Data Analytics with Pandas

Introduction to Pandas:

- Pandas is a powerful Python library used for data manipulation and analysis.
- It provides high-performance, easy-to-use data structures such as DataFrame and Series.
- Pandas is widely used in data science and analytics for tasks like data cleaning, exploration, and transformation.

Key Concepts:

1. Data Structures:
    - Series: A one-dimensional labeled array that can hold any data type.
    - DataFrame: A two-dimensional table with labeled axes (rows and columns) that can hold heterogeneous data types.
2. Data Loading:
    - Reading data from various file formats: CSV, Excel, JSON, etc.
    - Exploring the data using functions like **head()**, **tail()**, and **describe()**.
3. Data Cleaning:
    - Handling missing values: detecting, removing, or imputing missing data.
    - Handling duplicates: identifying and removing duplicate rows.
    - Data type conversion: converting data types to the appropriate format.
4. Data Selection and Filtering:
    - Selecting columns and rows using indexing and slicing.
    - Conditional selection using boolean indexing.
    - Filtering rows based on specific conditions using functions like **loc()** and **iloc()**.
5. Data Manipulation:
    - Adding, updating, and deleting columns.
    - Renaming columns and changing column data types.
    - Sorting data based on one or multiple columns.
6. Data Aggregation and Grouping:
    - Computing summary statistics using functions like **mean()**, **sum()**, **count()**, etc.
    - Grouping data based on one or more columns and applying aggregation functions.
7. Data Visualization:
    - Using pandas' integration with Matplotlib or Seaborn to create visualizations.

- Plotting data using line plots, bar plots, scatter plots, etc.
- Customizing visualizations with labels, titles, legends, etc.

8. Data Analysis Techniques:
    - Exploratory Data Analysis (EDA): summarizing data, identifying patterns, and visualizing distributions.
    - Data Transformation: applying mathematical operations, scaling data, or creating new features.
    - Data Merging and Joining: combining multiple data sources based on common columns.

Title: DataFrames Creation and Related Functions

1. Creating DataFrames:
    - Creating an empty DataFrame:
        - **df = pd.DataFrame()**
    - Creating a DataFrame from a dictionary:
        - **data = {'Name': ['John', 'Jane', 'Tom'], 'Age': [25, 30, 35], 'City': ['New York', 'London', 'Paris']}**
        - **df = pd.DataFrame(data)**
    - Creating a DataFrame from a list of lists:
        - **data = [['John', 25, 'New York'], ['Jane', 30, 'London'], ['Tom', 35, 'Paris']]**
        - **df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])**
    - Creating a DataFrame from a CSV file:
        - **df = pd.read_csv('filename.csv')**
2. DataFrame Exploration:
    - Displaying the first few rows of a DataFrame:
        - **df.head()**
    - Displaying the last few rows of a DataFrame:
        - **df.tail()**
    - Getting information about the DataFrame:
        - **df.info()**
    - Describing the statistical summary of the DataFrame:
        - **df.describe()**
3. DataFrame Operations:

- Accessing columns:
    - **df['column_name']** or **df.column_name**
- Accessing rows by index:
    - **df.loc[row_index]** or **df.iloc[row_index]**
- Accessing a subset of rows and columns:
    - **df.loc[row_index, column_name]** or **df.iloc[row_index, column_index]**
- Adding a new column:
    - **df['new_column'] = values**
- Renaming columns:
    - **df.rename(columns={'old_name': 'new_name'}, inplace=True)**
- Dropping columns or rows:
    - **df.drop('column_name', axis=1, inplace=True)** (column)
    - **df.drop(row_index, axis=0, inplace=True)** (row)

4. Data Manipulation:
    - Filtering rows based on conditions:
        - **df[df['column'] > value]**
    - Sorting the DataFrame:
        - **df.sort_values('column_name', ascending=True)**
    - Grouping data by a column and performing aggregations:
        - **df.groupby('column_name').agg({'column_to_aggregate': 'function'})**
    - Applying functions to columns:
        - **df['new_column'] = df['column'].apply(function)**
5. Data Cleaning:
    - Handling missing values:
        - **df.isnull()** (checking for missing values)
        - **df.dropna()** (dropping rows with missing values)
        - **df.fillna(value)** (replacing missing values with a specific value)
    - Handling duplicates:
        - **df.duplicated()** (checking for duplicates)
        - **df.drop_duplicates()** (dropping duplicate rows)

Title: Aggregate Functions in Pandas

Aggregate functions in pandas allow us to compute summary statistics and perform aggregations on our data. These functions help us gain insights into the distribution, central tendency, and other characteristics of our datasets.

1. Common Aggregate Functions:
    - **mean()**: Computes the arithmetic mean (average) of the values in a column or the entire DataFrame.
        - Example: **df['column'].mean()**
    - **sum()**: Calculates the sum of the values in a column or the entire DataFrame.
        - Example: **df['column'].sum()**
    - **count()**: Counts the number of non-null values in a column or the entire DataFrame.
        - Example: **df['column'].count()**
    - **min()**: Finds the minimum value in a column or the entire DataFrame.
        - Example: **df['column'].min()**
    - **max()**: Finds the maximum value in a column or the entire DataFrame.
        - Example: **df['column'].max()**
    - **median()**: Computes the median (middle value) of the values in a column or the entire DataFrame.
        - Example: **df['column'].median()**
    - **std()**: Calculates the standard deviation of the values in a column or the entire DataFrame.
        - Example: **df['column'].std()**
    - **var()**: Computes the variance of the values in a column or the entire DataFrame.
        - Example: **df['column'].var()**
2. Aggregating by Groups:
    - Grouping data based on one or more columns and applying aggregations is a powerful feature in pandas.
    - The **groupby()** function is used to group the DataFrame based on specific columns.
        - Example: **df.groupby('grouping_column')**
    - Aggregating functions can then be applied to the grouped data to obtain statistics for each group.
        - Example: **df.groupby('grouping_column')['column'].mean()**

- Common aggregate functions can be applied to grouped data, such as **mean()**, **sum()**, **count()**, etc.

3. Aggregating with Custom Functions:
    - In addition to the built-in aggregate functions, pandas allows us to apply custom functions to our data using the **agg()** function.
    - The **agg()** function accepts a dictionary that maps columns to the desired aggregation functions.
        - Example: **df.groupby('grouping_column').agg({'column1': 'function1', 'column2': 'function2'})**
    - Custom functions can be defined using lambda expressions or regular functions to perform specific calculations or transformations.
4. Additional Aggregate Functions:
    - **quantile()**: Computes the specified quantile of the values in a column or the entire DataFrame.
        - Example: **df['column'].quantile(0.75)** (returns the 75th percentile)
    - **value_counts()**: Counts the frequency of unique values in a column.
        - Example: **df['column'].value_counts()**
    - **nunique()**: Counts the number of unique values in a column.
        - Example: **df['column'].nunique()**

Title: Join, Merge, and Concatenating DataFrames

1. Concatenating DataFrames:
    - Concatenation is the process of combining DataFrames along a particular axis (rows or columns).
    - The **concat()** function in pandas is used for concatenating DataFrames.
        - Syntax: **pd.concat([df1, df2, df3], axis=0)** (concatenating vertically)
        - Syntax: **pd.concat([df1, df2, df3], axis=1)** (concatenating horizontally)
    - When concatenating DataFrames, they are stacked on top of each other (vertically) or side by side (horizontally) based on the axis specified.
    - By default, concatenation preserves the original indices of the DataFrames. To create a new index, use the **ignore_index=True** parameter.
2. Joining DataFrames:
    - Joining combines two DataFrames based on a common column between them.
    - The **join()** function is used to perform joins in pandas.
        - Syntax: **df1.join(df2, on='column_name', how='join_type')**

- Common join types:
    - Inner join (**how='inner'**): Keeps only the matching records from both DataFrames.
    - Left join (**how='left'**): Keeps all records from the left DataFrame and the matching records from the right DataFrame.
    - Right join (**how='right'**): Keeps all records from the right DataFrame and the matching records from the left DataFrame.
    - Outer join (**how='outer'**): Keeps all records from both DataFrames.
- The **on** parameter specifies the common column used for joining the DataFrames. If the column names differ, use **left_on** and **right_on** to specify the respective columns.

3. Merging DataFrames:
    - Merging is similar to joining, but it can be performed based on multiple columns.
    - The **merge()** function is used for merging DataFrames.
        - Syntax: **pd.merge(df1, df2, on=['column1', 'column2'], how='merge_type')**
    - Common merge types:
        - Inner merge (**how='inner'**): Keeps only the matching records based on all specified columns.
        - Left merge (**how='left'**): Keeps all records from the left DataFrame and the matching records based on all specified columns from the right DataFrame.
        - Right merge (**how='right'**): Keeps all records from the right DataFrame and the matching records based on all specified columns from the left DataFrame.
        - Outer merge (**how='outer'**): Keeps all records from both DataFrames based on all specified columns.
    - The **on** parameter takes a list of columns on which to merge the DataFrames.
4. Handling Duplicate Columns:
    - When merging or concatenating DataFrames, duplicate column names may occur. To handle this, use suffixes to differentiate the columns.
        - Example: **pd.merge(df1, df2, on='column', suffixes=('_left', '_right'))**
    - This will append the specified suffixes to the duplicate column names to make them unique.
5. Handling Missing Values:
    - When merging or joining DataFrames, missing values may occur when there are no matches.
    - By default, missing values are represented as NaN (Not a Number).

- To handle missing values, use the **how** parameter to specify the merge or join type that suits your requirements.

Title: Changing Data Types of a Column in Pandas

1. Understanding Data Types:
    - Data types determine how data is stored and interpreted by the computer.
    - Pandas provides various data types, such as integer, float, string, datetime, etc., to handle different types of data.
2. Checking Data Types:
    - To check the data type of a column in a DataFrame, use the **dtypes** attribute.
        - Syntax: **df['column'].dtypes**
3. Changing Data Types:
    - Pandas provides several methods to change the data type of a column based on the desired type.
    - Using the **astype()** function:
        - The **astype()** function converts a column to a specified data type.
            - Syntax: **df['column'] = df['column'].astype('desired_data_type')**
    - Converting to Numeric Data Type:
        - To convert a column to a numeric data type (integer or float), use the **pd.to_numeric()** function.
            - Syntax: **df['column'] = pd.to_numeric(df['column'], errors='coerce')**
            - The **errors='coerce'** parameter handles non-convertible values by setting them as NaN.
    - Converting to DateTime Data Type:
        - To convert a column to a DateTime data type, use the **pd.to_datetime()** function.
            - Syntax: **df['column'] = pd.to_datetime(df['column'])**
    - Converting to Categorical Data Type:
        - To convert a column to a categorical data type, use the **astype()** function or the **pd.Categorical()** constructor.
            - Syntax: **df['column'] = df['column'].astype('category')**
            - Alternative Syntax: **df['column'] = pd.Categorical(df['column'])**
    - Using Custom Conversion Functions:
        - If the desired data type cannot be achieved with the above methods, custom conversion functions can be used.

- Define a function that performs the necessary conversions and apply it to the column using the **apply()** function.
    - Syntax: **df['column'] = df['column'].apply(custom_conversion_function)**

4. Handling Conversion Errors:
    - During conversion, errors may occur due to incompatible values in the column.
    - The **errors** parameter in the conversion functions can handle these errors:
        - **errors='coerce'**: Invalid values are converted to NaN.
        - **errors='ignore'**: Invalid values remain unchanged.
5. Handling Missing Values:
    - After changing the data type of a column, missing values may be represented as NaN.
    - To handle missing values, use appropriate methods such as **fillna()** or **dropna()**.
6. Verifying Data Type Changes:
    - To verify the data type changes, use the **dtypes** attribute or the **info()** function of the DataFrame.
        - Syntax: **df.dtypes** or **df.info()**
7. Considerations:
    - Changing data types may result in data loss or unexpected behavior, so ensure the conversion is appropriate for the data.
    - It is recommended to perform data type changes after handling missing values and outliers.

Title: Data Transformation in Pandas

Data transformation involves modifying and reorganizing data to make it suitable for analysis. Pandas provides several functions and methods to perform various data transformation operations. Here are some important notes on data transformation in pandas:

1. Removing Duplicates:
    - To remove duplicate rows from a DataFrame, use the **drop_duplicates()** method.
        - Syntax: **df.drop_duplicates()**
2. Filtering Rows:
    - Filtering allows you to extract specific rows from a DataFrame based on certain conditions.
    - Use boolean indexing to filter rows that meet specific criteria.
        - Syntax: **df[df['column'] condition]**

3. Handling Missing Values:
    - Missing values can hinder data analysis. Pandas provides functions to handle missing values effectively.
    - To check for missing values, use the **isnull()** function or **notnull()** function.
        - Syntax: **df.isnull()**, **df.notnull()**
    - To drop rows or columns with missing values, use the **dropna()** method.
        - Syntax: **df.dropna()**
    - To fill missing values with a specified value, use the **fillna()** method.
        - Syntax: **df.fillna(value)**
4. Applying Functions:
    - Pandas provides **apply()** and **applymap()** functions to apply custom functions to DataFrame or Series elements.
    - **apply()** function:
        - Applies a function to each column or row of a DataFrame.
        - Syntax: **df.apply(function, axis=0)** (axis=0 applies function to columns, axis=1 applies function to rows)
    - **applymap()** function:
        - Applies a function element-wise to each element of a DataFrame.
        - Syntax: **df.applymap(function)**
5. Replacing Values:
    - To replace specific values in a DataFrame, use the **replace()** method.
        - Syntax: **df.replace(to_replace, value)**
    - To replace values based on conditions, use boolean indexing.
        - Syntax: **df.loc[condition, 'column'] = new_value**
6. Sorting Data:
    - Sorting helps arrange data in a specific order based on column values.
    - To sort a DataFrame by one or more columns, use the **sort_values()** method.
        - Syntax: **df.sort_values(by=['column1', 'column2'], ascending=[True, False])**
7. Grouping and Aggregating:
    - Grouping data allows you to apply functions to subsets of data based on a specific column or set of columns.
    - Use the **groupby()** function to group data and apply aggregations using functions like **sum()**, **mean()**, **count()**, etc.

8. Reshaping Data:
    - Pandas provides functions to reshape data from wide to long or long to wide formats.
    - **melt()** function: Converts wide-format data to long-format.
        - Syntax: **pd.melt(df, id_vars=['column1', 'column2'], value_vars=['column3', 'column4'], var_name='new_column', value_name='new_value')**
    - **pivot()** function: Converts long-format data to wide-format.
        - Syntax: **df.pivot(index='index_column', columns='column_to_pivot', values='values_column')**

Title: Filtering Data with the Filter Function in Pandas

The filter function in pandas allows you to select specific columns or rows from a DataFrame based on a set of criteria. It provides a convenient way to filter and extract the desired subset of data. Here are some important notes about the filter function in pandas:

1. Syntax:
    - The filter function is applied to a DataFrame and takes two parameters:
        - **items**: Specifies the columns to include in the filtered DataFrame. It can be a list of column names or a regular expression pattern.
        - **like**: Specifies the substring to match against column names for inclusion in the filtered DataFrame.
2. Filtering Columns:
    - To filter columns based on a list of column names, use the **items** parameter.
        - Syntax: **df.filter(items=['column1', 'column2', ...])**
    - You can also use a regular expression pattern to filter columns that match a specific pattern.
        - Syntax: **df.filter(items='pattern')**
    - The **items** parameter supports both exact matches and partial string matches.
3. Filtering Columns with **like** Parameter:
    - The **like** parameter allows you to filter columns based on a substring present in their names.
    - It supports partial string matches and is case-sensitive.
        - Syntax: **df.filter(like='substring')**
    - The **like** parameter is useful when you want to select columns that share a common naming pattern.
4. Filtering Rows:

- In addition to filtering columns, the filter function can be used to filter rows based on conditions.
- To filter rows based on a condition, use boolean indexing with the filter function.
    - Syntax: **df.filter(condition)**
- The condition can be any logical expression that evaluates to a boolean Series, indicating which rows to include in the filtered DataFrame.

5. Filtering Rows and Columns Simultaneously:
    - You can combine column and row filtering by chaining the filter function.
        - Syntax: **df.filter(items=['column1', 'column2']).filter(condition)**
    - This allows you to filter both columns and rows in a single operation.
6. Limitations of the Filter Function:
    - The filter function works only on column labels, not on the contents of the DataFrame.
    - It is primarily designed for simple column filtering and should not be used for complex data manipulations.
7. Alternative Methods:
    - For more complex filtering operations or conditional selection based on values, consider using boolean indexing, loc/iloc, query, or other pandas methods.

The filter function in pandas provides a convenient way to filter columns and rows based on specific criteria. It is particularly useful when you want to quickly extract a subset of data from a DataFrame.

Title: Saving a Pandas DataFrame

Saving a Pandas DataFrame allows you to store the data in various file formats for future use or sharing with others. Pandas provides several methods to save DataFrames in different formats. Here are notes on different ways to save a Pandas DataFrame:

1. CSV (Comma-Separated Values) Format:
    - The CSV format is widely used for storing tabular data in a plain-text format.
    - To save a DataFrame as a CSV file, use the **to_csv()** method.
        - Syntax: **df.to_csv('filename.csv', index=False)**
    - The **index=False** parameter excludes the index from being saved as a separate column.
2. Excel Format:
    - Excel format is useful for storing data in spreadsheets with multiple sheets and formatting options.
    - To save a DataFrame as an Excel file, use the **to_excel()** method.

        - Syntax: **df.to_excel('filename.xlsx', sheet_name='Sheet1', index=False)**
    - The **sheet_name** parameter specifies the name of the sheet within the Excel file.
3. JSON (JavaScript Object Notation) Format:
    - JSON format is commonly used for storing structured data in a human-readable format.
    - To save a DataFrame as a JSON file, use the **to_json()** method.
        - Syntax: **df.to_json('filename.json', orient='records')**
    - The **orient='records'** parameter saves each row of the DataFrame as a separate JSON record.
4. Pickle Format:
    - Pickle is a Python-specific format that allows you to save any Python object, including DataFrames, in a binary format.
    - To save a DataFrame as a pickle file, use the **to_pickle()** method.
        - Syntax: **df.to_pickle('filename.pkl')**
    - Pickle files can be used to store and load DataFrames efficiently.
5. Parquet Format:
    - Parquet is a columnar storage file format designed for optimized reading and writing of large datasets.
    - To save a DataFrame as a Parquet file, use the **to_parquet()** method.
        - Syntax: **df.to_parquet('filename.parquet')**
    - Parquet files are particularly useful for big data and analytical workloads.
6. Other Formats:
    - Pandas also provides methods to save DataFrames in other formats such as HDF5, Feather, and SQL databases.
    - Refer to the pandas documentation for more details on saving DataFrames in these formats.
7. Compression:
    - For most file formats, you can apply compression to reduce the file size.
    - Specify the compression type using the **compression** parameter in the respective save method.
        - Example: **df.to_csv('filename.csv', index=False, compression='gzip')**
    - Common compression types include 'gzip', 'zip', 'xz', 'bz2', etc.

When saving a Pandas DataFrame, consider the format that best suits your needs in terms of compatibility, file size, and data structure. It's important to provide a meaningful filename and ensure the necessary dependencies are installed to handle specific file formats.