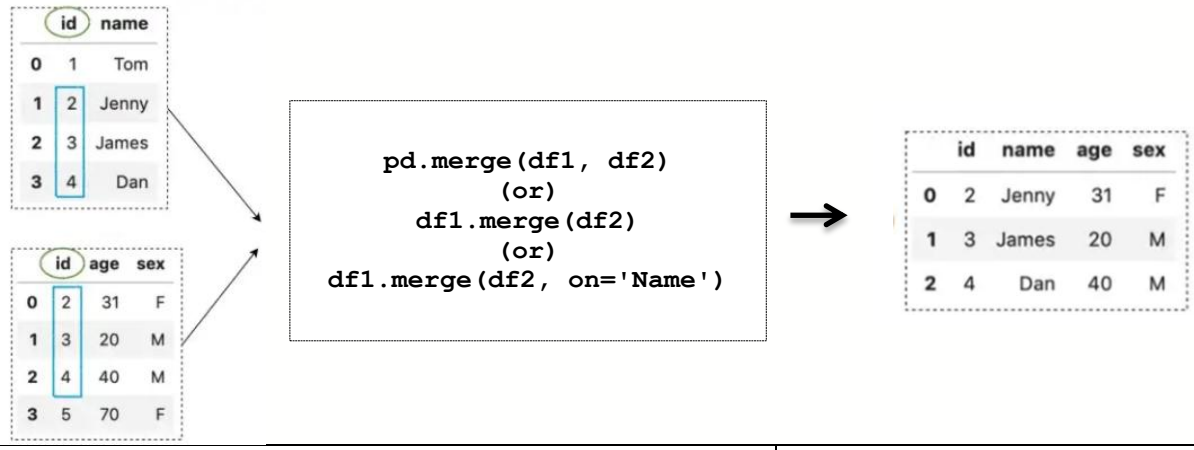
  df[['col1', 'col2']].merge(df2[['col1', 'col3']], on='col1')

Code 1#: Merging two DataFrames

```
df1 = pd.DataFrame({
    'id': [1, 2, 3, 4],
    'name': ['Tom', 'Jenny', 'James', '
Dan'],
})
df2 = pd.DataFrame({
    'id': [2, 3, 4, 5],
    'age': [31, 20, 40, 70],
    'sex': ['F', 'M', 'M', 'F']
})
print("\n",df1)
print("\n",df2)

final = pd.merge(df1, df2)
print("\n",final)
```

Output:

```
   id   name
0   1    Tom
1   2  Jenny
2   3  James
3   4    Dan

   id  age sex
0   2   31   F
1   3   20   M
2   4   40   M
3   5   70   F

   id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
```

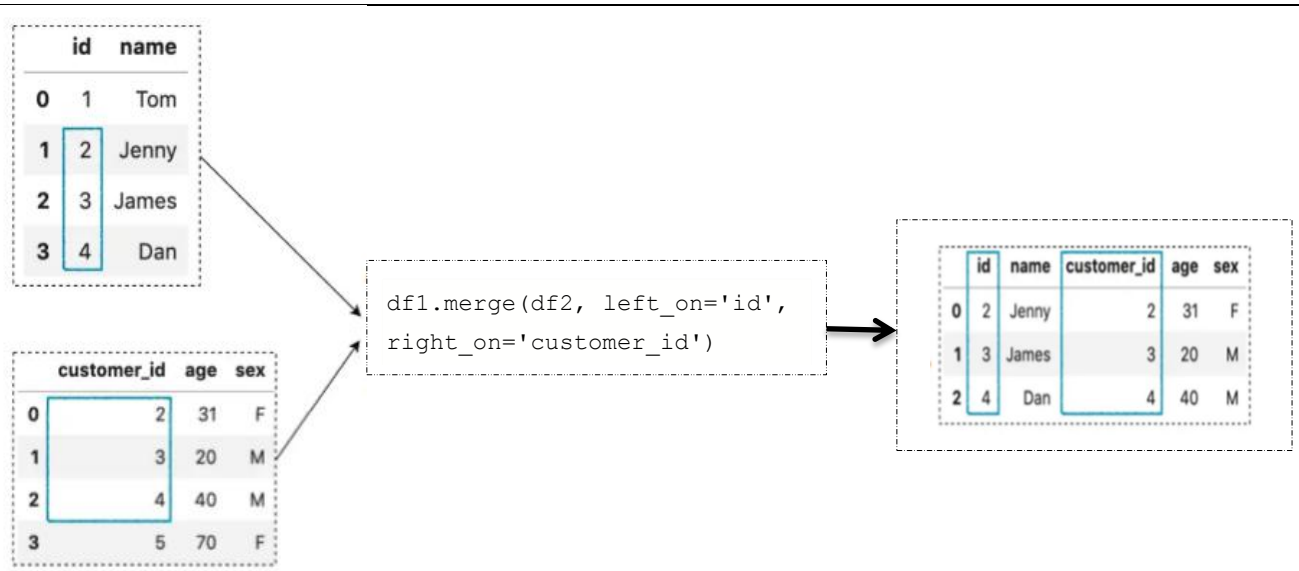


Code 2#: Merge two DataFrames via 'id' column.

```
final = df1.merge(df2, on='id')
print("\n",final)
```

Output:

```
   id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
```



Code 3#: Merge with different column names - specify a left_on and right_on

```
final = df1.merge(df2, left_on='id',
                  right_on='customer_id
')
```

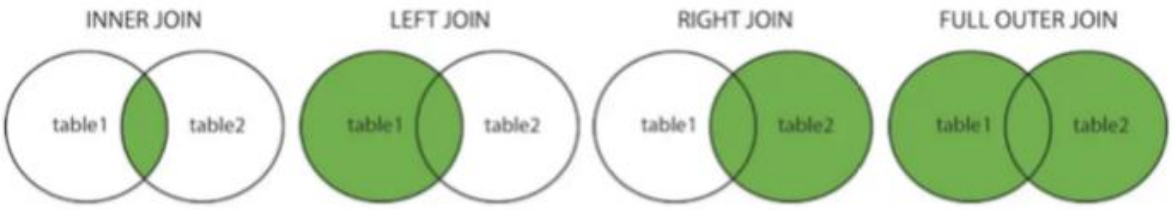
Output:

```
   id   name  customer_id  age sex
0   2  Jenny            2   31   F
1   3  James            3   20   M
2   4    Dan            4   40   M
```

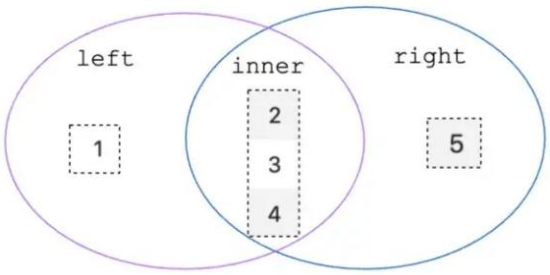
Various type of joins: inner, left, right and outer

They are 4 types of joins available to Pandas merge() function. The logic behind these joins is very much the same that you have in SQL when you join tables. You can perform a type of join by specifying the how argument with the following values:

- **inner:** Default join is inner in Pandas merge() function, and it produces records that have matching values in both DataFrames
- **left:** produces all records from the left DataFrame and the matched records from the right DataFrame
- **right:** produces all records from the right DataFrame and the matched records from the left DataFrame
- **outer:** produces all records when there is a match in either left or right DataFrame



```
pd.merge(df_customer, df_info, on='id', how=?)
```



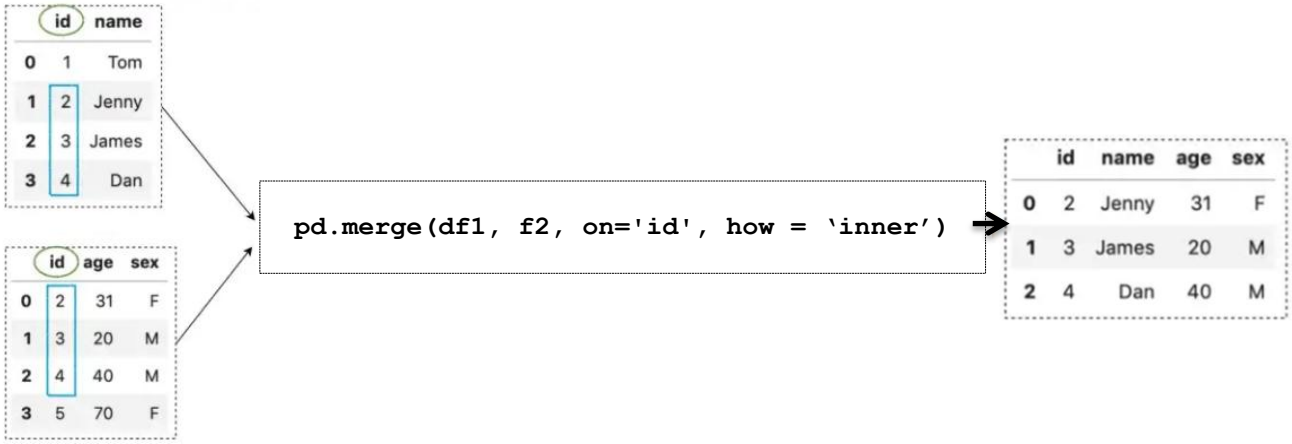
Code 4#: Merge using inner join

Pandas merge() is performing the inner join and it produces only the set of records that match in both DataFrame.

```
final = pd.merge(df1, df2, on='id',
                 how = 'inner')
```

Output:

```
   id    name   age sex
0   2   Jenny    31   F
1   3   James    20   M
2   4     Dan    40   M
```



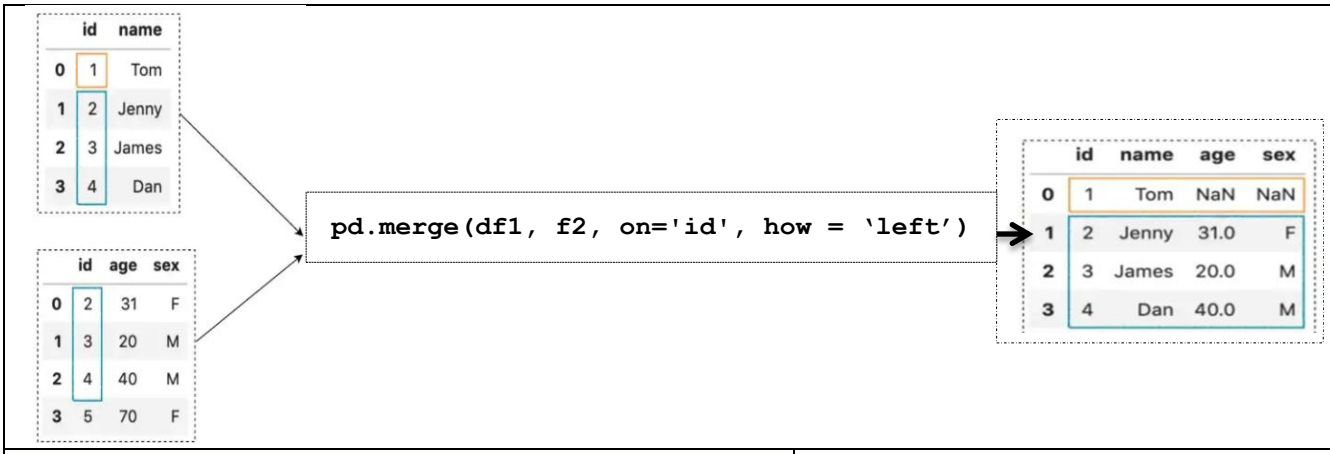
Code 4#: Merge using Left join

The left join produces all records from the left DataFrame, and the matched records from the right DataFrame. If there is no match, the left side will contain NaN.

```
final = pd.merge(df1, df2, on='id',
                 how = 'left')
```

Output:

```
    id    name    age  sex
0    1     Tom    NaN  NaN
1    2   Jenny   31.0    F
2    3   James   20.0    M
3    4     Dan   40.0    M
```



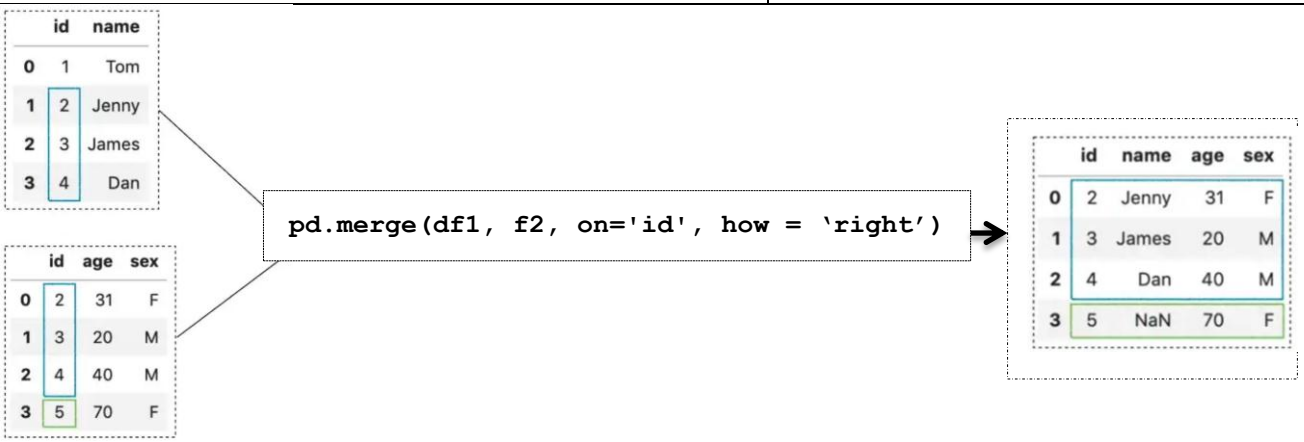
Code 4#: Merge using Right join

The right join produces all records from the right DataFrame, and the matched records from the left DataFrame. If there is no match, the right side will contain NaN.

```
final = pd.merge(df1, df2, on='id',
                 how = 'right')
```

Output:

```
   id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
3   5    NaN   70   F
```



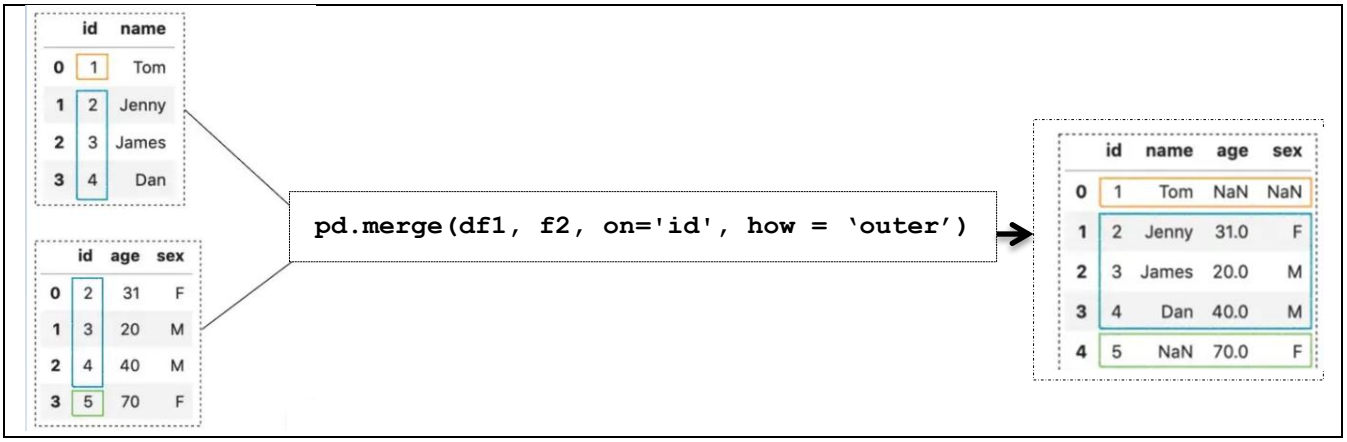
Code 4#: Merge using Outer join

The outer join produces all records when there is a match in either left or right DataFrame. NaN will be filled for no match on either sides.

```
final = pd.merge(df1, df2, on='id',
                 how = 'outer')
```

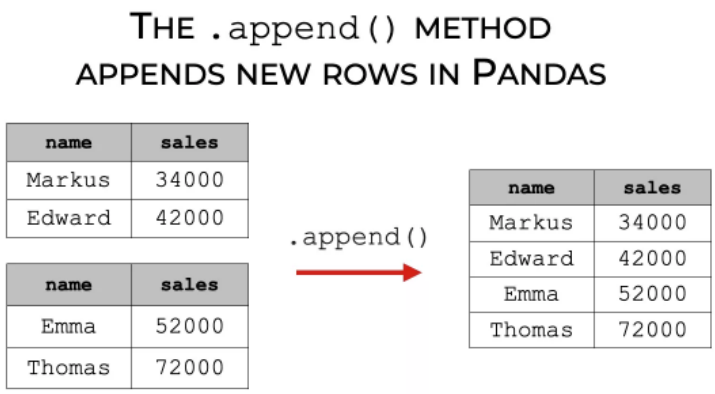
Output:

```
   id   name   age  sex
0   1    Tom   NaN  NaN
1   2  Jenny  31.0    F
2   3  James  20.0    M
3   4    Dan  40.0    M
4   5    NaN  70.0    F
```



pandas append():

To append the rows of one dataframe with the rows of another, we can use the Pandas append() function. With the help of append(), we can append columns too.



Steps

- Create a two-dimensional, size-mutable, potentially heterogeneous tabular data, df1.
- Print the input DataFrame, df1.
- Create another DataFrame, df2, with the same column names and print it.
- Use the append method, df1.append(df2, ignore_index=True), to append the rows of df2 with df2.
- Print the resultatnt DataFrame.

Code 5#: Append Function to join DataFrames

```
import pandas as pd
df1 = pd.DataFrame({"x": [5, 2],
                    "y": [4, 7],
                    "z": [1, 3]})
df2 = pd.DataFrame({"x": [1, 3],
                    "y": [1, 9],
                    "z": [1, 3]})
print ("\n", df1)
print ("\n", df2)
df3 = df1.append(df2)
print ("\n ", df3)
```

Output:

```
   x  y  z
0  5  4  1
1  2  7  3

   x  y  z
0  1  1  1
1  3  9  3

   x  y  z
0  5  4  1
1  2  7  3
0  1  1  1
1  3  9  3
```

df3 = df1.append(df2,ignore_index=True)	**Output:**
	`   x  y  z`
	`0  5  4  1`
	`1  2  7  3`
	`2  1  1  1`
	`3  3  9  3`

`import pandas as pd`
`df1 = pd.DataFrame({"x": [5, 2],`
`"y": [4, 7],`
`"z": [1, 3]})`
`df2 = pd.DataFrame({"a": [1, 3],`
`"b": [1, 9],`
`"c": [1, 3]})`
`print ("\n", df1)`
`print ("\n", df2)`
`df3 = df1.append(df2, ignore_index=True)`
`print ("\n ", df3)`
	Output:
	`   x  y  z`
	`0  5  4  1`
	`1  2  7  3`
	`   a  b  c`
	`0  1  1  1`
	`1  3  9  3`
	`     x    y    z    a    b    c`
	`0  5.0  4.0  1.0  NaN  NaN  NaN`
	`1  2.0  7.0  3.0  NaN  NaN  NaN`
	`2  NaN  NaN  NaN  1.0  1.0  1.0`
	`3  NaN  NaN  NaN  3.0  9.0  3.0`

Reshaping and pivoting

pivot() function

- Return reshaped DataFrame organized by given index / column values.
- Reshape data (produce a "pivot" table) based on column values.
- Uses unique values from specified index / columns to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns.
-

```
import numpy as np
import pandas as pd
df = pd.DataFrame({'s1': ['one', 'one', 'one', 'two', 'two','two'],
                   's2': ['P', 'Q', 'R', 'P', 'Q', 'R'],
                   's3': [2, 3, 4, 5, 6, 7],
                   's4': ['x', 'y', 'z', 'q', 'w', 't']})
print(df)
```

Output

```
    s1 s2  s3 s4
0  one  P   2  x
1  one  Q   3  y
2  one  R   4  z
3  two  P   5  q
4  two  Q   6  w
5  two  R   7  t
```

```
df.pivot(index='s1', columns='s2', values='s3')
```

Output

s2	P	Q	R
s1			
one	2	3	4
two	5	6	7

```
df.pivot(index='s1', columns='s2', values=['s3', 's4'])
```

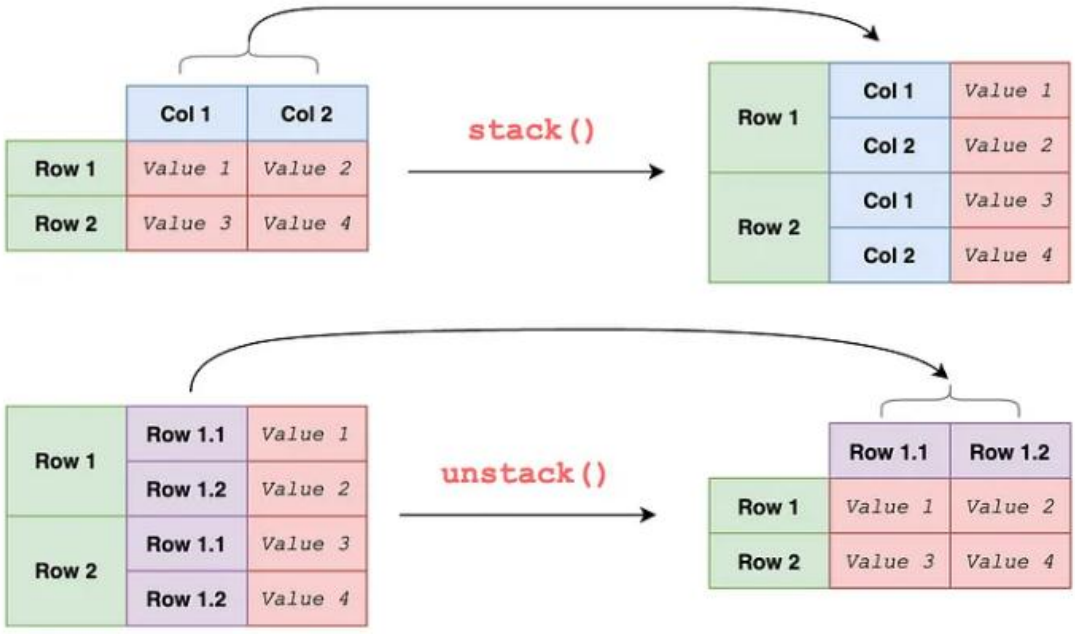
Output

	s3			s4		
s2	P	Q	R	P	Q	R
s1						
one	2	3	4	x	y	z
two	5	6	7	q	w	t

Stack() and unstack()

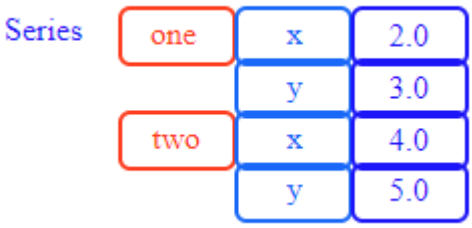
During EDA, we often need to rearrange data in a dataframe in some consistent manner. This can be done with hierarchical indexing using two actions:

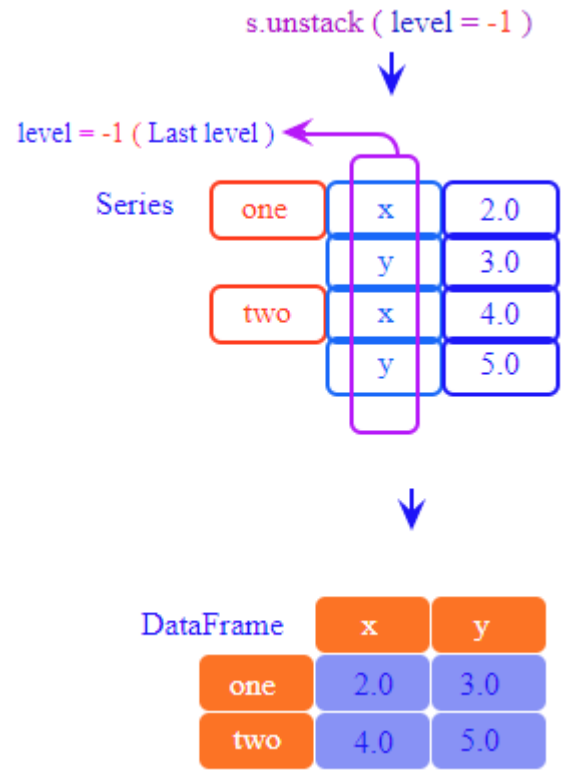
- Stacking: Stack rotates from any particular column in the data to the rows.
- Unstacking: Unstack rotates from the rows into the column

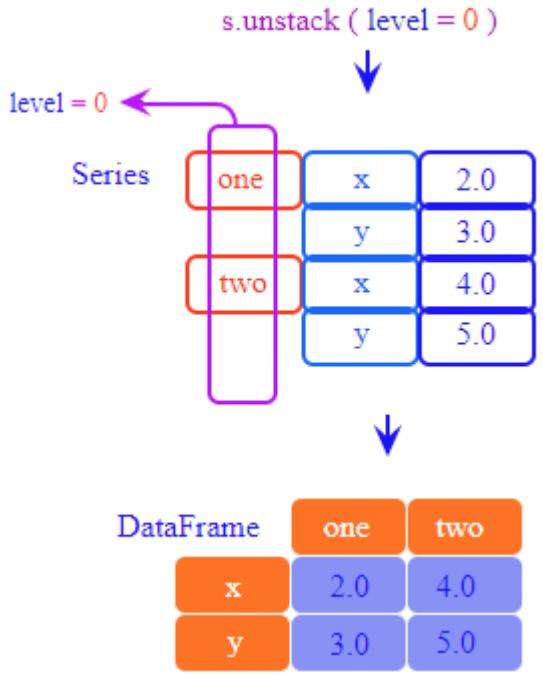


```
import numpy as np
import pandas as pd
index = pd.MultiIndex.from_tuples([('one', 'x'), ('one', 'y'),
                                   ('two', 'x'), ('two','y')])
s = pd.Series(np.arange(2.0, 6.0), index=index)
print(s)
```

Output








```
df = s.unstack(level=0)
df.unstack()
```

output

```
one  x    2.0
     y    3.0
two  x    4.0
     y    5.0
dtype: float64
```

Transformation.
While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input.

`df.sum()`	`key ABCABC`
	`data 15`
	`dtype: object`

`df.mean()`	`data 2.5`

`df.groupby('key').transform(lambda x: x - x.mean())`	`data`
	`0 -1.5`
	`1 -1.5`
	`2 -1.5`
	`3 1.5`
	`4 1.5`
	`5 1.5`