**Python Libraries for Statistics**

**3.1 Introduction to NumPy:**

NumPy is a powerful and widely used Python library for numerical computing. The name "NumPy" stands for "Numerical Python." It provides support for large, multi-dimensional arrays and matrices, along with an extensive collection of mathematical functions to operate on these arrays efficiently. NumPy is the fundamental package for scientific computing in Python and forms the foundation for many other libraries in the scientific and data analysis ecosystem.

* **Syntax for installing NumPy in Python:**

To use NumPy in Python, you need to install it first. You can install it using **pip**, which is a package manager for Python. Open a terminal or command prompt and type the following command:

* **Code:**

****

* **Syntax for importing NumPy in Python:**

Once installed, you can import NumPy into your Python code using the following syntax:

* **Code:**



* **Here are some key reasons why NumPy is commonly used in data science:**

1. **Efficient Array Operations:** NumPy provides a powerful N-dimensional array object called **numpy. ndarray**. These arrays allow you to perform element-wise operations efficiently, which is crucial when dealing with large datasets. NumPy is implemented in C and optimized for performance, making it much faster than using Python's built-in data structures for numerical computations.
2. **Broadcasting:** NumPy enables broadcasting, which allows you to perform operations on arrays of different shapes and sizes without explicitly looping through the elements. This feature simplifies code and improves performance.
3. **Mathematical Functions:** NumPy provides a wide range of mathematical functions, including basic arithmetic operations, trigonometry, logarithms, exponentiation, and more. These functions are optimized for performance and can be applied element-wise to arrays.
4. **Random Number Generation:** NumPy includes a powerful random number generation module (numpy.random) that provides various methods for generating random data, which is essential for simulating and testing statistical models.
5. **Linear Algebra:** NumPy provides comprehensive support for linear algebra operations, such as matrix multiplication, eigendecomposition, singular value decomposition, and more. These functionalities are essential for a wide range of data science tasks, including machine learning algorithms.
6. **Indexing and Slicing:** NumPy arrays allow for flexible indexing and slicing, similar to Python lists. This enables efficient extraction of subsets of data and reshaping of arrays.
7. **Memory Efficiency:** NumPy arrays are more memory-efficient compared to standard Python lists, as they store elements of the same data type contiguously in memory.
8. **Integration with Other Libraries:** Many other data science libraries and frameworks, such as Pandas, SciPy, and scikit-learn, are built on top of NumPy. This means that NumPy arrays serve as a common data structure for exchanging data between different libraries.
9. **Interoperability:** NumPy arrays can be easily converted to and from other data structures, such as Python lists, Pandas DataFrames, and more. This makes it seamless to integrate NumPy into existing workflows.
10. **Performance:** Due to its optimized and efficient C-based implementation, NumPy can handle large datasets and complex calculations with much better performance compared to using pure Python.

* **Applications of NumPy:**

1. **Data analysis:** NumPy can be used to perform a variety of data analysis tasks, such as calculating statistics, plotting data, and fitting models.
2. **Machine learning:** NumPy is a powerful tool for machine learning. It can be used to create and train machine learning models, and to perform machine learning tasks such as classification and regression.
3. **Scientific computing:** NumPy is a powerful tool for scientific computing. It can be used to solve differential equations, perform numerical integration, and simulate physical systems.

* **Functions using NumPy:**

1. **Array creation functions in NumPy:**

* **Creating a 1D Array:**

This creates a one-dimensional NumPy array containing the values 1, 2, 3, and 4.

* **Code:**



* **Output:**

****

* **Creating a 2D Array:**

This creates a two-dimensional NumPy array with a single row containing the values 1, 2, 3, and 4.

* **Code:**



* **Output:**

****

1. **Array Type and Dimensions:**

The type function shows that two\_d\_array is of type numpy.ndarray, and the ndim attribute gives its dimensionality (2D).

* **Code:**

****

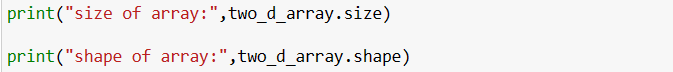
* **Output:**

****

1. **Array Size and Shape:**

The **size** attribute tells you the total number of elements in the array, and the **shape** attribute gives the dimensions of the array as a tuple.

* **Code:**

****

* **Output:**

****

1. **Creating an Array of Ones and Zeros:**

These functions generate arrays filled with ones or zeros, specified by the **shape** parameter, and using the specified data type (**int** in this case).

* **Code:**

****

* **Output:**

****

1. **Creating an Empty Array:**

**np.empty** creates an uninitialized array with the specified shape and data type. The values are not guaranteed to be zero or any specific value.

* **Code:**

****

* **Output:**

****

1. **Creating a Range of Values:**

**np.arange** generates an array of evenly spaced values within the specified range, here from 1 to 12.

* **Code:**

****

* **Output:**

****

1. **Creating a Linearly Spaced Array:**

**np.linspace** generates an array of linearly spaced values within the specified range and number of elements.

* **Syntax:**

**np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)**

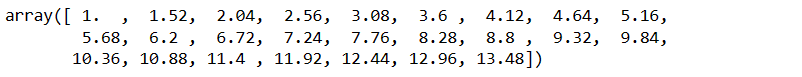
|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Default Value** |
| start | The starting value of the sequence. |  |
| stop | The end value of the sequence. |  |
| num | The number of samples to generate. Default is 50. | 50 |
| endpoint | Whether to include the stop value in the sequence. | True |
| retstep | Whether to return the step size between samples. | False |
| dtype | The data type of the output array. | Inferred |
| axis | The axis in the result to store the samples. | 0 |

* **Code:**

****

****

* **output:**

****

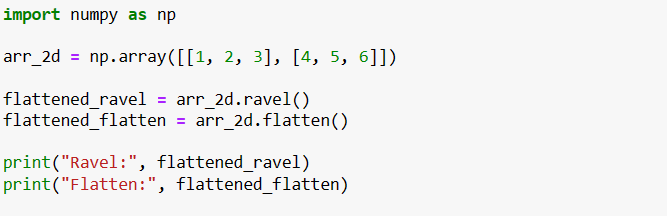
****

* **Array Manipulation Functions in NumPy:**

1. **Flattening an Array (ravel() and flatten()):**

Flattening an array means converting a multi-dimensional array into a one-dimensional array. NumPy offers two main functions to achieve this: ravel() and flatten().

* **Code:**

****

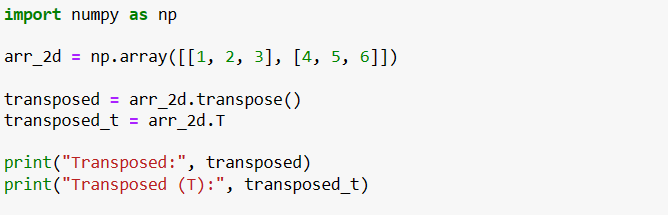
* **Output:**

****

1. **Transposing an Array (transpose() and .T):**

Transposing an array swaps its rows and columns. This is especially useful for manipulating arrays in linear algebra operations.

* **Code:**

****

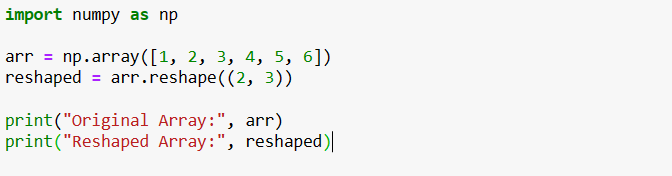
* **Output:**

****

1. **Reshaping an Array (reshape()):**

Reshaping an array involves changing its dimensions while preserving the order of elements. This is useful for converting arrays into different shapes.

* **Code:**

****

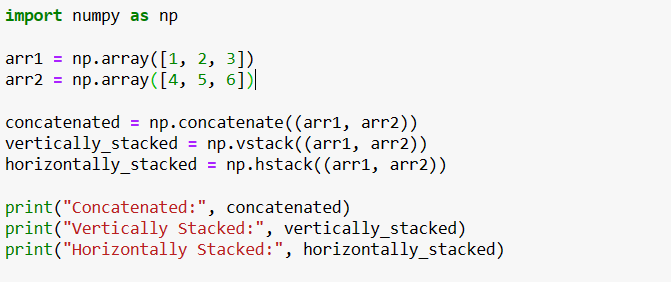
* **Output:**

****

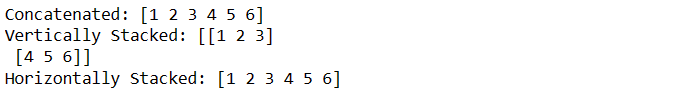
1. **Joining Arrays (concatenate() and vstack()/hstack()):**

NumPy provides functions to concatenate or stack arrays together, either vertically or horizontally.

* **Code:**

****

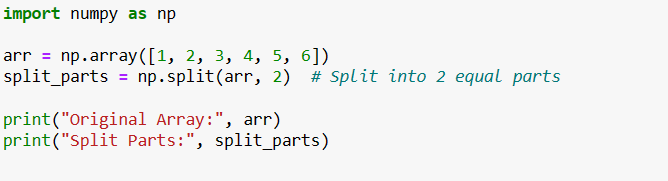
* **Output:**

****

1. **Splitting an Array (split() and vsplit()/hsplit()):**

To divide arrays into smaller parts, you can use the split() function, along with vsplit() and hsplit() for vertical and horizontal splitting.

* **Code:**

****

* **Output:**

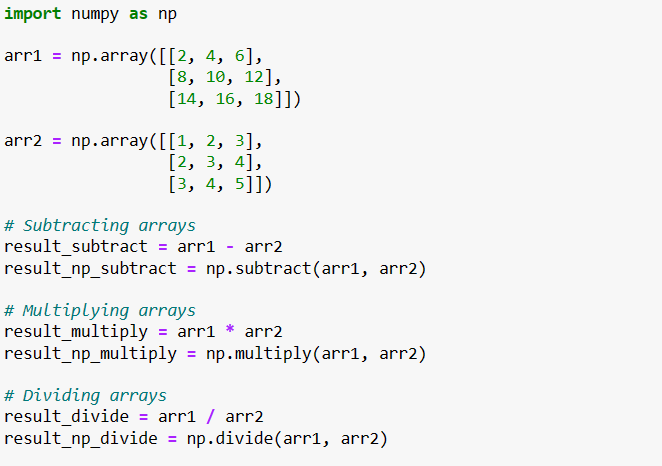
****

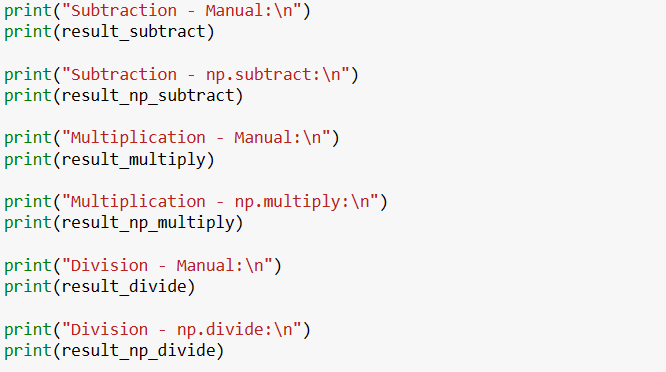
* **Mathematical Functions in NumPy:**

1. **Basic Arithmetic Operations:**

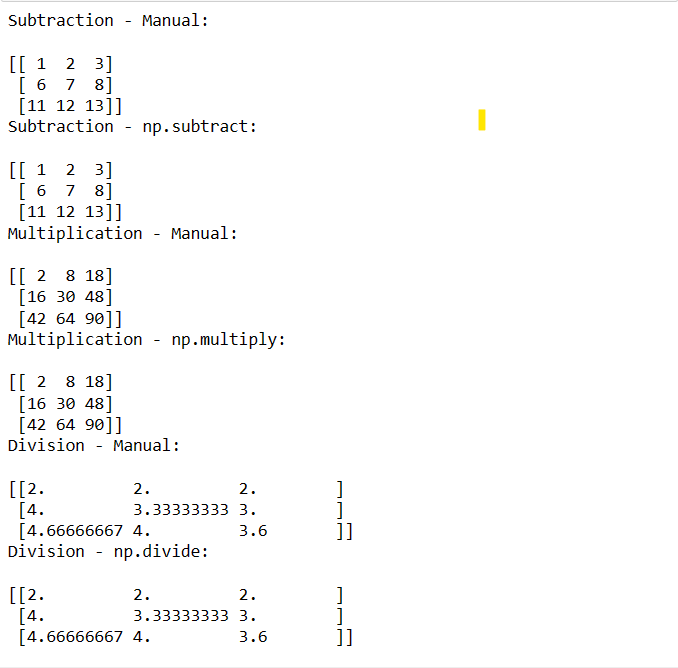
NumPy's basic arithmetic functions enable element-wise operations on arrays, simplifying operations across large datasets.

* **Code:**

****

****

* **Output:**

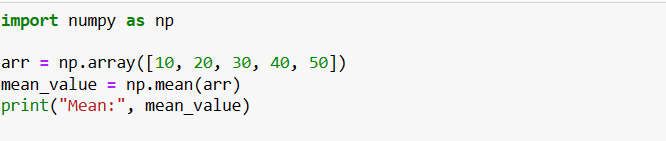


* **Statistical Functions:**

NumPy offers a variety of statistical functions, enabling you to compute measures like mean, standard deviation, and more across arrays.

1. **Mean:** Calculates the mean of the elements in a NumPy array.

* **Code:**

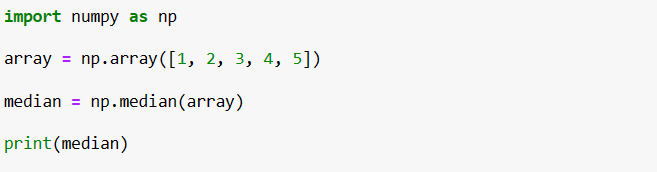
****

* **Output:**

****

1. **Median:** This function calculates the median of the elements in a NumPy array.

* **Code:**

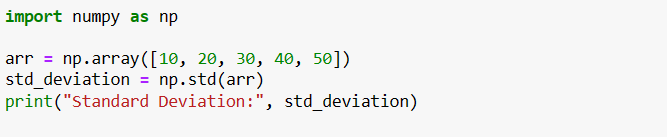
****

* **Output:**

****

1. **Standard deviation:** This function calculates the standard deviation of the elements in a NumPy array.

* **Code:**

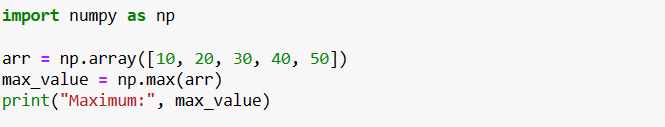
****

* **Output:**

****

1. **Maximum:** This function returns the maximum value of the elements in a NumPy array.

* **Code:**

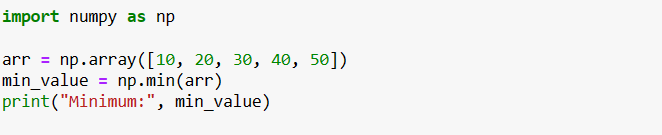


* **Output:**

****

1. **Minimum:** This function returns the minimum value of the elements in a NumPy array.

* **Code:**

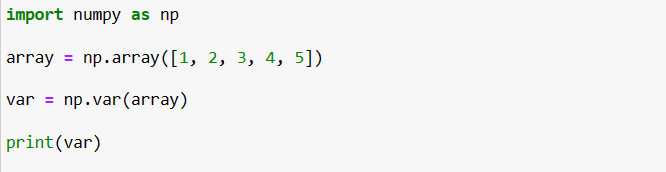
****

* **Output:**

****

1. **Variance:** This function calculates the variance of the elements in a NumPy array.

* **Code:**

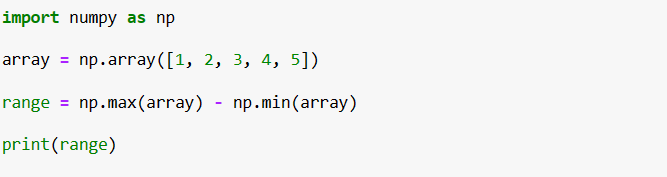
****

* **Output:**

****

1. **Range:** This function returns the range of the elements in a NumPy array. The range is the difference between the maximum and minimum values.

* **Code:**

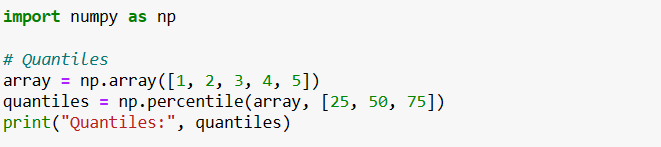
****

* **Output:**

****

1. **Quantiles:** This function calculates the nth percentile of the elements in a NumPy array.

* **Code:**

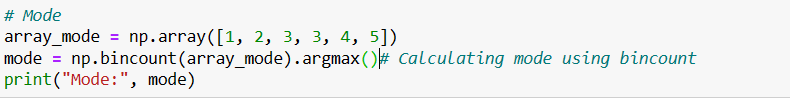
****

* **Output:**

****

1. **Mode:** This function calculates the most frequent value in a NumPy array.

* **Code:**

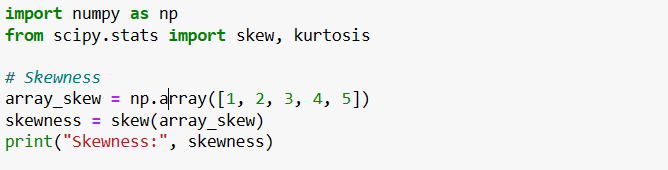
****

* **Output:**

****

1. **Skewness:** This function calculates the skewness of the elements in a NumPy array. Skewness measures the asymmetry of the distribution of the elements in the array.

* **Code:**

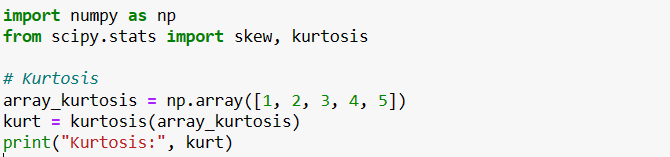


* **Output:**

****

1. **Kurtosis:** This function calculates the kurtosis of the elements in a NumPy array. Kurtosis measures the peakedness of the distribution of the elements in the array.

* **Code:**

****

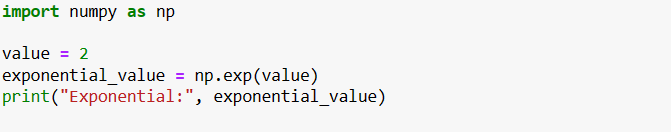
* **Output:**

****

* **Logarithmic and Exponential Functions:**

1. **exp():** Calculates the exponential of an element or array.

* **Code:**

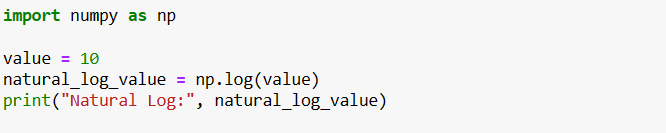
****

* **Output:**

****

1. **log():** Calculates the natural logarithm of an element or array.

* **Code:**

****

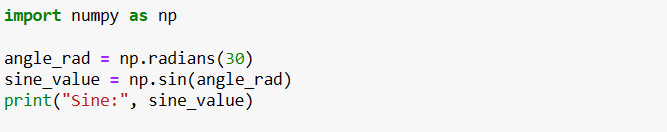
* **Output:**

****

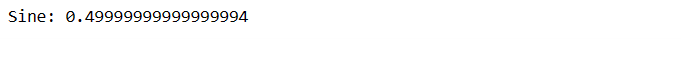
* **Trigonometric Functions in NumPy:**

1. **sin():** Calculates the sine of an angle.

* **Code:**

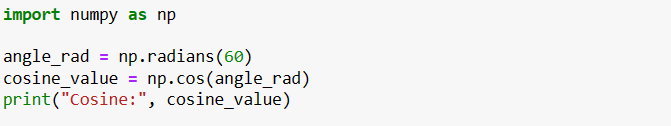
****

* **Output:**



1. **cos():** Calculates the cosine of an angle.

* **Code:**

****

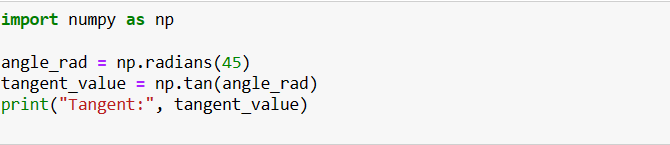
* **Output:**



1. **tan():** Calculates the tangent of an angle

.

* **Code:**

****

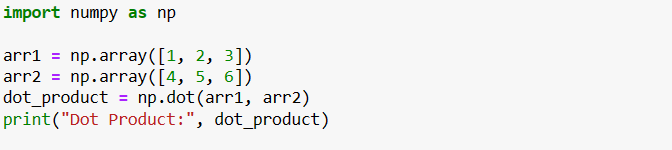
* **Output:**



* **Linear algebra functions:**

1. **dot():** Calculates the dot product of two NumPy arrays.

* **Code:**

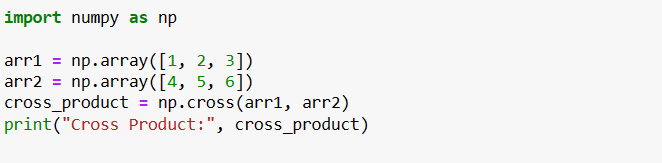
****

* **Output:**

****

1. **cross():** Calculates the cross product of two NumPy arrays.

* **Code:**

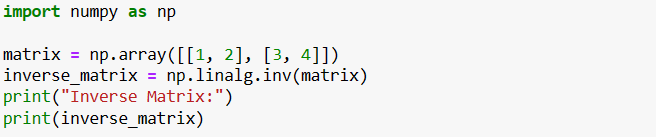
****

* **Output:**

****

1. **inv():** Calculates the inverse of a NumPy array.

* **Code:**

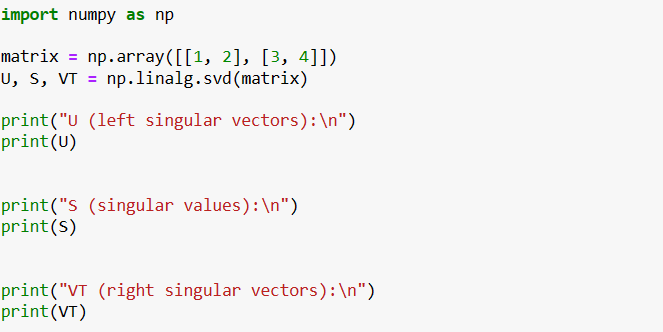
****

* **Output:**

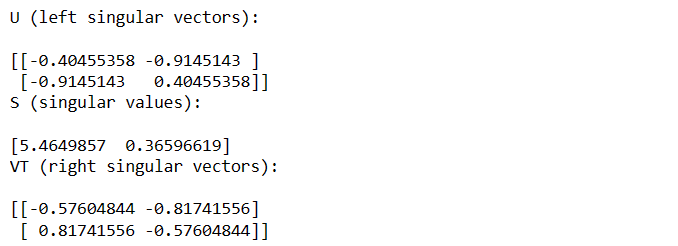
****

1. **svd():** Calculates the singular value decomposition of a NumPy array.

* **Code:**

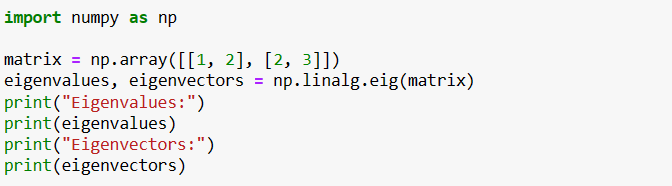
****

* **Output:**

****

1. **eig():** Calculates the eigenvalues and eigenvectors of a NumPy array.

* **Code:**

****

* **Output:**

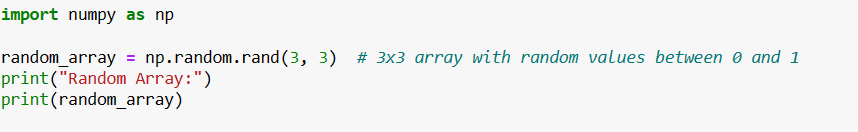
****

* **Random number functions:**

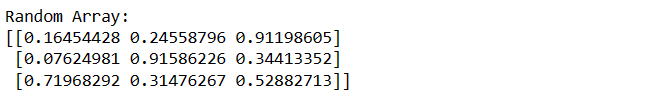
1. **random.rand() - Creates a NumPy array with random numbers:**

This function generates random numbers from a uniform distribution over the interval [0, 1) and creates a NumPy array of the specified shape with these random numbers.

* **Code:**

****

* **Output::**

****

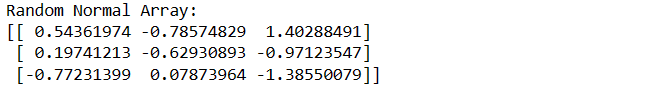
1. **random.randn() - Creates a NumPy array with normally distributed random numbers:**

This function generates random numbers from a standard normal distribution (mean = 0, standard deviation = 1) and creates a NumPy array of the specified shape with these random numbers.

* **Code:**

****

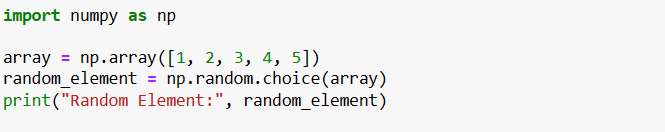
* **Output:**

****

1. **random.choice() - Chooses a random element from a NumPy array:**

This function randomly selects an element from a given NumPy array based on a uniform distribution. You can also specify probabilities for each element in the array to control the selection process.

* **Code:**

****

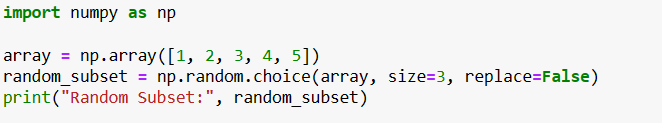
* **Output:**

****

1. **random.sample() - Samples a random subset of elements from a NumPy array:**

This function samples a specified number of elements randomly from a given NumPy array without replacement, creating a new array containing the sampled elements. The order of elements in the original array is preserved in the sampled array.

* **Code:**

****

* **Output:**

****

**3.1 Introduction to Pandas:**

Pandas is a powerful and popular open-source data manipulation and analysis library for the Python programming language. It provides data structures and functions that are specifically designed to efficiently work with structured data, such as tabular data (similar to spreadsheets or SQL tables), time series data, and more. Pandas is widely used in data science, data analysis, and machine learning projects due to its flexibility and ease of use.

* **You can install Pandas using the following steps:**

**1.Using pip:** Open your command-line interface and run the following command:



This will download and install Pandas from the Python Package Index (PyPI).

**2.Using conda (if you are using Anaconda distribution):** Open your command-line interface and run the following command:



Once you have Pandas installed, you can import it into your Python script or interactive environment using the following import statement:



The alias **"pd"** is commonly used as a shorthand for **"pandas"** to save typing.

* **Importance of Pandas:**

1. **Data Handling:** Pandas provides powerful tools for reading, writing, and manipulating data, making it easier to work with structured and tabular data. It handles various data formats such as CSV, Excel, SQL databases, JSON, and more.
2. **Data Cleaning and Transformation:** Real-world data is often messy and incomplete. Pandas offers functions to clean and preprocess data by handling missing values, removing duplicates, transforming data types, and more.
3. **Data Exploration:** With Pandas, you can quickly gain insights into your data by summarizing, aggregating, and visualizing it. This exploratory data analysis helps you understand the characteristics and trends within your dataset.
4. **Data Manipulation:** Pandas allows you to filter, sort, group, and reshape data to meet your analysis requirements. This flexibility is essential for tasks like generating reports, extracting specific information, and creating new derived features.
5. **Integration with Other Libraries:** Pandas seamlessly integrates with other popular libraries like NumPy (for numerical operations) and Matplotlib/Seaborn (for data visualization), providing a comprehensive toolkit for data analysis.

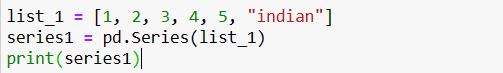
* **Use-Cases of Pandas:**

1. **Financial Analysis:** Pandas is used to analyze financial data, calculate statistics, and perform time-series analysis for stock market trends, portfolio management, risk assessment, and economic forecasting.
2. **Marketing and Customer Segmentation:** Data scientists use Pandas to segment customers based on various attributes, enabling targeted marketing campaigns and personalized recommendations.
3. **Machine Learning Data Preprocessing:** Before training machine learning models, data often needs to be cleaned, transformed, and normalized. Pandas simplifies these tasks by providing efficient methods to preprocess data.
4. **Scientific Research:** Pandas is valuable for scientific data analysis, helping researchers manipulate and analyze experimental data, perform statistical analysis, and visualize results.
5. **Business Intelligence and Reporting:** Data analysts use Pandas to create interactive dashboards, generate reports, and present insights to stakeholders.
6. **Social Media Analysis:** Pandas can be used to process and analyze social media data, extracting trends, sentiment analysis, and engagement metrics.
7. **Healthcare Analytics:** In the healthcare sector, Pandas is employed to manage and analyze patient data, medical records, and clinical trial results.
8. **Geospatial Analysis:** Pandas, along with libraries like GeoPandas, can be used to analyze geospatial data, such as mapping, spatial querying, and geographical visualization.

* **Pandas introduces two main data structures:**

1. **Series:** A Pandas Series is a fundamental data structure that represents a one-dimensional labeled array. It's a powerful tool for handling and analyzing data in Python. Each element in a Series has a corresponding label called an index. This index allows for easy and efficient data access and manipulation.

* **Creating a Series from a List:**
* **Code:**

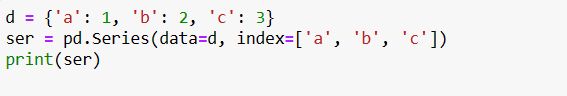
****

* **Output:**

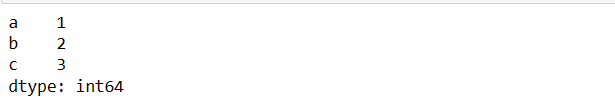
****

In this section, we explore the fundamental concept of a Pandas Series. A Series is like a column in a spreadsheet, capable of holding different types of data. We demonstrate this by creating a Series from a list that contains a mix of numeric and string values. The printed output displays the Series, showing each value alongside its index. Indices are automatically assigned.

* **Creating a Series from a Dictionary:**
* **Code:**

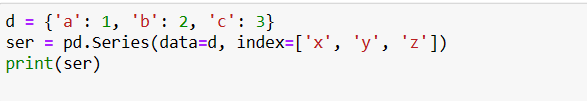
****

* **Output:**

****

A Series can also be constructed from a Python dictionary. Think of it as a way to translate a dictionary's key-value pairs into a tabular format. In this case, we create a Series using dictionary data and specify custom index labels. The printed output illustrates the Series structure.

* **Creating a Series from a Dictionary with Different Index:**
* **Code:**

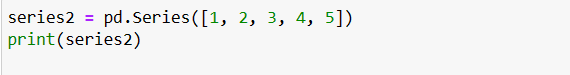
****

* **Output:**

****

By providing index labels that differ from the dictionary keys, we create a Series that aligns values with their corresponding labels. However, since the dictionary keys don't match the new indices, Pandas fills in the values with NaN, indicating missing or undefined data.

* **Creating a Series from a List (Simplified)**
* **Code:**

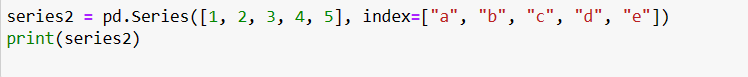
****

* **Output:**

****

Simplicity is at the core of Pandas. We can create a Series directly from a list of values. This example demonstrates that even without explicitly specifying index labels, Pandas assigns default integer indices.

* **Creating a Series from a List with Custom Index:**
* **Code:**

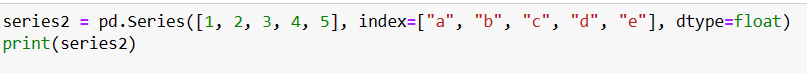
****

* **Output:**

****

Customization is key in data analysis. Here, we tailor a Series by adding custom index labels to our list of values. Now, accessing elements becomes more intuitive as they're associated with labels we define.

* **Creating a Series from a List with Custom Index and Data Type:**
* **Code:**

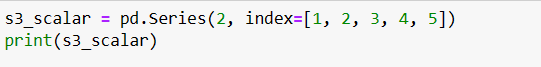
****

* **Output:**

****

Pandas is also equipped to handle single values. Here, we create a Series from a scalar value, which is automatically assigned an index of 0. This can be handy for creating a consistent structure.

* **Creating a Series from a Scalar Value with Custom Indices:**
* **Code:**

****

* **Output:**

****

Personalization doesn't stop at data values; it extends to indices as well. We showcase how to specify custom indices for a scalar Series. This flexibility allows us to align our data with the context of our analysis.

* **Creating a Series from a Python Dictionary:**
* **Code:**

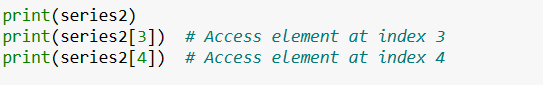
****

* **Output:**



Rounding off our introduction to Series creation, we demonstrate how a dictionary can be seamlessly transformed into a Series. The dictionary's keys become the indices, providing an organized structure for data exploration.

* **Accessing Elements in a Series:**
* **Code:**

****

* **Output:**

****

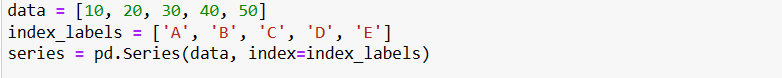
Accessing individual elements in a Series is akin to locating items on a shelf by their labels. We display the Series and then show how to retrieve specific elements using their index labels.

* **Slicing a Series:**

Slicing a Series refers to the process of extracting a subset of elements from the Series based on their positions in the index. It's a way to select a range of elements from the Series, much like how you might extract a portion of a list in Python.

In Pandas, you can use slicing to access a contiguous set of elements from a Series using index positions. The syntax for slicing is similar to what you might use for slicing lists or arrays in Python. For example, consider the following Series:

* **Code:**

****

If you want to slice elements from index position 1 to 3 (inclusive of 1 and 3), you can use the following

* **code:**

****

* **Output:**



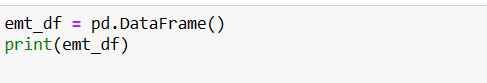
1. **DataFrame:** A two-dimensional tabular data structure that consists of columns, each of which can have different data types. It's similar to a table in a relational database or a spreadsheet.

* **Importance of DataFrames:**

Pandas DataFrames hold immense importance in data analysis and manipulation tasks for several reasons:

1. **Data Representation:** DataFrames offer an organized and human-readable representation of data, allowing you to view, access, and manage data more effectively.
2. **Data Cleaning and Transformation:** Data is rarely pristine. DataFrames empower you to preprocess and clean data by handling missing values, duplicates, and inconsistent formats.
3. **Data Exploration and Analysis:** DataFrames simplify exploratory data analysis by enabling summary statistics, aggregation, grouping, and visualization, helping you extract insights and patterns.
4. **Data Integration and Transformation:** DataFrames play a crucial role in integrating data from multiple sources, transforming data structures, and preparing data for machine learning.
5. **Data Visualization:** Combined with visualization libraries like Matplotlib and Seaborn, DataFrames enable informative and visually appealing data presentations.
6. **Time-Series Analysis:** DataFrames are well-suited for handling time-series data, a common requirement in fields like finance, economics, and scientific research.

* **Pandas in DataFrame:**
* **Creating an Empty DataFrame:**
* **Code:**

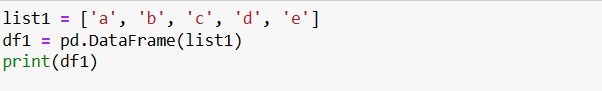
****

* **Output:**

****

In this section, we introduce the concept of a DataFrame a two-dimensional tabular data structure in Pandas. We begin by creating an empty DataFrame using the **pd.DataFrame()** constructor. An empty DataFrame serves as a foundation onto which we can add columns and data.

* **Creating a DataFrame from a List:**
* **Code:**

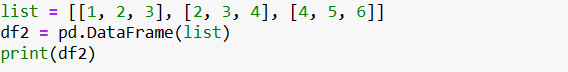
****

* **Output:**

****

This portion delves into creating a DataFrame from a simple Python list. The DataFrame constructor transforms the list elements into a single column with default numerical indices. This basic example showcases the DataFrame's structure and introduces the concept of columns.

* **Creating a DataFrame from a List of Lists:**
* **Code:**

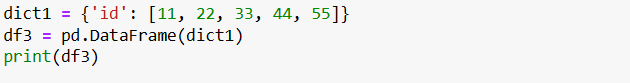
****

* **Output:**

****

Here, we explore creating a DataFrame from a list of lists. The result is a DataFrame with rows and columns, where each inner list corresponds to a row in the DataFrame. This example illustrates how structured data can be organized in a tabular format.

* **Creating a DataFrame from a Dictionary:**
* **Code:**

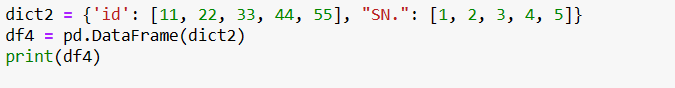


* **Output:**

****

This segment highlights creating a DataFrame from a Python dictionary. The keys of the dictionary become column headers, and the associated values form the column data. Dictionaries provide a powerful way to structure and organize data into DataFrame format.

* **Creating a DataFrame from a Dictionary with Multiple Columns:**
* **Code:**

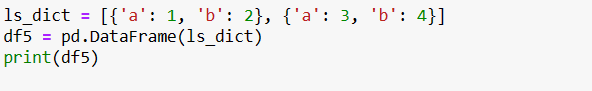
****

* **Output:**

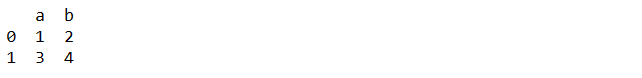
****

We showcase expanding the dictionary-based DataFrame creation to multiple columns. Each key in the dictionary becomes a separate column, and the associated values populate the respective columns. This example demonstrates how to handle more complex data structures.

* **Creating a DataFrame from a List of Dictionaries:**
* **Code:**

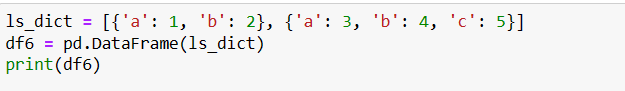
****

* **Output:**

****

This section introduces the concept of creating a DataFrame from a list of dictionaries. Each dictionary corresponds to a row in the DataFrame, with keys mapping to column headers and values filling the cells. This approach offers a structured way to represent heterogeneous data.

* **Creating a DataFrame from a List of Dictionaries with Varying Keys:**
* **Code:**

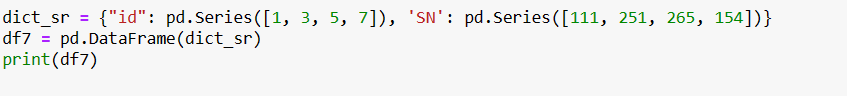
****

* **Output:**

****

Expanding on the previous example, we cover cases where dictionaries have varying keys. The resulting DataFrame accommodates all the keys found in the dictionaries, and for missing keys, the corresponding cells are populated with NaN, indicating missing data.

* **Creating a DataFrame from a Dictionary of Series:**
* **Code:**

****

* **Output:**

****

This segment introduces a more advanced approach: creating a DataFrame from a dictionary of Pandas Series. Each Series becomes a column in the DataFrame, and the DataFrame constructor aligns the data based on their index labels. This method is particularly useful for incorporating labeled data into DataFrame structures.

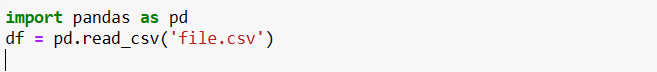
* **Reading & Writing Data with Pandas:**
* **Reading Data:**

Pandas provides powerful tools for reading data from various file formats into a DataFrame, a structured two-dimensional tabular data structure. This capability is particularly useful for data analysis and manipulation. Here are some essential methods for reading data:

* **Reading CSV Files:**

To read data from a CSV (Comma-Separated Values) file into a DataFrame, you can use the **pd.read\_csv()** function. CSV files are commonly used for storing tabular data, where each line represents a row and columns are separated by commas.

* **Code:**

****

The read\_csv() function loads the contents of the CSV file into a DataFrame, where each row becomes a DataFrame row, and columns are automatically determined from the file's header row.

* **Reading Excel Files**

Pandas also supports reading data from Excel files using the **pd.read\_excel()** function. Excel files can contain multiple sheets, and each sheet can be loaded into a separate DataFrame.

* **Code:**

****

The **read\_excel()** function reads the data from the specified Excel file, making it available in DataFrame format for further analysis.

* **Writing Data:**

Pandas enables you to easily export your DataFrame data to various file formats. This is invaluable for sharing your results, collaborating with others, or simply saving your work for future reference.

* **Writing to CSV Files**

To save a DataFrame as a CSV file, you can use the **to\_csv()** function. This function allows you to specify the file name and path where the CSV file will be created.

* **Code:**

****

The **to\_csv()** function writes the DataFrame's contents to a CSV file. The **index=False** parameter prevents the DataFrame's index from being saved as an additional column in the CSV file.

* **Writing to Excel Files**

Exporting a DataFrame to an Excel file is achieved using the **to\_excel()** function. You can specify the file name and the sheet name within the Excel file where the DataFrame will be saved.

* **Code:**

****

The **to\_excel()** function stores the DataFrame's contents in the specified Excel file, allowing you to organize and share your data in an Excel format.

* **Data Inspection Functions:**

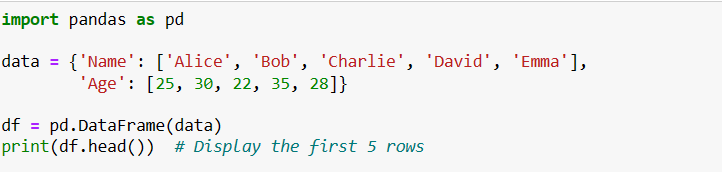
Data inspection refers to the process of examining and analyzing a dataset to gain an understanding of its structure, contents, quality, and overall characteristics. It involves exploring the data to identify patterns, anomalies, missing values, data types, and other attributes that provide insights into the dataset's nature. Data inspection is a crucial step in the data analysis workflow, as it lays the foundation for data cleaning, transformation, and further analysis.

* **Key aspects of data inspection include:**
  1. **Overview of Data:** Data inspection provides an initial overview of the dataset, including the number of rows and columns, data types of each column, and memory usage. This information helps in understanding the basic structure of the dataset.
  2. **Identifying Missing Values:** Inspecting data reveals if there are any missing values in the dataset. Missing values can impact analysis and modeling, so identifying them is essential for data cleaning.
  3. **Descriptive Statistics:** Data inspection often involves calculating basic statistics such as mean, median, standard deviation, minimum, maximum, and percentiles for numerical columns. These statistics provide insights into the central tendencies and distributions of the data.
  4. **Data Distribution:** By visualizing histograms, box plots, or other plots, data inspection allows you to understand the distribution of values in different columns. This is particularly useful for identifying outliers or skewed distributions.
  5. **Categorical Data Analysis:** For categorical columns, data inspection can involve analyzing unique values, their frequencies, and proportions. This is important for understanding the composition of categorical variables.
  6. **Data Exploration:** Data inspection enables you to explore relationships between variables. For instance, examining correlations between numerical variables helps identify potential dependencies.
  7. **Data Quality Assessment:** Inspecting data helps in assessing its quality. This includes checking for inconsistencies, incorrect data types, or unexpected values that might require data cleaning.
  8. **Initial Visualization:** Basic visualizations, such as scatter plots, bar charts, or line plots, can be generated during data inspection to gain initial insights into the data's patterns and trends.

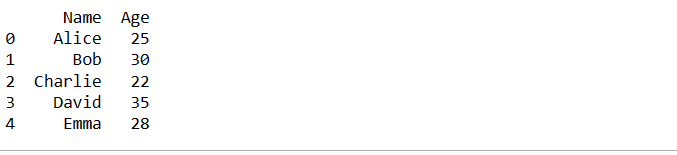
1. **df.head():**

The head() function is used to display the first few rows of a DataFrame. By default, it displays the first 5 rows, but you can specify the number of rows to show as an argument.

* **Code:**

****

* **Output:**

****

**2. df.tail():**

Similar to **head(), t**he **tail()** function displays the last few rows of a DataFrame. By default, it shows the last 5 rows, but you can specify the number of rows to display.

* **Code:**



* **Output:**

****

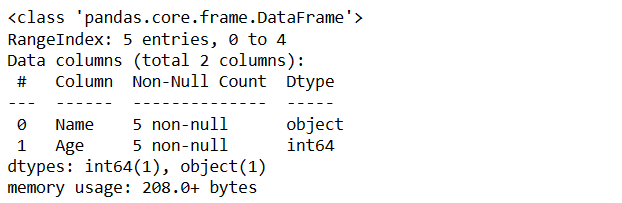
**3. df.info():**

The **info()** function provides essential information about a DataFrame, including the data types of each column, the number of non-null values, and memory usage. It's a quick way to understand the structure of your data.

* **Code:**

****

* **Output:**

****

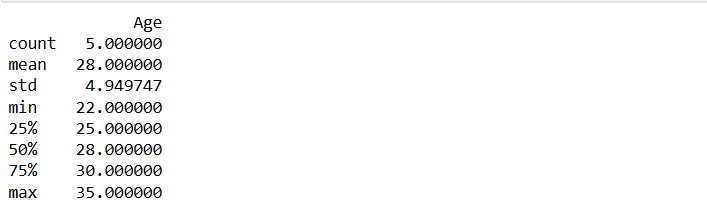
**4. df.describe():**

The **describe()** function generates summary statistics for numerical columns in the DataFrame. It provides information like mean, standard deviation, minimum, maximum, and percentiles.

* **Code:**

****

* **Output:**

****

* **Data Selection techniques in Pandas**:

Data selection techniques in Pandas refer to methods that allow you to extract specific data from a DataFrame based on various criteria. These techniques are essential for analyzing, transforming, and visualizing data effectively. Here are some common data selection techniques:

**1. Column Selection:**

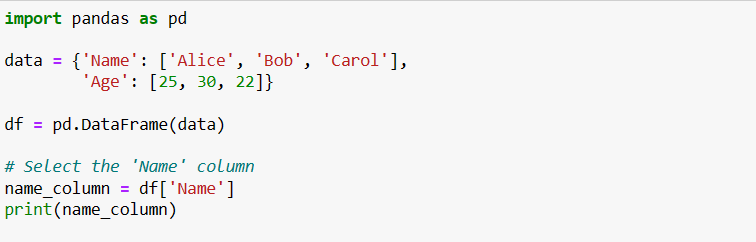
* **Select a Single Column:**

This technique allows you to extract a single column from a DataFrame. The column is returned as a Pandas Series.

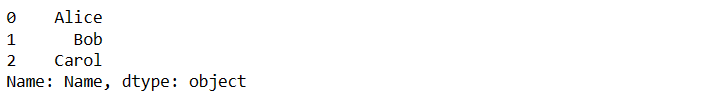
* **Syntax:**

**selected\_column = df['column\_name']**

* **Example:**
* **Code:**

****

* **Output:**

****

* **Select Multiple Columns:**

This technique allows you to extract multiple columns from a DataFrame. The selected columns are returned as a new DataFrame.

* **Syntax:**

**selected\_columns = df[['col1', 'col2']]**

* **Example:**
* **Code:**

****

* **Output:**

****

**2. Row Selection**

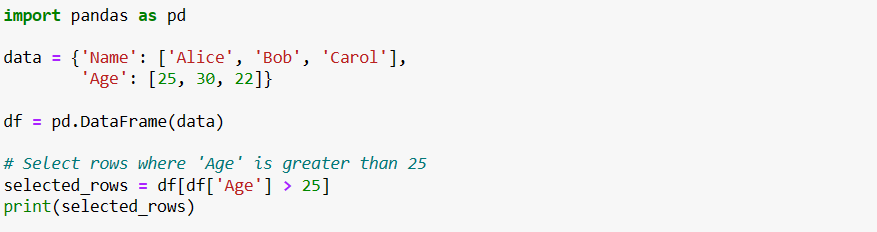
* **Select Rows Based on Condition:**

This technique allows you to select rows from a DataFrame based on a specific condition applied to a column. It returns a DataFrame containing only the rows that satisfy the given condition.

* **Syntax:**

**selected\_rows = df[df['column'] > value]**

* **Example:**
* **Code:**

****

* **Output:**

****

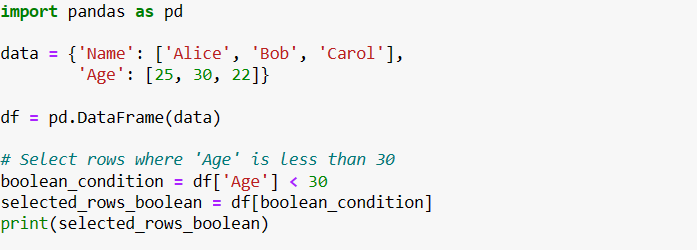
* **Select Rows Using Boolean Indexing: df[boolean\_condition]**

This technique involves using a boolean condition (True/False) to select rows from a DataFrame. The condition can be applied to any column, and it returns a DataFrame containing only the rows where the condition is True.

* **Syntax:**

**selected\_rows\_boolean = df[boolean\_condition]**

* **Example:**
* **Code:**

****

* **Output:**

****

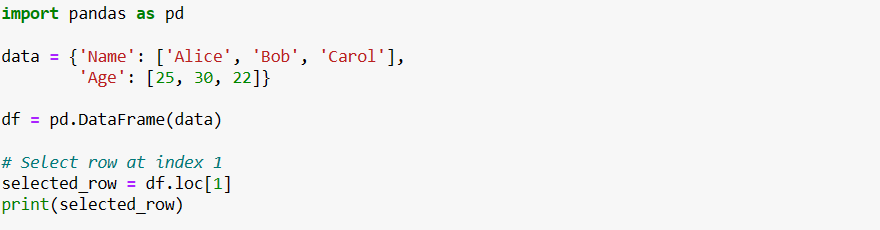
* **Select Rows by Index:**

This technique allows you to select rows from a DataFrame using their index labels. It returns a single row as a Pandas Series.

* **Syntax:**

**selected\_row\_by\_label = df.loc[row\_index]**

* **Example:**
* **Code:**

****

* **Output:**

****

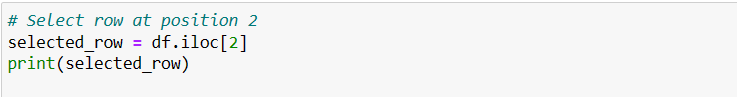
* **Select Rows by Position: df.iloc[row\_position]**

This technique allows you to select rows from a DataFrame using their integer positions. It returns a single row as a Pandas Series.

* **Syntax:**

**selected\_row\_by\_position = df.iloc[row\_position]**

* **Example:**
* **Code:**

****

* **Output:**

****

**3. Combining Row and Column Selection:**

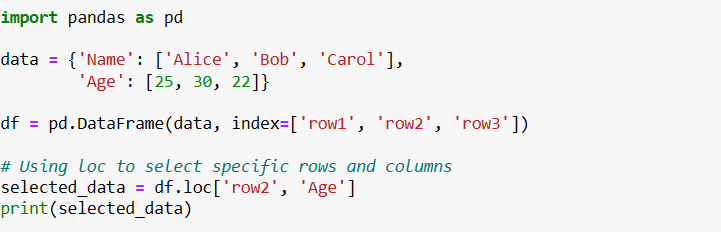
* **Select Specific Value Using Label-Based Indexing:**

The loc function in Pandas is used for label-based indexing. It allows you to select rows and columns from a DataFrame based on their labels (index names) rather than their integer positions. This function provides a more intuitive way to access data by using meaningful row and column labels.

* **Syntax:**

**selected\_data = df.loc[row\_labels, column\_labels]**

* **Parameters:**
* **row\_labels:** Labels of the rows you want to select.
* **column\_labels:** Labels of the columns you want to select.
* **Example:**
* **Code:**

****

* **Output:**



In this example, the **specific\_value** is obtained by specifying the label of the **row (2)** and the label of the column **('Age').** This method provides a clear and intuitive way to retrieve a value based on the row and column labels.

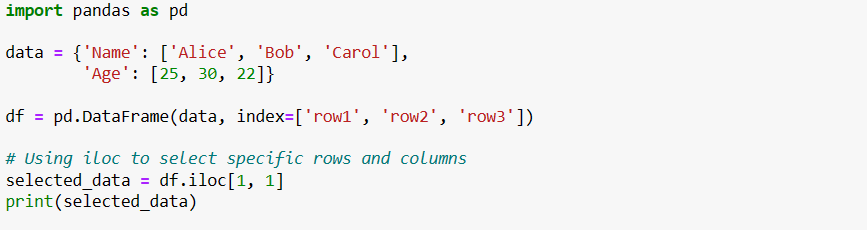
* **Select Specific Value Using Position-Based Indexing**:

The iloc function in Pandas is used for position-based indexing. It allows you to select rows and columns from a DataFrame based on their integer positions. This function is particularly useful when you want to access data programmatically without the need for meaningful labels.

* **Syntax:**

**selected\_data** = df.iloc[row\_positions, column\_positions]

* **Parameters:**
* **row\_positions:** Integer positions of the rows you want to select.
* **column\_positions:** Integer positions of the columns you want to select.
* **Example:**
* **Code:**

****

* **Output:**

****

In this example, the **iloc[1, 1]** is obtained by specifying the integer position of the row (**1**, which is the second row) and the integer position of the column (**1**, which is the second column). Position-based indexing is particularly helpful when you want to access values in a more programmatic manner.

**4. Indexing and Slicing:**

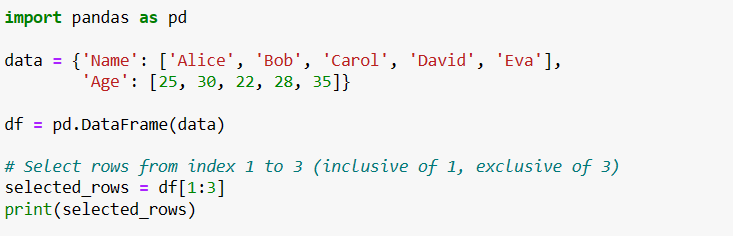
* **Select Rows Using Index Range:**

This technique allows you to slice a range of consecutive rows from a DataFrame using integer positions. The end index is exclusive, meaning the row at the end index will not be included in the selection.

* **Syntax:**

**selected\_rows\_range = df[start:end]**

* **Parameters:**
* **start:** The starting index of the slice (inclusive).
* **end:** The ending index of the slice (exclusive).
* **Example:**
* **Code:**

****

* **Output:**

****

* **Select Rows Using Labels and Indexers:**

This technique allows you to slice a range of consecutive rows from a DataFrame using label-based indexing. The end label is inclusive in this case, meaning the row at the end label will also be included in the selection.

* **Syntax:**

**selected\_rows\_labels = df.loc[start:end]**

* **Parameters:**
* **start:** The starting label of the slice (inclusive).
* **end:** The ending label of the slice (inclusive).
* **Example:**
* **Code:**

****

* **Output:**

****

* **Select Rows Using Positions:**

This technique allows you to slice a range of consecutive rows from a DataFrame using position-based indexing.

* **Syntax:**

**selected\_rows\_positions = df.iloc[start:end]**

* **Parameters:**
* **start:** The starting position of the slice (inclusive).
* **end:** The ending position of the slice (exclusive).
* **Example:**
* **Code:**



* **Output:**

****

* **Data Manipulation Functions in Pandas:**

**1. Descriptive Statistics in Pandas:**

Descriptive statistics are essential for understanding the distribution and characteristics of numeric data in a DataFrame. Pandas provides several methods to generate summary statistics and calculate basic statistical measures for numeric columns.

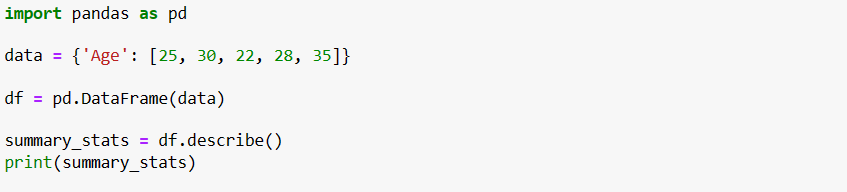
* **df.describe() Method:**

The describe() method generates a comprehensive summary of descriptive statistics for each numeric column in the DataFrame. It includes count, mean, standard deviation, minimum, maximum, and quartiles (25th, 50th, and 75th percentiles).

* **Syntax:**

**summary\_stats = df.describe()**

* **Example:**
* **Code:**

****

* **Output:**

****

* **df.mean(), df.median(), df.std() Methods:**

These methods calculate specific statistical measures for numeric columns in the DataFrame:

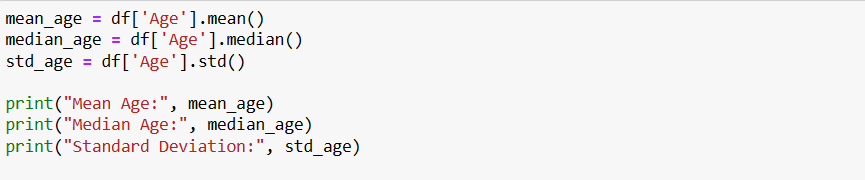
* **df.mean():** Calculate the mean (average) of numeric columns.
* **df.median():** Calculate the median (middle value) of numeric columns.
* **df.std():** Calculate the standard deviation of numeric columns.
* **Syntax:**

**mean\_value = df.mean()**

**median\_value = df.median()**

**std\_value = df.std()**

* **Example:**
* **Code:**

****

* **Output:**

****

* **df.min() and df.max() Methods:**

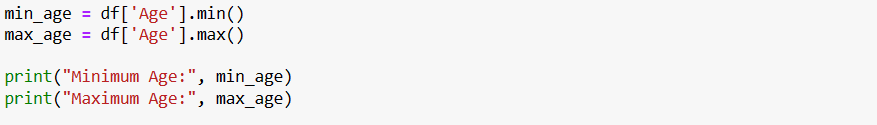
These methods find the minimum and maximum values within numeric columns:

* **df.min():** Identifies the minimum value in numeric columns.
* **df.max():** Identifies the maximum value in numeric columns.
* **Syntax:**

**min\_value = df.min()**

**max\_value = df.max()**

* **Example:**
* **Code:**

****

* **Output:**

****

**2. Sorting and Ranking in Pandas:**

Sorting and ranking data in a DataFrame are essential for organizing and analyzing data based on specific columns. Pandas provides methods that allow you to sort data based on column values and assign ranks to elements in a column.

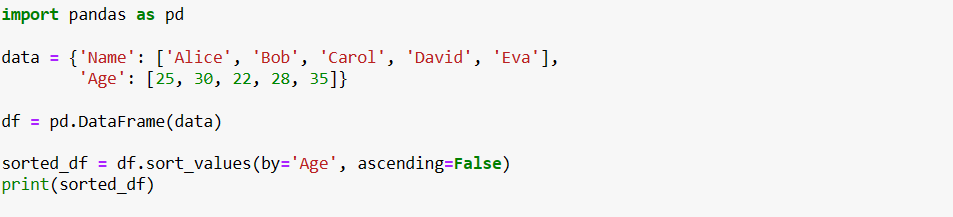
* **df.sort\_values() Method:**

The **sort\_values()** method is used to arrange the rows of a DataFrame in ascending or descending order based on one or more columns.

* **Syntax:**

**sorted\_df = df.sort\_values(by='column\_name', ascending=True)**

* **Parameters:**
* **by:** Specifies the column name(s) to sort by.
* **ascending:** Controls the sorting order, where True (default) sorts in ascending order, and False sorts in descending order.
* **Example:**
* **Code:**

****

* **Output:**

****

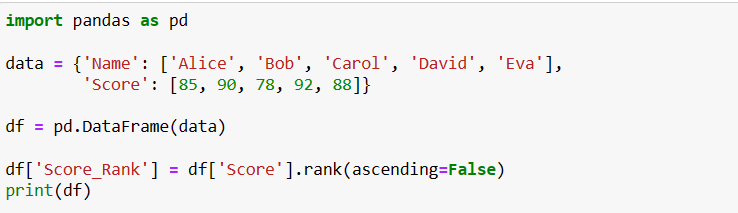
* **df.rank() Method:**

The **rank()** method assigns ranks to the elements in a column. Ranks can be calculated based on ascending or descending order.

* **Syntax:**

**df['rank\_column'] = df['column'].rank(ascending=True)**

* **Parameters:**
* **column:** The column for which ranks are assigned.
* **ascending:** Controls the ranking order, where True (default) ranks in ascending order, and False ranks in descending order.
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Grouping and Aggregation:**

* **Grouping Data:**

Grouping data involves categorizing rows in a DataFrame based on the values in one or more columns. This allows you to analyze subsets of data separately. The **groupby()** function in Pandas is used to initiate this grouping process.

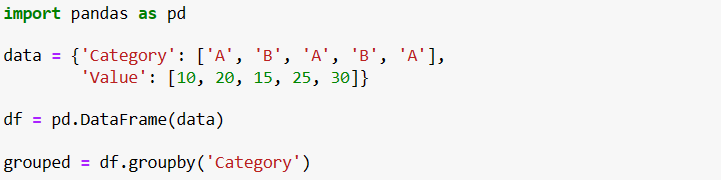
* **Syntax:**

**grouped = df.groupby('grouping\_column')**

* **Example:**

Suppose you have a DataFrame df with columns 'Category' and 'Value'. To group data based on the 'Category' column:

* **Code:**

****

* **Aggregation:**

After grouping data, you can perform aggregation to calculate summary statistics for each group. Aggregation functions like **sum(), mean(), median(), max(), and min()** are commonly used.

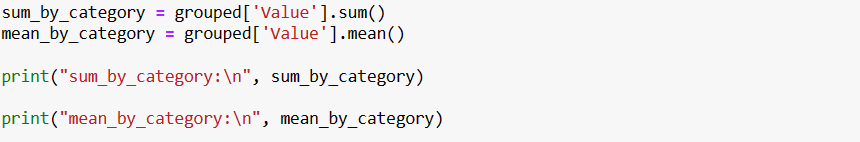
* **Syntax:**

**aggregated = grouped['numeric\_column'].aggregation\_function()**

* **Example:**

Continuing with the previous example, you can calculate the sum and mean of 'Value' for each category:

* **Code:**

****

* **Output:**

****

* **You can perform multiple aggregations at once using the agg() function:**
* **Syntax:**

**aggregated = grouped.agg({'numeric\_column': ['sum', 'mean']})**

* **Example:**
* **Code:**

****

* **Output:**

****

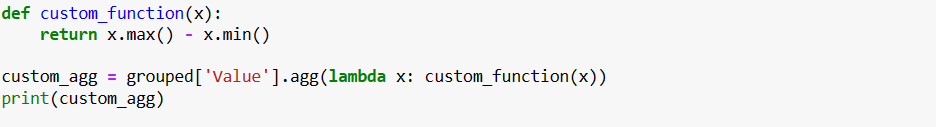
* **Custom aggregation is also possible using lambda functions or user-defined functions:**
* **Syntax:**

**custom\_agg = grouped['numeric\_column'].agg(lambda x: custom\_function(x))**

* **Example:**

Let's say you want to calculate the range (max - min) of **'Value'** for each category:

* **Code:**

****

* **Output:**

****

1. **Joining and Merging:**

* **Join DataFrames:**

Joining involves combining DataFrames based on their indices or columns. The join() method is used for this purpose.

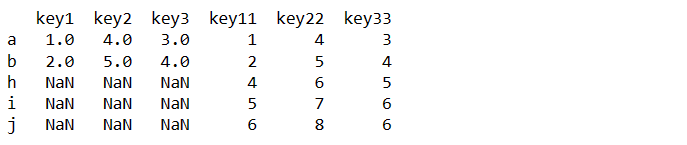
* **Syntax:**

**result = df1.join(df2, on='common\_column', how='join\_type')**

* **Parameters:**
* **on:** The column to join on.
* **how:** Type of join**: 'left', 'right', 'inner', or 'outer'.**
* **Example:**
* **Code:**

****

* **Output:**

****

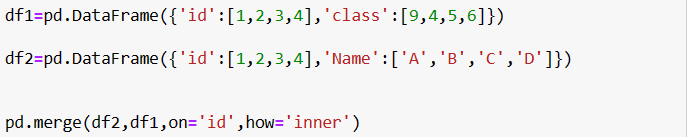
* **Merge DataFrames:**

Merging combines DataFrames using specified columns as keys. The merge() function provides various join strategies.

* **Syntax:**

**result = pd.merge(df1, df2, on='common\_column', how='merge\_type')**

* **Parameters:**
* **on:** The column(s) to merge on.
* **how:** Type of merge: **'left', 'right', 'inner', or 'outer'.**
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Pivoting and Reshaping in Pandas:**

Pivoting and reshaping are methods that allow you to restructure data, making it more suitable for analysis, visualization, and interpretation. These techniques are essential for working with data that needs to be presented or analyzed in various formats.

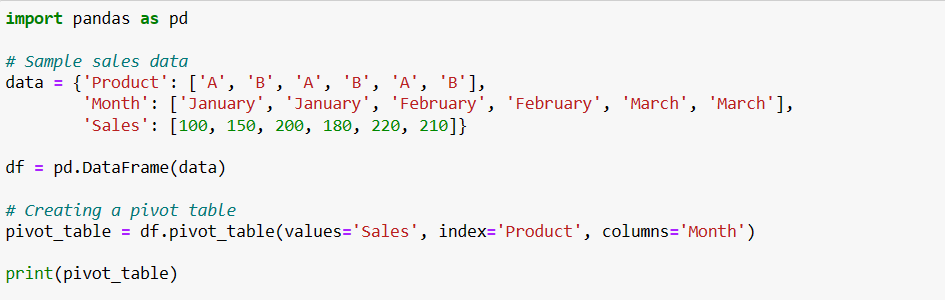
* **Pivot Data:**
* **Pivot\_table() Method:**

The **pivot\_table()** method is used to reshape data into a pivot table. This technique allows you to convert data from a long format to a wide format, where you can easily analyze and compare values across different categories.

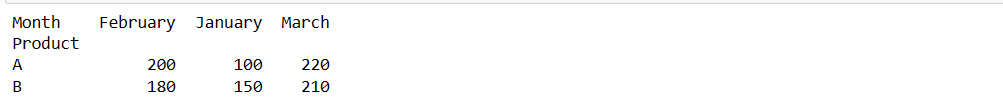
* **Syntax:**

**pivot\_table=df.pivot\_table(values='value\_column',index='index\_column',columns='column\_to\_pivot')**

* **Parameters:**
* **values:** The column whose values will populate the new columns**.**
* **index:** The column that will become the new index.
* **columns:** The column whose unique values will become new columns.
* **Example:**
* **Code:**

****

* **Output:**

****

* **Melt Data:**
* **Melt() Function:**

The **melt()** function transforms data from a wide format to a long format. It's particularly useful when you need to reshape data for specific types of analysis or visualization.

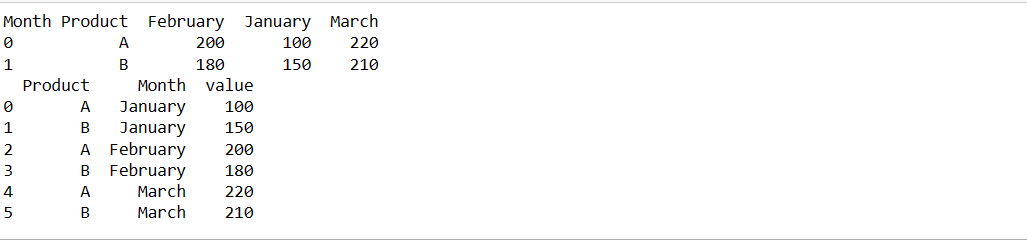
* **Syntax:**

**melted\_df = pd.melt(df, id\_vars=['id\_columns'], value\_vars=['value\_columns'])**

* **Parameters:**
* **id\_vars:** Columns that will remain as identifiers while melting.
* **value\_vars:** Columns that will be melted.
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Applying Functions:**

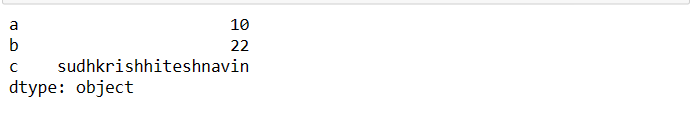
* **Applies a Function Along an Axis of the DataFrame:**
* This method applies a given function to the specified axis of the DataFrame.
* It's commonly used to perform operations column-wise or row-wise.
* **Syntax:**

**df.apply(func, axis=0)**

* **Parameters:**
* **func:** The function to apply. It can be a built-in function, a user-defined function, or a lambda function.
* **axis:** Specifies the axis along which to apply the function **(0 for columns, 1 for rows).**
* **Example:**
* **Code:**

****

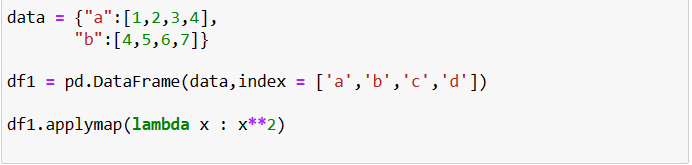
* **Output:**

****

* **Applies a Function Element-wise**
* This method applies a given function element-wise to every element in the DataFrame.
* It's suitable for performing simple operations on each individual value.
* **Syntax:**

**df.applymap(func)**

* **Parameters:**
* **func:** The function to apply.
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Handling Missing Data:**

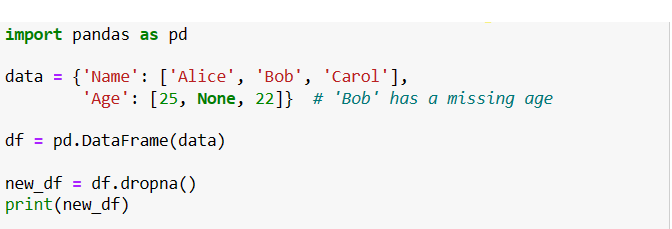
* **Drop Rows with Missing Values:**

The **dropna()** method is used to remove rows that contain missing values. You can use it to clean up your DataFrame by eliminating rows with incomplete or irrelevant information.

* **Syntax:**

**new\_df = df.dropna()**

* **Parameters:**
* **axis:** Specifies the axis along which to drop missing values. Use **axis=0** to drop rows with missing values (default), and **axis=1** to drop columns.
* **how:** Determines how to drop rows/columns. Options are **'any'** (default) to drop if any value is missing, or **'all'** to drop if all values are missing.
* **thresh:** Specifies the minimum number of non-null values required to keep a row/column. For example, setting **thresh=2** means a row/column with at least 2 non-null values will be kept.
* **subset:** Specifies a list of columns to consider when dropping missing values. Only rows with missing values in the specified columns will be dropped.
* **Example:**
* **Code:**



* **Output:**

****

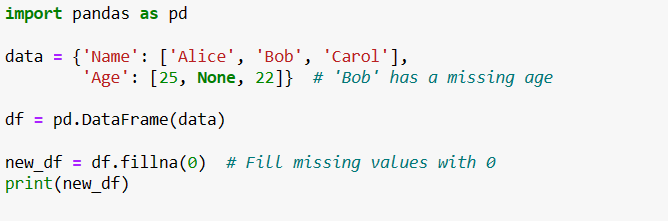
* **Fill Missing Values:**

The **fillna()** method is used to fill missing values in your DataFrame with specified values. This can be useful if you want to replace missing data with a default value or an estimated value.

* **Syntax:**

**new\_df = df.fillna(value)**

* **Parameters:**
* **value:** Specifies the value to use for filling missing data. This can be a scalar value, a dictionary of column-value pairs, or a Series**.**
* **method:** Determines the method to fill missing values. Options are **'ffill' (forward fill)** to fill with the previous value**, 'bfill' (backward fill)** to fill with the next value, or **None (default**) to fill with the specified value.
* **axis:** Specifies the axis along which to fill missing values. Use **axis=0 to** fill along rows (default), and **axis=1 to** fill along columns**.**
* **inplace:** If **True**, the original DataFrame will be modified in place, and no new DataFrame will be returned.
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Adding and Deleting Columns in Pandas:**

* **Adding Columns:**

In Pandas, adding columns to a DataFrame is a fundamental operation that allows you to enrich your data or compute new information based on existing columns. Here are two common ways to add new columns:

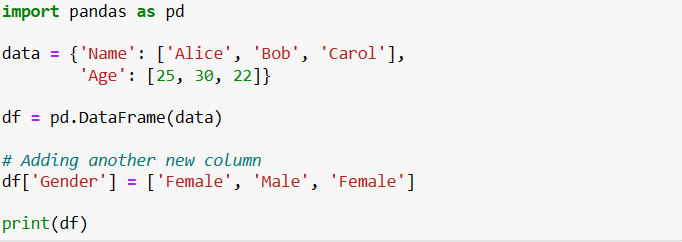
* **Using Assignment:**

You can directly assign a list, array, or Series to a new column name to add it to the DataFrame. This is a straightforward way to introduce new data.

* **Syntax:**

**df['new\_column'] = [value1, value2, ...]**

* **Example:**
* **Code:**

****

* **Output:**

****

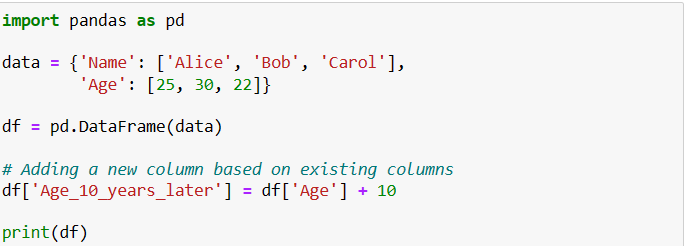
* **Using Calculations:**

You can perform operations on existing columns and assign the result to a new column. This is useful for deriving insights or creating new features.

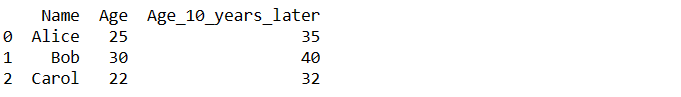
* **Syntax:**

**df['new\_column'] = df['existing\_column'] \* 2**

* **Example:**
* **Code:**

****

* **Output:**

****

* **Deleting Columns:**

Removing columns from a DataFrame can help simplify your data or focus on specific aspects. There are two common methods to delete columns:

* **Using del Keyword:**

The **del keyword** is a quick way to remove a column from the DataFrame. It's an in-place operation that directly modifies the DataFrame.

* **Syntax:**

**del df['column\_to\_delete']**

* **Example:**
* **Code:**

****

* **Output:**

****

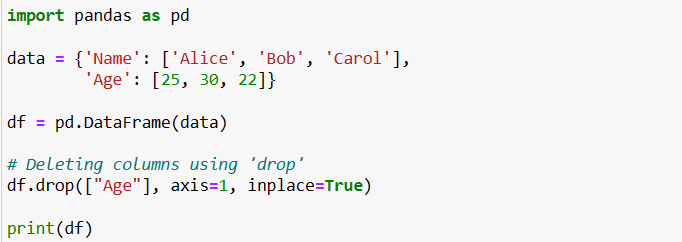
* **Using drop() Method:**

The **drop()** method provides more control and flexibility when removing columns. Remember to set the axis parameter to 1 to indicate that you're removing columns

* **Syntax:**.

**df.drop(['column\_to\_delete1', 'column\_to\_delete2'], axis=1, inplace=True)**

* **Example:**
* **Code:**

****

* **Output:**

****

1. **Duplicated and Unique Values:**

Duplicated values are data entries that occur more than once in a dataset. Pandas provides methods to detect and handle duplicated values in DataFrames.

* **df.duplicated():**

The **df.duplicated()** method returns a boolean Series indicating whether each row is a duplicate or not. Each occurrence of a duplicated row is marked as True, and only the first occurrence is marked as False.

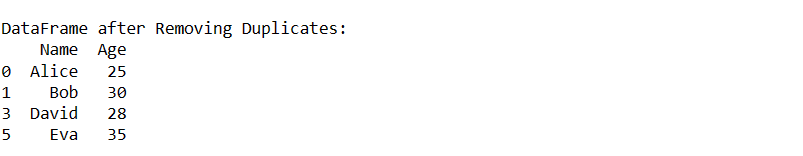
* **Syntax:**

**duplicated\_series = df.duplicated(subset=None, keep='first')**

* **Parameters:**
* **subset:** Specifies the columns to consider when identifying duplicates. By default, it considers all columns.
* **keep:** Specifies which occurrence to mark as False. Options are 'first' (default), 'last', and False to mark all duplicates as True.
* **Example:**
* **Code:**

****

* **Output:**

****

* **df.drop\_duplicates():**

The **df.drop\_duplicates()** method removes duplicated rows from the DataFrame.

* **Syntax:**

**cleaned\_df = df.drop\_duplicates(subset=None, keep='first', inplace=False)**

* **Parameters:**
* **subset:** Specifies the columns to consider when identifying duplicates. By default, it considers all columns.
* **keep:** Specifies which occurrence to keep. Options are 'first' (default), 'last', and False to drop all duplicates.
* **inplace:** If True, modifies the DataFrame in place and returns None. If False (default), returns a new DataFrame with duplicates removed.
* **Example:**
* **Code:**

****

* **Output:**

****

1. **Time Series Operations:**

Time series data involves sequences of data points indexed by time. Pandas provides specialized data structures and functions for working with time series data efficiently.

* **Creating a Date Range:**
* **Code:**

****

The **pd.date\_range** function generates a range of dates between the specified start and end dates.

* **Creating a DataFrame with Dates:**
* **Code:**

****

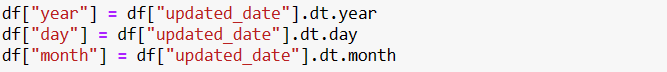
This code creates a DataFrame named df\_date with a single column named "date," which contains the date range generated previously.

* **Converting String Dates to Datetime:**
* **Code:**

****

The **pd.to\_datetime** function is used to convert string dates into datetime objects, which makes them suitable for date-based calculations and operations.

* **Extracting Year, Month, and Day:**
* **Code:**

****

The **.dt** accessor allows you to extract various components from datetime objects, such as year, month, and day.

* **Working with Timedeltas:**
* **Code:**

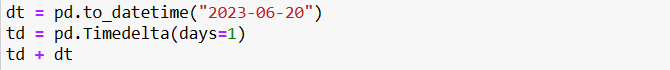
****

* **Output:**

****

The **pd.Timedelta** function creates a timedelta object that represents a duration of time. In this case, it's a duration of 1 day, 5 hours, and 45 minutes.

* **Performing Arithmetic with Dates:**
* **Code:**

****

* **Output:**

****

You can perform arithmetic operations with datetime objects and timedeltas. In this case, you're adding a timedelta of 1 day to the datetime "2023-06-20," resulting in "2023-06-21 00:00:00."

* **Window functions in Pandas**

Window functions, also known as windowing or analytic functions, are powerful tools in pandas (and SQL) that allow you to perform calculations across a "window" or "window frame" of rows related to the current row. These functions can be used to compute values that depend on a specific range of rows within the dataset, which can be especially useful for time series analysis, ranking, and aggregations.

Pandas provides a variety of window functions through **the .rolling() and .expanding() methods.** Here's an overview of each:

1. **Rolling Window Functions (df.rolling()):**

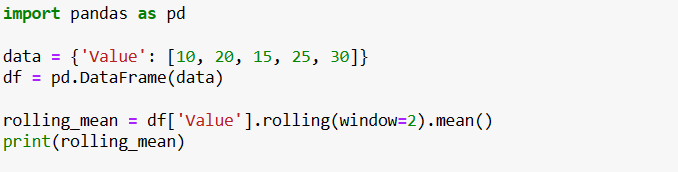
Rolling window functions operate over a sliding window of rows defined by a fixed-size or variable-size window.

* **Syntax:**

**rolling\_obj = df['column'].rolling(window=window\_size)**

**result = rolling\_obj.<aggregation\_function>()**

* **Parameters:**
* window\_size: Specifies the number of rows in the rolling window.
* **Example - Calculating Rolling Mean:**
* **Code:**

****

* **Output:**

****

1. **Expanding Window Functions (df.expanding()):**

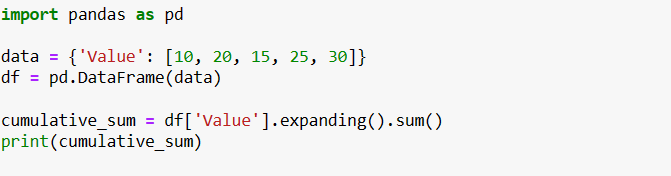
Expanding window functions calculate an aggregate value for all rows from the start of the DataFrame up to the current row.

* **Syntax:**

**expanding\_obj = df['column'].expanding()**

**result = expanding\_obj.<aggregation\_function>()**

* **Example -** Calculating Cumulative Sum:
* **Code:**

****

* **Output:**

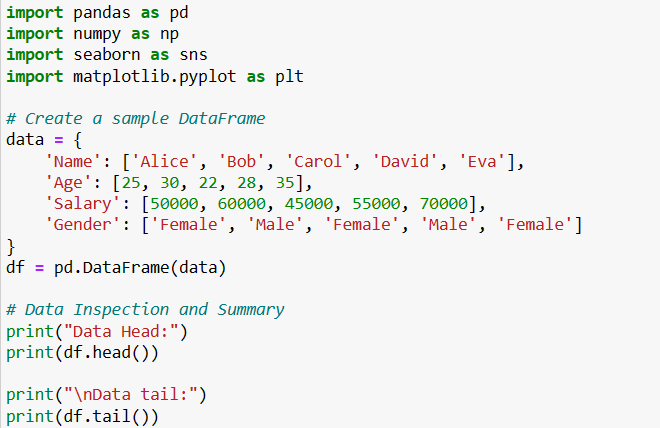
****

* **Common Aggregation Functions:**
* **.sum():** Calculate the sum of values in the window.
* **.mean():** Calculate the mean of values in the window.
* **.min(), .max():** Calculate the minimum or maximum value in the window.
* **.std(), .var():** Calculate the standard deviation or variance of values in the window.
* **.count():** Count the number of non-null values in the window.
* **Exploratory Data Analysis (EDA)**

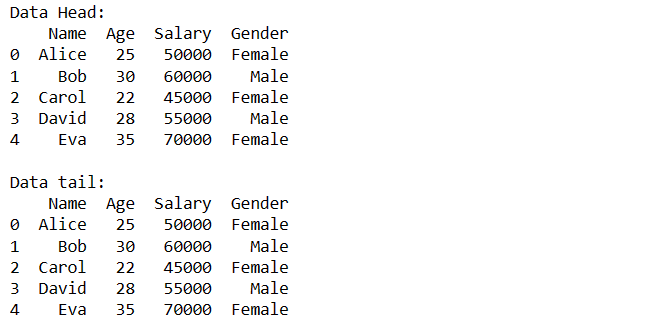
Exploratory Data Analysis (EDA) involves examining and understanding your data to uncover patterns, relationships, and insights. Pandas offers various functions that are commonly used in EDA to explore and analyze datasets. Here are some commonly used functions for EDA in pandas:

1. **Data Inspection:**

* **df.head(), df.tail():** Display the first or last few rows of the DataFrame.
* **Example:**
* **Code:**

****

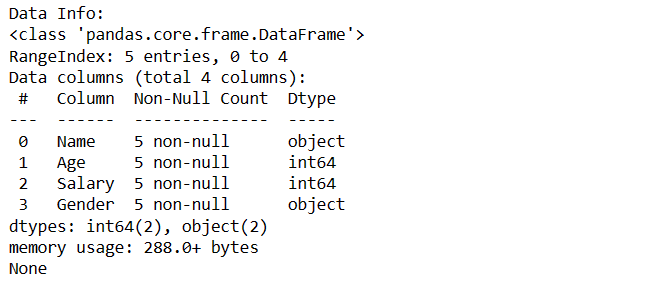
* **Output:**

****

* **df.info():** Display data types, non-null counts, and memory usage.
* **Code:**



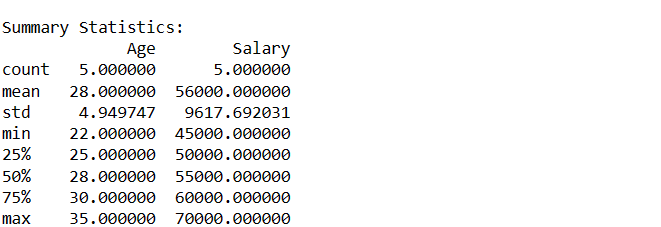
* **Output:**

****

* **df.describe():** Generate summary statistics for numeric columns.
* **Code:**



* **Output:**

****

* **df.shape:** Get the dimensions (rows, columns) of the DataFrame.
* **Code:**



* **Output:**

****

* **df.columns:** Get the column names of the DataFrame.
* **Code:**

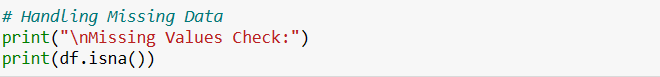


* **Output:**

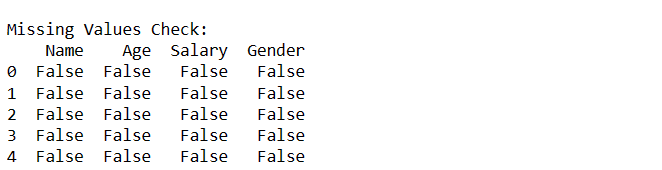
****

1. **Handling Missing Data:**

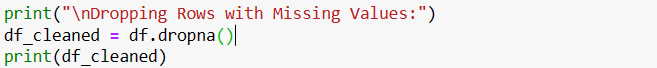
* **df.isna(), df.isnull():** Check for missing values.
* **Code:**



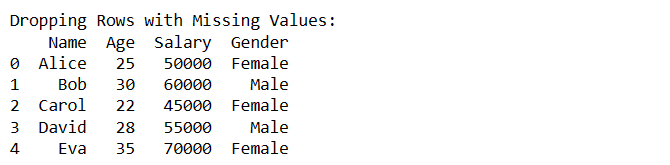
* **Output:**

****

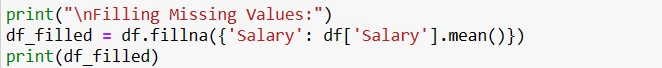
* **df.dropna():** Remove rows or columns with missing values.
* **Code:**



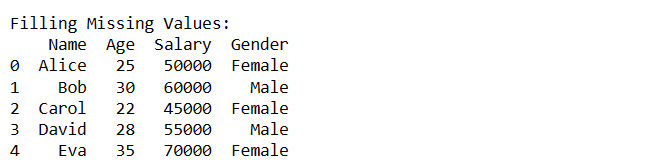
* **Output:**

****

* **df.fillna():** Replace missing values with specified values.
* **Code:**

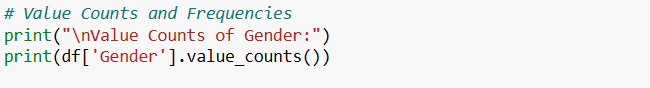


* **Output:**

****

1. **Value Counts and Frequencies:**

* **df['column'].value\_counts():** Count unique values in a column.
* **Code:**



* **Output:**

****