**Introduction to array**

An array is a collection of elements of the same data type, arranged in a contiguous block of memory. In programming, arrays are used to store and manipulate large sets of data efficiently. They can be one-dimensional, two-dimensional or multi-dimensional depending on the requirements of the problem.

Arrays are commonly used in scientific computing, data analysis, and machine learning for storing and manipulating large sets of data. For example, an array can be used to store a set of temperature readings, a set of pixel values for an image, or a set of numerical values for a mathematical function.

One of the advantages of using arrays is that they allow for fast and efficient access to elements. Elements in an array can be accessed using their index, which is an integer value that represents the position of the element in the array. Arrays also support operations such as sorting, filtering, and transforming data, which can be useful in a variety of applications.

In Python, arrays are implemented using the NumPy package, which provides support for multi-dimensional arrays and a large library of mathematical functions and operations. NumPy arrays are efficient and optimized for scientific computing and data analysis, making them a popular choice for many applications.

**Introduction to NumPy**

NumPy stands for "Numerical Python" and is a Python package that provides support for large, multi-dimensional arrays and matrices, along with a large library of mathematical functions to operate on these arrays. NumPy is a fundamental package for scientific computing with Python

It is built on top of the C programming language, which allows for faster computations and better memory management.

One of the main advantages of NumPy is its ability to perform mathematical operations on arrays of data quickly and efficiently. This is due to its use of vectorization, which allows for operations to be applied to entire arrays at once, rather than iterating through each element individually.

Some of the key features of NumPy include:

* Support for multi-dimensional arrays and matrices
* A large library of mathematical functions and operations
* Fast and efficient computation due to vectorization
* Integration with other scientific computing packages such as SciPy and Pandas.

**Numerical Operations in NumPy**

Some of the common numerical operations in NumPy include:

1. Mathematical Functions: NumPy provides a large library of mathematical functions that can be applied to arrays. Some examples include sin, cos, tan, log, exp, sqrt, and many more.
2. Basic Arithmetic: NumPy arrays support basic arithmetic operations such as addition, subtraction, multiplication, and division. These operations can be performed on arrays of the same shape or on arrays with compatible shapes.
3. Linear Algebra: NumPy provides a suite of linear algebra functions for performing matrix operations such as matrix multiplication, matrix inversion, and eigenvalue calculations.
4. Statistical Functions: NumPy provides a range of statistical functions for analyzing data, including mean, median, standard deviation, variance, and more.
5. Sorting and Searching: NumPy provides functions for sorting and searching arrays, including sorting by value, sorting by index, and searching for specific values in an array.
6. Broadcasting: NumPy provides a powerful feature called broadcasting, which allows operations to be performed on arrays of different shapes and sizes. This feature can save time and memory by avoiding the need to create additional copies of data.

**Overview of Pandas**

Pandas is a Python library for data manipulation and analysis. It provides a set of powerful data structures and functions for working with structured data, including tabular data such as spreadsheets and SQL tables. Pandas is built on top of the NumPy library, and integrates well with other scientific computing libraries in the Python ecosystem.

The key data structures provided by Pandas are the Series and DataFrame objects. A Series is a one-dimensional labeled array, while a DataFrame is a two-dimensional table-like data structure, with columns of potentially different types. Pandas also provides a range of functions for manipulating and transforming data, including filtering, merging, grouping, and pivoting operations.

Pandas is widely used in data science and data analysis, and is often used in conjunction with other Python libraries such as NumPy, Matplotlib, and Scikit-learn. Some common use cases for Pandas include data cleaning, data wrangling, exploratory data analysis, and data visualization.

**Forms in Pandas**

In Pandas, there are two main types of data structures for storing and manipulating data: Series and DataFrame.

A Series is a one-dimensional array-like object that can hold any data type, including integer, float, string, and object. Each element in a Series has an associated index label, which can be used to access individual elements.

A DataFrame is a two-dimensional table-like data structure, consisting of rows and columns, similar to a spreadsheet or a SQL table. Each column in a DataFrame is a Series object, and the columns are aligned by their index labels. The DataFrame has row and column index labels, which can be used to access individual elements or subsets of the data.

In addition to these two main data structures, Pandas also provides other data structures, such as Panel (a three-dimensional array-like data structure) and Panel4D (a four-dimensional array-like data structure).

Overall, Pandas provides a flexible and powerful toolkit for working with different types of data structures and performing data manipulation tasks, such as filtering, merging, grouping, and reshaping data.

**Pandas function**

Pandas provides a wide range of functions for manipulating and analyzing data. Here are some of the most commonly used functions:

1. **read\_csv()**: Reads a CSV file into a DataFrame.
2. **head()**: Returns the first n rows of a DataFrame.
3. **tail()**: Returns the last n rows of a DataFrame.
4. **info()**: Provides information about a DataFrame, including column names, data types, and non-null values.
5. **describe()**: Computes summary statistics for each column in a DataFrame.
6. **shape**: Returns the number of rows and columns in a DataFrame.
7. **isnull()**: Returns a boolean mask indicating which values in a DataFrame are null.
8. **fillna()**: Fills null values in a DataFrame with a specified value.
9. **groupby()**: Groups a DataFrame by one or more columns and performs an aggregation function on each group.
10. **pivot\_table()**: Creates a pivot table from a DataFrame, aggregating data according to specified rows and columns.

These are just a few of the many functions provided by Pandas. Depending on the task at hand, there are many more functions available for data manipulation and analysis.

**Dataframe basics**

In Pandas, a DataFrame is a two-dimensional table-like data structure, consisting of rows and columns. Each column in a DataFrame is a Series object, and the columns are aligned by their index labels. The DataFrame has row and column index labels, which can be used to access individual elements or subsets of the data.

Here are some basic operations and functions for working with DataFrames in Pandas:

1. Creating a DataFrame: You can create a DataFrame by passing a dictionary or a list of lists to the **pd.DataFrame()** function.
2. Accessing rows and columns: You can access rows and columns of a DataFrame using the **.loc[]** and **.iloc[]** indexers. The **.loc[]** indexer is used to select rows and columns by their labels, while the **.iloc[]** indexer is used to select rows and columns by their integer positions.
3. Viewing data: You can use the **.head()** and **.tail()** methods to view the first or last few rows of a DataFrame, and the **.info()** method to display information about the DataFrame, such as column names, data types, and non-null values.
4. Filtering data: You can filter a DataFrame using boolean indexing, by passing a boolean condition to the DataFrame's indexing operator **[]**.
5. Adding and removing columns: You can add a new column to a DataFrame by assigning a Series to a new column name. You can remove a column using the **.drop()** method.
6. Aggregating data: You can aggregate data using methods such as **.mean()**, **.sum()**, **.min()**, **.max()**, and **.count()**. You can also group data using the **.groupby()** method and apply aggregation functions to each group.
7. Merging and joining data: You can merge or join two or more DataFrames using methods such as **.merge()**, **.concat()**, and **.join()**.

These are just a few of the many operations and functions available for working with DataFrames in Pandas. Depending on the task at hand, there are many more functions available for data manipulation and analysis.

**Key operations in dataframe and series**

Here are some key operations in Pandas for working with DataFrames and Series:

Operations on DataFrames:

1. Indexing and selecting data: You can select rows and columns using various methods like **loc**, **iloc**, **at**, and **iat**. You can also select columns by their name, using the indexing operator **[]**.
2. Filtering data: You can filter a DataFrame using boolean indexing, by passing a boolean condition to the DataFrame's indexing operator **[]**.
3. Handling missing values: You can handle missing values using methods such as **dropna**, **fillna**, and **interpolate**.
4. Aggregating data: You can aggregate data using methods such as **mean**, **sum**, **min**, **max**, and **count**. You can also group data using the **groupby** method and apply aggregation functions to each group.
5. Merging and joining data: You can merge or join two or more DataFrames using methods such as **merge**, **concat**, and **join**.
6. Reshaping data: You can reshape a DataFrame using methods such as **stack**, **unstack**, **pivot**, and **melt**.

Operations on Series:

1. Indexing and selecting data: You can select elements of a Series using various methods like **loc**, **iloc**, **at**, and **iat**. You can also select elements by their label or position, using the indexing operator **[]**.
2. Filtering data: You can filter a Series using boolean indexing, by passing a boolean condition to the Series's indexing operator **[]**.
3. Handling missing values: You can handle missing values using methods such as **dropna**, **fillna**, and **interpolate**.
4. Aggregating data: You can aggregate data using methods such as **mean**, **sum**, **min**, **max**, and **count**.
5. Applying functions: You can apply a function to each element of a Series using the **apply** method.
6. Sorting data: You can sort a Series using methods such as **sort\_values** and **sort\_index**.

These are just a few of the many operations available for working with DataFrames and Series in Pandas. Depending on the task at hand, there are many more functions available for data manipulation and analysis.

**Data analysis in Pandas**

Pandas is a popular tool for data analysis and provides a range of functions for data manipulation and analysis. Here are some key steps involved in data analysis with Pandas:

1. Data cleaning and preparation: Before analyzing data, it is often necessary to clean and prepare the data. This may involve handling missing values, removing duplicates, transforming data types, and merging or joining multiple data sources.
2. Exploratory data analysis (EDA): EDA involves exploring the data to gain insights and identify patterns. This may involve computing summary statistics, visualizing the data using plots and charts, and identifying outliers or anomalies in the data.
3. Data aggregation and grouping: Aggregating and grouping data involves computing summary statistics for groups of data. This can be done using the **groupby()** method in Pandas, which allows you to group data by one or more columns and apply summary functions such as **mean()**, **sum()**, **min()**, and **max()** to the groups.
4. Data visualization: Visualizing data is an important part of data analysis, as it allows you to communicate insights and patterns to others. Pandas provides a range of visualization functions, including bar plots, line plots, scatter plots, histograms, and box plots.
5. Statistical analysis: Pandas also provides functions for statistical analysis, such as hypothesis testing, correlation analysis, and regression analysis. These functions can be used to test hypotheses, identify relationships between variables, and make predictions based on the data.
6. Machine learning: Pandas can also be used in conjunction with machine learning libraries such as scikit-learn and TensorFlow to build and train machine learning models. This involves preparing the data, selecting appropriate features, and training the model using the data.

These are just some of the key steps involved in data analysis with Pandas. Depending on the task at hand, there are many more functions and techniques available for data manipulation and analysis.

**Data Wrangling**

**Data cleaning**

Data cleaning is an essential step in data wrangling, which is the process of transforming and preparing raw data for analysis. Data cleaning involves identifying and correcting errors, inconsistencies, and inaccuracies in the data to ensure that it is accurate, complete, and consistent.

Some common tasks involved in data cleaning include removing duplicate data, filling in missing values, correcting data formats, standardizing data, and handling outliers. The goal of data cleaning is to ensure that the data is of high quality and can be used for analysis without introducing biases or errors.

Data cleaning is a crucial step in data wrangling because it sets the foundation for accurate and reliable analysis. If data is not cleaned properly, it can lead to incorrect conclusions, misleading insights, and flawed decisions. Therefore, it is important to invest time and resources into data cleaning to ensure that the data is of high quality and can be used effectively for analysis.

**Missing values handling**

Missing values are a common problem in data wrangling, and handling them is an essential part of the data cleaning process. There are several approaches to handling missing values in data, including:

1. Deleting missing values: In some cases, it may be appropriate to simply delete any rows or columns that contain missing values. This approach is typically used when the missing values are relatively few and do not affect the overall structure of the data.
2. Imputing missing values: Imputation involves filling in missing values with estimated or predicted values. Common imputation techniques include mean imputation, mode imputation, and regression imputation. The choice of imputation technique will depend on the type of data and the nature of the missing values.
3. Creating a separate category for missing values: In some cases, it may be appropriate to create a separate category for missing values, rather than deleting them or imputing them. This approach is often used when the missing values represent a meaningful category or when deleting or imputing them would introduce bias.
4. Using machine learning algorithms: Machine learning algorithms can be used to predict missing values based on the available data. This approach is particularly useful when the missing values are numerous and the data is complex.

It is important to carefully consider the approach to handling missing values, as each approach has its advantages and disadvantages. The choice of approach will depend on the specific characteristics of the data and the goals of the analysis.

**Handling categorical & numerical features**

Handling categorical and numerical features is an important step in data wrangling, as different types of features require different data preparation techniques.

Categorical features represent data that can be divided into discrete categories, such as gender or color. There are several techniques for handling categorical features in data wrangling, including:

1. One-hot encoding: One-hot encoding involves creating a binary variable for each category of a categorical feature. For example, if a categorical feature is color and it has three categories (red, blue, and green), one-hot encoding would create three binary variables: is\_red, is\_blue, and is\_green.
2. Label encoding: Label encoding involves assigning each category of a categorical feature a unique integer value. This approach is useful when the categories have an inherent order or when the number of categories is large.
3. Frequency encoding: Frequency encoding involves replacing each category of a categorical feature with its frequency in the data. This approach is useful when the categories have similar importance or when there is a large number of categories.

Numerical features represent data that can be measured or quantified, such as age or income. There are several techniques for handling numerical features in data wrangling, including:

1. Standardization: Standardization involves scaling the values of a numerical feature to have a mean of 0 and a standard deviation of 1. This approach is useful when the numerical features have different scales.
2. Min-max scaling: Min-max scaling involves scaling the values of a numerical feature to have a minimum value of 0 and a maximum value of 1. This approach is useful when the numerical features have a clear minimum and maximum value.
3. Log transformation: Log transformation involves taking the logarithm of a numerical feature to reduce its range and make it more normally distributed. This approach is useful when the numerical features have a skewed distribution.

In summary, handling categorical and numerical features in data wrangling requires different techniques, and it is important to choose the appropriate techniques based on the characteristics of the data and the goals of the analysis.

**Outlier detection & imputation in data wrangling**

Outlier detection and imputation are important steps in data wrangling, as outliers can significantly affect the accuracy and reliability of statistical analyses.

Outliers are data points that are significantly different from the other data points in the dataset. They can be caused by errors in data collection, measurement errors, or simply represent extreme values of the data. Outliers can affect statistical analyses by biasing the results and reducing the accuracy of the predictions.

Outlier detection involves identifying and removing or handling outliers in the dataset. There are several techniques for outlier detection, including:

1. Z-score: The Z-score is a statistical measure that indicates how many standard deviations a data point is from the mean. Data points with a Z-score greater than a certain threshold (e.g., 3) are considered outliers.
2. Interquartile range (IQR): The IQR is a measure of the spread of the data that is less sensitive to extreme values than the standard deviation. Data points outside of the upper or lower IQR range are considered outliers.
3. Visualization: Visualization techniques, such as scatter plots and box plots, can be used to visually identify outliers in the data.

Once outliers have been identified, they can be removed, imputed, or handled in some other way. Outlier imputation involves replacing the outlier values with estimated or predicted values. There are several techniques for outlier imputation, including:

1. Winsorizing: Winsorizing involves replacing extreme values with the nearest value within a certain range. For example, the top 5% of values could be replaced with the value at the 95th percentile.
2. Median imputation: Median imputation involves replacing outlier values with the median value of the dataset. This approach is less sensitive to outliers than mean imputation.
3. Machine learning algorithms: Machine learning algorithms can be used to predict outlier values based on the available data. This approach is particularly useful when the outlier values are numerous and the data is complex.

In summary, outlier detection and imputation are important steps in data wrangling, and several techniques can be used to identify and handle outliers in the data. The choice of technique will depend on the specific characteristics of the data and the goals of the analysis.

**Exploratory data analysis in data wrangling**

Exploratory Data Analysis (EDA) is a crucial step in data wrangling. It involves analyzing and summarizing the main characteristics of the data to better understand its distribution, relationships, and any patterns or trends that might exist. EDA is important because it helps to identify errors, anomalies, or missing data, and it can also provide insights into potential biases or confounding factors that may be present in the data.

There are several techniques and tools that can be used to conduct EDA, including:

1. Summary statistics: Summary statistics such as mean, median, mode, variance, and standard deviation can provide a basic understanding of the distribution of the data.
2. Data visualization: Data visualization techniques such as histograms, scatterplots, box plots, and heat maps can help to identify patterns and relationships in the data. They can also be used to identify outliers or other anomalies.
3. Correlation analysis: Correlation analysis can help to identify relationships between different variables in the data. For example, it can help to determine whether there is a positive or negative correlation between two variables, or whether there is no correlation at all.
4. Hypothesis testing: Hypothesis testing can be used to test whether a certain hypothesis about the data is true or not. For example, it can be used to determine whether two groups of data are significantly different from each other, or whether a correlation between two variables is statistically significant.

The goals of EDA are to identify patterns and relationships in the data, to detect any potential errors or anomalies, and to develop hypotheses or questions that can be further explored using more advanced statistical analyses. The insights gained from EDA can help to guide the selection of appropriate statistical models and methods for the analysis, and they can also help to identify potential biases or confounding factors that need to be addressed.

**Decide suitable algorithms In data wrangling**

Selecting the appropriate algorithm for data wrangling depends on the type of data, the specific problem being addressed, and the intended outcome of the analysis. There are several data wrangling algorithms available that can be used to transform, clean, or preprocess data for analysis. Some common algorithms used in data wrangling include:

1. Principal Component Analysis (PCA): PCA is a technique used to reduce the dimensionality of a dataset by identifying the most important features that capture the majority of the variance in the data. This can be useful when dealing with large datasets or when there are many features to consider.
2. k-means clustering: k-means clustering is a technique used to group similar data points into clusters based on their similarity. This can be useful for identifying patterns or trends in the data or for identifying potential outliers.
3. Decision Trees: Decision trees are a type of algorithm used in data wrangling to classify data into different groups based on a set of criteria. They can be useful for identifying patterns or relationships in the data or for predicting future outcomes.
4. Regression Analysis: Regression analysis is a technique used to identify relationships between variables in the data. It can be useful for predicting outcomes or for identifying potential confounding factors that may affect the analysis.
5. Support Vector Machines (SVM): SVM is a type of algorithm used for classification and regression analysis. It can be useful for identifying patterns or trends in the data or for predicting future outcomes.

The selection of the most suitable algorithm for data wrangling will depend on the specific characteristics of the data and the research question being addressed. It is important to consider factors such as data type, size, and complexity, as well as the goals of the analysis when selecting an algorithm. Additionally, it is often helpful to experiment with different algorithms to determine which one works best for a particular dataset or problem.

**Sampling**

Sampling is a technique used in data wrangling to select a representative subset of data from a larger dataset for further analysis. There are several reasons why sampling may be necessary, including:

1. Time and resource constraints: Sampling may be necessary when it is not feasible to analyze the entire dataset due to time or resource constraints.
2. Statistical inference: Sampling can be used to make inferences about the larger population from which the sample was drawn. By selecting a representative sample, statistical inference can be used to estimate population parameters such as means, variances, or proportions.
3. Data exploration: Sampling can be useful for exploring patterns or relationships in the data without analyzing the entire dataset.

There are several types of sampling methods that can be used in data wrangling, including:

1. Simple random sampling: Simple random sampling involves selecting data points from the dataset at random, with each data point having an equal chance of being selected.
2. Stratified sampling: Stratified sampling involves dividing the dataset into strata or subgroups based on a particular characteristic, and then selecting a sample from each stratum based on its proportion to the overall population.
3. Cluster sampling: Cluster sampling involves dividing the dataset into clusters or groups and then randomly selecting clusters for analysis.
4. Systematic sampling: Systematic sampling involves selecting data points at regular intervals from the dataset.

The choice of sampling method will depend on the specific characteristics of the dataset and the goals of the analysis. It is important to ensure that the sample selected is representative of the population from which it was drawn, as biased sampling can lead to inaccurate results and conclusions. Additionally, it is important to consider the sample size, as larger samples generally provide more accurate estimates of population parameters but may require more resources for analysis.

Data Diagnostics using Sklearn - (Model Selection, train-test split, Confusion Matrix, Standard Scaler, Normalization, transformations.

Scikit-learn (or sklearn) is a popular machine learning library in Python that provides a wide range of tools for data diagnostics and preprocessing. Some of the common data diagnostic techniques available in sklearn include:

1. Model selection: sklearn provides several model selection techniques, including cross-validation and grid search, to help identify the best machine learning model for a given dataset.
2. Train-test split: sklearn provides a simple way to split a dataset into a training set and a test set, allowing for the evaluation of model performance on unseen data.
3. Confusion matrix: sklearn provides a confusion matrix function that can be used to evaluate the performance of classification models. The confusion matrix provides information about the number of true positives, true negatives, false positives, and false negatives.
4. Standard scaler: sklearn provides a StandardScaler function that can be used to standardize the features in a dataset, by scaling them to have a mean of 0 and a standard deviation of 1. This can be useful for models that are sensitive to the scale of the input features, such as SVM or k-NN.
5. Normalization: sklearn provides several normalization functions, including MinMaxScaler and RobustScaler, that can be used to scale features to a specific range or to be more robust to outliers.
6. Transformations: sklearn provides several transformation functions, such as PCA and kernel methods, that can be used to transform the data in various ways to improve model performance or reduce the dimensionality of the data.

Overall, sklearn provides a powerful set of tools for data diagnostics and preprocessing, making it an essential library for many machine learning tasks.

**Python data analysis**

**Introduction to matplot lib**

Matplotlib is a popular plotting library in Python that is used to create high-quality visualizations of data. It is open-source and provides a wide range of customizable plots including line plots, scatter plots, bar plots, histograms, and more.

Matplotlib was originally developed by John D. Hunter in 2003 as a tool to plot data for his neuroscience research. Since then, it has become one of the most widely used libraries for data visualization in Python.

The library is designed to work seamlessly with NumPy arrays and can be used in combination with other Python libraries such as Pandas and Seaborn to create complex plots and visualizations.

Matplotlib provides a flexible and powerful API that allows users to customize every aspect of their plots including colors, markers, labels, legends, fonts, and more. Additionally, it provides a range of built-in styles and themes that can be used to quickly change the appearance of plots.

Overall, Matplotlib is an essential tool for data analysts, scientists, and engineers who need to create high-quality visualizations of their data. It is easy to use, versatile, and provides a wide range of customization options that can be used to create stunning visualizations for any kind of data.

Matplotlib provides a vast range of functions for creating different types of plots, here are some of the most commonly used ones with examples:

1. **plt.plot():** This function is used to create line plots.
2. **plt.scatter():** This function is used to create scatter plots.
3. **plt.bar():** This function is used to create bar plots.
4. **plt.boxplot():** This function is used to create boxplots.
5. **plt.imshow():** This function is used to create image plots.
6. **plt.pie():** This function is used to create pie charts.

**Introduction to Seaborn**

Seaborn is a Python data visualization library based on Matplotlib. It provides a higher-level interface for creating beautiful and informative statistical graphics. Seaborn is built on top of Matplotlib and adds some additional functionality and features to it.

Seaborn provides a set of powerful tools for data visualization and statistical analysis. It includes a wide range of built-in datasets for exploring and analyzing different types of data.

Some of the key features of Seaborn include:

1. Integration with Pandas: Seaborn is designed to work seamlessly with Pandas, a popular data manipulation library in Python. This makes it easy to visualize and analyze data stored in Pandas dataframes.
2. Beautiful default styles: Seaborn provides a set of beautiful default styles for plots, making it easy to create aesthetically pleasing visualizations with minimal effort.
3. Support for complex plots: Seaborn provides support for creating complex statistical visualizations such as heatmaps, cluster maps, and violin plots.
4. Flexible customization options: Seaborn provides a wide range of customization options for plots, including colors, fonts, and sizes.
5. Powerful statistical functions: Seaborn provides a range of statistical functions such as regression plots, density plots, and categorical plots that make it easy to analyze and visualize data.

Overall, Seaborn is a powerful data visualization library that makes it easy to create beautiful and informative visualizations of data in Python.

Seaborn provides a range of functions for creating different types of plots. Here are some of the most commonly used ones with examples:

1. **sns.lineplot():** This function is used to create line plots.
2. **sns.scatterplot():** This function is used to create scatter plots.
3. **sns.barplot():** This function is used to create bar plots.
4. **sns.histplot():** This function is used to create histograms.
5. **sns.boxplot():** This function is used to create boxplots.
6. **sns.heatmap():** This function is used to create heatmaps.
7. **sns.pairplot():** This function is used to create a matrix of scatter plots.

**Introduction to Iris dataset & 2D scatter plot**

The Iris flower dataset is a popular multivariate dataset that was introduced by the statistician and biologist Ronald Fisher in his 1936 paper "The use of multiple measurements in taxonomic problems". It contains measurements of four features of three species of Iris flowers (Iris setosa, Iris versicolor, and Iris virginica). The four features are sepal length, sepal width, petal length, and petal width, all measured in centimeters.

The dataset is often used for data analysis and machine learning tasks such as classification and clustering. It is also a popular dataset for learning data visualization techniques.

Here is an example of creating a 2D scatter plot of the Iris dataset using Seaborn:

This code loads the Iris dataset using the **sns.load\_dataset()** function, which returns a Pandas DataFrame. Then, it creates a 2D scatter plot using the **sns.scatterplot()** function, where **x** and **y** specify the columns to use for the x and y axes, **hue** specifies the column to use for coloring the points based on the species, and **data** specifies the DataFrame to use for plotting.

The resulting plot shows the relationship between the sepal length and width for each species of Iris flower, with each species represented by a different color:This plot can help us understand the relationship between the two features and how they differ between the three species of Iris flowers.

**3D scatter plot**

Here's an example of creating a 3D scatter plot of the Iris dataset using Matplotlib: This code first imports Matplotlib and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**. It then creates a 3D scatter plot using the **plt.scatter()** function, where **x**, **y**, and **z** specify the columns to use for the x, y, and z axes, respectively, and **c** specifies the colors to use based on the species. The **Axes3D** module is used to create a 3D projection for the plot, and the **set\_xlabel()**, **set\_ylabel()**, and **set\_zlabel()** functions are used to label the axes.

The resulting plot shows the relationship between the sepal length, sepal width, and petal length for each species of Iris flower, with each species represented by a different color:

This plot can help us understand the relationship between the three features and how they differ between the three species of Iris flowers in a 3D space.

**Pair plot**

Here's an example of creating a pair plot of the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**. It then creates a pair plot using the **sns.pairplot()** function, which plots all pairwise relationships between the features and colors the points based on the species.

The resulting plot shows scatter plots of each pairwise combination of features, with the diagonal showing histograms of each feature:

This plot can help us understand the relationships between each pair of features and how they differ between the three species of Iris flowers. It can also help us identify any outliers or patterns in the data.

**Limitations of pair plots**

Pair plots are a useful tool for visualizing the relationships between pairs of features in a dataset. However, they have some limitations:

1. Limited to pairwise comparisons: Pair plots are limited to showing pairwise relationships between features. They can't show the relationships between more than two features at a time.
2. Limited to continuous variables: Pair plots are best suited for visualizing the relationships between continuous variables. They may not be as effective for categorical or ordinal variables.
3. Computationally expensive: Pair plots can be computationally expensive for large datasets, as they generate multiple scatter plots and histograms.
4. Limited to low-dimensional data: Pair plots are most effective for low-dimensional datasets, where the number of features is relatively small. For high-dimensional datasets, other visualization techniques may be more effective.
5. Not suitable for non-linear relationships: Pair plots assume that the relationships between features are linear. They may not be as effective for visualizing non-linear relationships between features.
6. Limited to data distribution: Pair plots can only show the distribution of data. They cannot show the underlying statistical properties of the data, such as correlations or statistical significance.

In summary, while pair plots are a useful tool for visualizing relationships between pairs of features in a dataset, they have limitations when it comes to showing relationships between more than two features, non-linear relationships, and high-dimensional datasets. Other visualization techniques and statistical analysis may be needed to complement pair plots in these situations.

**Histogram example for above dataset**

Here's an example of creating a histogram of the sepal length for the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**. It then creates a histogram of the sepal length using the **sns.histplot()** function, with **data** specifying the dataset and **x** specifying the column to use for the x-axis.

The resulting plot shows the distribution of sepal lengths in the Iris dataset:

This plot can help us understand the distribution of sepal lengths in the Iris dataset, including the range of values, the central tendency, and the shape of the distribution.

**Introduction to probability function**

A probability function is a mathematical function that assigns a probability to each possible outcome of a random variable. In the context of the Iris dataset, we can use probability functions to compute the probability of different outcomes based on the features of the dataset.

For example, suppose we want to compute the probability of an Iris flower belonging to the "setosa" species based on its sepal length. We can do this using a probability function such as the Gaussian distribution, which models the probability of a continuous variable taking on a particular value.

Here's an example of computing the probability of an Iris flower belonging to the "setosa" species based on its sepal length using the Gaussian distribution in Python:

This code first imports NumPy and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**. It then computes the mean and standard deviation of sepal lengths for the setosa flowers using NumPy's **mean()** and **std()** functions.

Next, it defines a Gaussian probability function using the formula for the Gaussian distribution, which takes three parameters: **x**, the value to compute the probability for; **mu**, the mean of the distribution; and **sigma**, the standard deviation of the distribution.

Finally, it computes the probability of an Iris flower with sepal length 5 belonging to the setosa species using the Gaussian probability function. The result is a probability value between 0 and 1, which represents the likelihood of an Iris flower with sepal length 5 belonging to the setosa species based on the distribution of sepal lengths for setosa flowers in the Iris dataset.

Note that this example is just one way to use probability functions with the Iris dataset. There are many other types of probability functions that can be used, as well as different features and target variables that can be analyzed.

**Univariate analysis using probability function**

Univariate analysis refers to the analysis of a single variable in a dataset, without considering the relationships between variables. Probability functions can be used in univariate analysis to model the distribution of a single variable and compute various statistical measures such as mean, median, mode, variance, and standard deviation.

Here's an example of using probability functions for univariate analysis of the sepal length variable in the Iris dataset:

This code first imports NumPy and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**. It then computes the mean and standard deviation of sepal lengths using NumPy's **mean()** and **std()** functions.

Next, it defines a Gaussian probability function using the formula for the Gaussian distribution, which takes three parameters: **x**, the value to compute the probability for; **mu**, the mean of the distribution; and **sigma**, the standard deviation of the distribution.

Finally, it computes the probability density function (PDF) for sepal lengths using the Gaussian distribution and plots the result using Seaborn's **lineplot()** function. The PDF is a continuous function that represents the distribution of sepal lengths in the Iris dataset. It shows the probability of observing a sepal length within a certain range of values, with higher values indicating higher probabilities.

This example demonstrates how probability functions can be used in univariate analysis to model the distribution of a single variable and visualize the results. The same approach can be used for other variables in the Iris dataset, or for other datasets and probability functions.

**Cumulative distribution function**

The cumulative distribution function (CDF) is a function that maps a value to the probability that a random variable is less than or equal to that value. It provides a way to describe the overall distribution of a dataset, and can be used to compute various statistical measures such as quartiles, percentiles, and median.

Here's an example of using the CDF to analyze the sepal length variable in the Iris dataset:

This code first imports NumPy and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it computes the CDF for sepal lengths by first computing a histogram of the values using NumPy's **histogram()** function with **density=True** to obtain a normalized distribution. It then uses NumPy's **cumsum()** function to compute the cumulative sum of the histogram values, which gives the CDF.

Finally, it plots the CDF using Seaborn's **lineplot()** function, with the x-axis representing the bin edges of the histogram and the y-axis representing the cumulative probabilities.

The resulting plot shows how the sepal length values are distributed across the entire range of the variable, with the CDF increasing monotonically from 0 to 1. For example, we can use the CDF to determine that approximately 80% of the sepal length values are less than or equal to 6.9. We can also use the CDF to compute other statistical measures such as quartiles, percentiles, and median.

This example demonstrates how the CDF can be used to analyze the distribution of a variable in a dataset and compute various statistical measures. The same approach can be used for other variables in the Iris dataset, or for other datasets and probability distributions.

**Variance & standard deviation**

Variance and standard deviation are measures of the spread or dispersion of a dataset. The variance is the average of the squared differences from the mean, while the standard deviation is the square root of the variance.

Here's an example of computing the variance and standard deviation for the sepal length variable in the Iris dataset using NumPy:

This code first imports NumPy and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it computes the variance and standard deviation of sepal lengths using NumPy's **var()** and **std()** functions.

Finally, it prints the results to the console.

The output of this code will be:

This indicates that the sepal length variable in the Iris dataset has a variance of 0.68 and a standard deviation of 0.83. This means that the values are spread out around the mean by an average of 0.83 units.

This example demonstrates how variance and standard deviation can be used to quantify the spread of a dataset. The same approach can be used for other variables in the Iris dataset, or for other datasets and variables.

**Median, percentiles & quartiles introduction**

Median, percentiles, and quartiles are measures of the central tendency and spread of a dataset. The median is the value that separates the data into two halves, while percentiles and quartiles divide the data into equal parts.

Here's an example of computing the median, percentiles, and quartiles for the sepal length variable in the Iris dataset using NumPy:

This code first imports NumPy and Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it computes the median, percentiles, and quartiles of sepal lengths using NumPy's **median()**, **percentile()**, and **quantile()** functions.

Finally, it prints the results to the console.

The output of this code will be:

This indicates that the sepal length variable in the Iris dataset has a median value of 5.8. The 25th, 50th, and 75th percentiles are 5.1, 5.8, and 6.4, respectively. The same values are obtained when computing the 1st, 2nd, and 3rd quartiles.

These measures can be used to gain insights into the distribution of the data. For example, the median provides a measure of the central tendency, while the quartiles and percentiles provide a measure of the spread of the data. By looking at the quartiles and percentiles, we can see that the sepal length values are relatively evenly distributed between 5.1 and 6.4.

**Interquartile range boxplot with whiskers**

The interquartile range (IQR) is a measure of the spread of a dataset, based on the difference between the first and third quartiles. A boxplot is a graphical representation of the IQR, and it can be used to identify outliers and other features of the data distribution. Here's an example of creating an IQR boxplot with whiskers for the sepal length variable in the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it creates an IQR boxplot with whiskers for sepal lengths using Seaborn's **boxplot()** function. The **whis** parameter specifies the percentiles at which to draw the whiskers, and the **width** parameter specifies the width of the boxes.

Finally, it shows the plot using Seaborn's **plt.show()** function.

The output of this code will be a boxplot with whiskers, where the box represents the IQR and the whiskers extend to the minimum and maximum values within 5% and 95% percentiles, respectively.

This plot shows that the majority of the sepal length values are clustered around the median value of approximately 5.8. The box represents the IQR, which spans from approximately 5.1 to 6.4. The whiskers extend to the minimum and maximum values within 5% and 95% percentiles, respectively. There are no outliers in this variable.

This example demonstrates how boxplots can be used to visualize the spread of a dataset, and how the whiskers can be used to identify potential outliers. The same approach can be used for other variables in the Iris dataset, or for other datasets and variables.

**Violin plots, heatmaps & correlation plots introduction**

Violine Plots: A violin plot is a graphical representation of the distribution of data across different levels of a categorical variable. It is similar to a boxplot, but instead of just showing the quartiles and outliers, it also shows the density of the data at different levels of the categorical variable. Here's an example of creating a violin plot for the sepal length variable in the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it creates a violin plot for sepal lengths using Seaborn's **violinplot()** function. The **x** parameter specifies the categorical variable to group the data by (in this case, the species of the iris), and the **y** parameter specifies the variable to plot (in this case, the sepal length).

Finally, it shows the plot using Seaborn's **plt.show()** function.

The output of this code will be a violin plot for sepal lengths, grouped by species.

This plot shows that the setosa species has a shorter sepal length distribution, while the versicolor and virginica species have similar sepal length distributions with a slightly longer median value.

Heatmaps: A heatmap is a graphical representation of data that uses color-coding to represent different values. It is often used to visualize correlation matrices, where each cell of the matrix represents the correlation between two variables. Here's an example of creating a heatmap for the correlation matrix of the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it creates a correlation matrix using Pandas' **corr()** function, which calculates the pairwise correlation between all variables in the dataset.

Finally, it creates a heatmap for the correlation matrix using Seaborn's **heatmap()** function. The **annot** parameter specifies whether to annotate the heatmap with the correlation values.

The output of this code will be a heatmap for the correlation matrix of the Iris dataset.

This plot shows that the petal length and petal width variables are highly correlated (0.96), while the sepal length and sepal width variables are moderately correlated (-0.12). The diagonal of the heatmap shows the correlation of each variable with itself, which is always 1.

Correlation Plots: A correlation plot is a graphical representation of the pairwise correlations between different variables in a dataset. It is often used to identify which variables are most strongly correlated with each other, and which variables are most strongly correlated with the target variable. Here's an example of creating a correlation plot for the Iris dataset using Seaborn:

**Regression plots**

Regression Plots: A regression plot is a graphical representation of the relationship between two variables, where one variable is considered to be the dependent variable and the other variable is considered to be the independent variable. It is often used to visualize the relationship between a predictor variable and a response variable, and to fit a regression line to the data. Here's an example of creating a regression plot for the petal length and petal width variables in the Iris dataset using Seaborn:

This code first imports Seaborn, then loads the Iris dataset using **sns.load\_dataset()**.

Next, it creates a regression plot for the petal length and petal width variables using Seaborn's **regplot()** function. The **x** parameter specifies the predictor variable (in this case, petal length), the **y** parameter specifies the response variable (in this case, petal width), and the **data** parameter specifies the dataset.

Finally, it shows the plot using Seaborn's **plt.show()** function.

The output of this code will be a regression plot for the petal length and petal width variables.

This plot shows that there is a strong positive relationship between petal length and petal width, as indicated by the upward-sloping regression line.

**donut charts introduction**

To create a donut chart instead of a pie chart, we can add a white circle to the center of the chart using Matplotlib's **circle()** function:

This code is similar to the previous example, except that it creates a **fig** and **ax** object using Matplotlib's **subplots()** function, and adds a white circle to the center of the chart using Matplotlib's **Circle()** function.

The output of this code will be a donut chart for the species variable.

This chart shows the same information as the pie chart, but with a white circle in the center to create the donut shape.