Pandas

1. What are Pandas?

Pandas is an open source Python library that is built on top of the NumPy library. It is designed for working with relational or labeled data, offering various data structures for manipulating, cleaning, and analyzing numerical data. One of its notable features is its ability to handle missing data effectively. With its focus on high performance and productivity, Pandas is widely used in data science, data analysis, and machine learning tasks. It provides a range of functionalities including data visualization, manipulation, and analysis, making it a powerful tool for data professionals.

2. What are the Different Types of Data Structures in Pandas?

Pandas offers three distinct data structures, contributing to its speed and efficiency.

Series: This one*dimensional array*like structure holds homogeneous data, allowing only one data type per series. While its values are mutable, the size remains immutable.

DataFrame: As a two dimensional array like structure, DataFrame accommodates heterogeneous data, organizing it in a tabular format. Both the size and values of DataFrame are mutable.

Panel: Although less commonly used, Pandas introduces Panel as a 3D data structure capable of storing heterogeneous data.

Among these, Pandas primarily supports Series and DataFrames. Series serves as a labeled one dimensional array suitable for representing a single column or row of data. Meanwhile, DataFrames act as two dimensional heterogeneous structures, organizing data into rows and columns within a tabular framework.

3. List Key Features of Pandas.

- DataFrame: Offers rapid and effective **data manipulation** with both **default and customized indexing**.
- Efficient **merging and joining capabilities** for high performance data operations.
- Seamlessly **aligns data** and **handles missing values**.
- Supports **label-based slicing**, **indexing**, and **subsetting**, even for large datasets.
- Facilitates **reshaping** and **pivoting** of datasets for streamlined analysis.
- Provides **versatile tools for loading data** from various file formats into memory.
- Allows for easy **deletion or insertion** of **columns** within data structures.
- Enables **grouping** of data for **aggregation** and **transformation** tasks.

- Incorporates functionality tailored for **time series analysis**.
- Overall, Pandas excels in efficient data analysis, boasting features such as fast manipulation and analysis, robust time series support, seamless missing data handling, rapid merging and joining, flexible reshaping and pivoting, powerful grouping capabilities, versatile data loading, and tight integration with NumPy.

4. What is Series in Pandas?

A Series in Pandas is a **one dimensional labeled array**, akin to a column in an Excel sheet. It can store **various data types** like integers, strings, or Python objects. The labels for each element are referred to as the **index**. Series holds **homogeneous data**, allowing for **mutable values** but **immutable size**. Creation of a Series involves using the ***pd.Series()*** function, which can convert Python **tuples, lists, or dictionaries into a Series**. It's important to note that a Series **cannot contain multiple columns**.

5. How can we Create a Copy of the Series?

A shallow copy of a Series in Pandas replicates the object without copying its indices and data. Instead, it merely duplicates references, causing changes in one to affect the other.

ser.copy(deep=False)

A deep copy of a Series in Pandas is an independent replica with its own indices and data. Changes made to the copy won't affect the original series.

ser.copy(deep=True)

6. What is a DataFrame in Pandas?

A DataFrame in Pandas is a versatile data structure designed for tabular data storage, organized into rows and columns. It's two-dimensional, meaning it has both rows and columns, and can adapt its size and content. DataFrames are heterogeneous, meaning they can accommodate various data types within a single structure. They're typically created by importing data from sources like SQL databases, CSV files, or Excel files. In essence, a DataFrame is like a table where the rows and columns represent the data, and it's constructed using the pd.DataFrame() function.

7. What is Reindexing in Pandas?

Reindexing in Pandas involves altering the row and column indices of a DataFrame. This process can be accomplished using the reindex() method provided by Pandas. When reindexing, if there are missing values or new values not present in the original DataFrame, the reindex() method assigns them as NaN.

8. How to add an Index, Row, or Column to an Existing Dataframe?

Adding Index :

We can add an index to an existing dataframe by using the Pandas ***set_index()*** method which is used to set a list, series, or dataframe as the index of a dataframe. The set_index() method has the following syntax:

df.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)

Adding Rows :

The ***df.loc[]*** is used to access a group of rows or columns and can be used to add a row to a dataframe.

DataFrame.loc[Row_Index]=new_row

We can also add multiple rows in a dataframe by using ***pandas.concat()*** function which takes a list of dataframes to be added together.

pandas.concat([Dataframe1,Dataframe2])

Adding Columns :

We can add a column to an existing dataframe by just declaring the column name and the list or dictionary of values.

DataFrame[data] = list_of_values

Another way to add a column is by using ***df.insert()*** method which takes a value where the column should be added, column name and the value of the column as parameters.

DataFrameName.insert(col_index, col_name, value)

We can also add a column to a dataframe by using ***df.assign()*** function

DataFrame.assign(kwargs)

9. How to Delete an Index, Row, or Column from an Existing DataFrame?

We can delete a row or a column from a dataframe by using the ***df.drop()*** method. and provide the row or column name as the parameter.

To delete a column :

DataFrame.drop(['Column_Name'], axis=1)

To delete a row :

DataFrame.drop([Row_Index_Number], axis=0)

10. Explain Categorical data in Pandas?

Categorical data is a **discrete set of values** for a particular outcome and has a **fixed range**. Also, the data in the category need not be numerical, it can be textual in nature. Examples are *gender, social class, blood type, country affiliation, observation time*, etc. There is no hard and fast rule for how many values a categorical value should have. One should apply one's domain knowledge to make that determination on the data sets.

11. Give a brief description of time series in Pandas

A time series is an organized collection of data that **depicts the evolution of a quantity through time**. Pandas have a wide range of capabilities and tools for working with time*series data in all fields.

Supported by pandas:

- Analyzing time*series data from a variety of sources and formats.
- Create time and date sequences with preset frequencies.
- Date and time manipulation and conversion with timezone information.
- A time series is resampled or converted to a specific frequency.
- Calculating dates and times using absolute or relative time increments is one way to.

12. Explain MultiIndexing in Pandas.

Multiple indexing is defined as essential indexing because it deals with data analysis and manipulation, especially for working with higher dimensional data. It also enables us to store and manipulate data with an arbitrary number of dimensions in lower*dimensional data structures like Series and DataFrame.

13. How can we convert DataFrame to an excel file?

To convert a DataFrame to an Excel file, use the **to_excel()** function. Specify DataFrame name and sheet name. For multiple sheets, use ExcelWriter object. Syntax: **data.to_excel(excel_writer, sheet_name='Sheet1', **kwargs)**. Parameters: ***excel_writer*** (string or ExcelWriter object), ***sheet_name*** (string, default='Sheet1'), ***columns*** (sequence or list of strings, optional), ***index*** (boolean, default=True), ***index_label*** (string or sequence of strings, optional).

14. What is TimeDelta?

Timedeltas are differences in times, expressed in different units, e.g. days, hours, minutes, and seconds. They can be both positive and negative.

15. Explain Pandas Timedelta.seconds Property

Timedelta.seconds in pandas is used to return the number of seconds. Its implementation is simpler than it sounds. We do not need any special parameters and the return type is in the form of seconds.

16. How to Iterate over Dataframe in Pandas?

There are various ways to iterate the rows and columns of a dataframe.
To iterate over rows in a DataFrame, use iterrows() which returns each index value with a series containing row data. Alternatively, iteritems() iterates over columns as key*value pairs. itertuples() returns tuples for each row, where the first element is the index and the rest are row values.
For column iteration, create a list of DataFrame columns using the list constructor.

17. What are the Important Conditions to keep in mind before Iterating?

Iterating is not the best option when it comes to Pandas Dataframe. Pandas provides a lot of functions using which we can perform certain operations instead of iterating through the dataframe. While iterating a dataframe, we need to keep in mind the following things:

- While printing the data frame, instead of iterating, we can use DataFrame.to_string() methods which will display the data in tabular form.
- If we are concerned about time performance, iteration is not a good option. Instead, we should choose vectorization as pandas have a number of highly optimized and efficient built*in methods.
- We should use the apply() method instead of iteration when there is an operation to be applied to a few rows and not the whole database.

18. Is iterating over a Pandas Dataframe a good practice? If not, what are the important conditions to keep in mind before iterating?

Iterating over a Pandas DataFrame should be avoided whenever possible due to its inefficiency. It's preferable to use built*in Pandas functions for operations. However, if iteration is necessary, certain conditions should be considered:

1. Applying a function to rows: Instead of iterating through rows, use the "apply()" method for efficiency.
2. Iterative manipulations: For performance*critical iterative tasks, consider alternatives like numba or cython.
3. Printing a DataFrame: Use the "to_string()" method to print a DataFrame efficiently.
4. Vectorization over iteration: Opt for vectorized operations as Pandas offers optimized built*in methods for efficient data manipulation.

19. How would you iterate over rows in a DataFrame in Pandas?

Although it is not a good practice to iterate over rows in Pandas if there is no other alternative we do so using either **iterrows()** or **itertuples()** built*in methods.

pandas.DataFrame.iterrows(): This method is used to iterate over DataFrame rows as (index, Series) pairs. There is only one drawback for this method: it does not preserve the dtypes across rows due to the fact that it converts each row into a Series. If you need to preserve the dtypes of the pandas object, then one should use itertuples() method instead.

```
# import pandas package as pd
import pandas as pd

# Define a dictionary containing students data
data = {'Name': ['Sneha', 'Shreya','Sabhya', 'Riya'],
'Age': [22, 18, 10, 19],
'Stream': ['Computer', 'Commerce','Arts', 'Mechanical'],
'Percentage': [89, 93, 97, 73]}
```

```
# Convert the dictionary into DataFrame
df = pd.DataFrame(data, columns=['Name', 'Age','Stream', 'Percentage'])

print("Given Dataframe :\n", df)

print("\nIterating over rows using iterrows() method :\n")

# iterate through each row and select
# 'Name' and 'Age' columns respectively.
for index, row in df.iterrows():
print(row["Name"], row["Age"])
```

pandas.DataFrame.itertuples(): This method is used to iterate over DataFrame rows as namedtuples. Also, itertuples() are faster compared to iterrows().

```
print("\nIterating over rows using itertuples() method :\n")

# iterate through each row and select
# 'Name' and 'Percentage' column respectively.
for row in df.itertuples(index=True, name='Pandas'):
print(getattr(row, "Name"), getattr(row, "Percentage"))
```

20. List some statistical functions in Python Pandas?

1. **"sum()":** Returns the sum of values.
2. **"min()":** Returns the minimum value.
3. **"max()":** Returns the maximum value.
4. **"abs()":** Returns the absolute value.
5. **"mean()":** Returns the mean (average) of values.
6. **"std()":** Returns the standard deviation.
7. **"prod()":** Returns the product of values.
8. **"count()":** Returns the number of non*null observations.
9. **"median()":** Returns the median value.
10. **"mode()":** Returns the mode value(s).
11. **"var()":** Returns the variance.
12. **"quantile()":** Returns the quantile value.
13. **"corr()":** Computes pairwise correlation of columns.
14. **"cov()":** Computes covariance between columns.
15. **"describe()":** Generates descriptive statistics.
16. **"kurtosis()":** Computes the kurtosis (tailedness) of a data distribution.
17. **"skew()":** Computes the skewness (asymmetry) of a data distribution.
18. **"cumsum()":** Computes the cumulative sum of values.
19. **"cummin()":** Computes the cumulative minimum of values.
20. **"cummax()":** Computes the cumulative maximum of values.
21. **"cumprod()":** Computes the cumulative product of values.

21. How to Read Text Files with Pandas?

We can read a text file in different ways using Pandas:

1. **read_csv():** This is used for comma separated files (CSV), where commas separate each field. We use `read_csv()` to load data from a text file.
2. **read_table():** Similar to `read_csv()`, but it uses a different delimiter, which is '\t' (tab) by default. We can specify other delimiters as well. For instance, we can read data with `read_table()` by setting the separator to a single space (' ').
3. **read_fwf():** This function is for fixed width files, where columns have a fixed width. It effectively reads the contents into separate columns, even when there are no delimiters. Additionally, it supports iterating or breaking the file into chunks for large datasets.

22. How are iloc() and loc() different?

- **"iloc()":**
    1. Selects data based on integer indices.
    2. Syntax: `dataframe.iloc[row_index, column_index]`
    3. Follows Python convention of excluding end value.
    4. Used for position*based selection regardless of labels.
- **"loc()":**
    1. Selects data based on label indices.
    2. Syntax: `dataframe.loc[row_label, column_label]`
    3. Includes both start and end labels in selection.
    4. Used for label*based selection in DataFrame.

23. How will you sort a DataFrame?

The sorting function in Pandas is **DataFrame.sort_values()**. It arranges DataFrame by column or row values. Key parameters include ***'by'*** to specify sorting criteria (optional), ***'axis'*** to indicate row (0) or column (1) sorting, and ***'ascending'*** to determine sorting order (default is ascending). Setting ascending=False sorts in descending order.

24. How would you convert continuous values into discrete values in Pandas?

Continuous values can be discretized using two functions:

1. **cut():** Segments and sorts data into evenly spaced bins based on values. Useful for creating groups of data values such as age ranges.
2. **qcut():** Bins data based on sample quantiles, ensuring an equal number of records in each bin. Helpful for studying data by quantiles, like computing quintiles.

25. What is the difference between join() and merge() in Pandas?

The main difference between df.join() and df.merge() lies in their approach to combining dataframes:

1. **Join vs. Merge:**
    - Join: Combines based on indexes of dataframes.
    - Merge: More flexible, allowing specification of columns along with indexes.
2. **Lookup Approach:**
    - Right Table Lookup: Join() uses df2 index; merge() allows choice of columns or index of df2.
    - Left Table Lookup: Join() uses df1 index by default; merge() uses df1 column(s), unless specified otherwise.
3. **Left vs. Inner Join:**
    - Join: Defaults to left join, retaining all rows of df1.
    - Merge: Defaults to inner join, returning only matching rows of df1 and df2.

26. What is the difference(s) between merge() and concat() in Pandas?

The distinction between merge() and concat() in Pandas lies in several key aspects:

1. **Combination Approach:**

- concat(): Stacks dataframes along rows or columns, essentially combining them.
- merge(): Combines dataframes based on shared column values, offering more flexibility as it merges based on specified conditions.

2. **Axis Parameter:**

- concat(): Requires an axis parameter to determine the direction of concatenation (0 for row wise, 1 for column wise).
- merge(): Doesn't necessitate an axis parameter as it combines dataframes side by side based on shared columns.

3. **Join vs. How:**

- Join (concat()): Dictates the type of join to perform (outer or inner).
- How (merge()): Specifies the merge method, including inner, outer, left, and right joins.

27. What's the difference between interpolate() and fillna() in Pandas?

The distinction between interpolate() and fillna() in Pandas lies in their approaches to handling missing values:

1. **interpolate():**
    - Usage: Estimates missing values by interpolating between existing data points.

- Method: Uses mathematical interpolation techniques like linear, polynomial, or spline to fill in missing values based on neighboring data points.
- Application: Useful for timeseries data or datasets with regularly spaced intervals.

2. **fillna():**
   - Usage: Fills missing values with specified constant or method.
   - Method: Simply replaces missing values with the specified constant or by using a predetermined method like 'ffill' (forward fill) or 'bfill' (backward fill).
   - Application: Suitable for general purpose data cleaning where missing values are replaced with specific values or methods.

28. What is Time Delta in Pandas?

The time delta is the difference in dates and time. Similar to the timedelta() object in the datetime module, a Timedelta in Pandas indicates the duration or difference in time. For addressing time durations or time variations in a DataFrame or Series, Pandas has a dedicated data type.

29. What is Data Aggregation in Pandas?

In Pandas, data aggregation refers to the act of **summarizing or decreasing data in order to produce a consolidated view or summary statistics** of one or more columns in a dataset. In order to calculate statistical measures like ***sum, mean, minimum, maximum, count***, etc., aggregation functions must be applied to groups or subsets of data.
The agg() function in Pandas is frequently used to aggregate data. Applying one or more aggregation functions to one or more columns in a DataFrame or Series is possible using this approach. Pandas' built*in functions or specially created user*defined functions can be used as aggregation functions.

DataFrame.agg({'Col_name1': ['sum', 'min', 'max'], 'Col_name2': 'count'})

30. Difference between map(), applymap(), and apply()

Here's a clearer breakdown of the differences between map(), applymap(), and apply():

1. **map():**
   - **Defined in:** Series
   - **Usage:** Applies a function or a dictionary to each element of the Series.
   - **Operation:** Works element*wise, enabling simple transformations or mappings on Series elements.
2. **applymap():**
   - **Defined in:** DataFrame
   - **Usage:** Applies a function to each element of the DataFrame.
   - **Operation:** Works element*wise, applying the provided function to each element in the DataFrame.

3. **apply():**
    - **Defined in:** Both Series and DataFrame
    - **Usage:** Applies a function along a specific axis of the DataFrame or Series.
    - **Operation:** Works on either entire rows or columns, element*wise, allowing aggregation or transformation across rows or columns.

31. Difference between pivot_table() and groupby()

Here's a concise comparison between pivot_table() and groupby():

1. **pivot_table():**
    - **Operation**: Summarizes and aggregates data in a tabular format, transforming it based on column values.
    - **Usage**: Used when comparing data across multiple dimensions.
    - **Flexibility**: Can handle multiple levels of grouping and aggregation, offering flexibility in summarizing data.
2. **groupby():**
    - **Operation**: Performs aggregation on grouped data of one or more columns, summarizing data within groups.
    - **Usage**: Used to group data based on categorical variables, followed by applying aggregation functions to the grouped data.
    - **Result**: Creates a GroupBy object, allowing various aggregation functions (e.g., sum, mean, count) to be applied to the grouped data.

32. How can we use Pivot and Melt Data in Pandas?

To **pivot** a DataFrame in Pandas, we utilize the **pivot_table()** method.
To **revert** the DataFrame to its original form, we can employ the **melt()** method.

33. How to make Label Encoding using Pandas?

Label encoding is used to convert categorical data into numerical data so that a machine-learning model can fit it. To apply label encoding using pandas we can use the **pandas.categorical().codes** or **pandas.factorize()** method to replace the categorical values with numerical values.

34. How to Handle Missing Data in Pandas?

Generally a dataset has some missing values, and it can happen for a variety of reasons, such as data collection issues, data entry errors, or data not being available for certain observations. This can cause a big problem. To handle these missing values Pandas provides various functions. These functions are used for detecting, removing, and replacing null values in Pandas DataFrame:

- **isnull():** It returns True for NaN values or null values and False for present values
- **notnull():** It returns False for NaN values and True for present values
- **dropna():** It analyzes and drops Rows/Columns with Null values

- **fillna():** It let the user replace NaN values with some value of their own
- **replace():** It is used to replace a string, regex, list, dictionary, series, number, etc.
- **interpolate():** It fills NA values in the dataframe or series.

35. How to Create a New Column Based on Existing Columns?
We can create a column from an existing column in a DataFrame by using the **df.apply()** and **df.map()** functions.

36. Which three main objects do pandas have?
The Three fundamental objects around which the whole pandas function revolves around are **Series, DataFrame ,** and **Index.**

37. Why does everyone use pandas?
Pandas allow a wide range of data manipulation operations such as **merging**, **reshaping**, **selecting**, as well as **data cleaning**, and **data wrangling features**. Apart from that, Pandas is very compatible with file-handling operations such as importing data from various file formats such as ***comma-separated values, JSON, Parquet, SQL database tables or queries, and Microsoft Excel***.

38. What is all () in pandas?
DataFrame.all() method checks whether all elements are True, potentially over an axis. It returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

39. How can you efficiently handle large datasets in Pandas?
To handle large datasets efficiently in Pandas, you can use techniques like chunking, selective loading of columns, optimizing data types, and using functions like `read_csv()` with parameters like `**chunksize**` and `**dtype**`.

40. Explain the difference between iloc() and loc() in Pandas.
`iloc()` is used for integer-location based indexing, while `loc()` is used for label-based indexing. `iloc[]` selects data based on integer indices, while `loc[]` selects data based on labels.

41. How can you efficiently perform group-wise operations on large datasets in Pandas?
For efficient group-wise operations on large datasets, you can use the `groupby()` function in Pandas, which groups data based on specified columns and allows applying aggregation functions using `agg()` or `apply()`.

42. What is the purpose of the `apply()` function in Pandas?

The `apply()` function in Pandas is used to apply a function along an axis of a DataFrame or Series. It is often used to apply custom functions or operations that cannot be directly applied using built-in Pandas functions.

43. How do you handle duplicate values in a DataFrame in Pandas?
To handle duplicate values in a DataFrame in Pandas, you can use methods like `duplicated()` to identify duplicate rows and `drop_duplicates()` to remove duplicate rows from the DataFrame.

44. Describe the purpose of the `stack()` and `unstack()` functions in Pandas.
The `stack()` function in Pandas is used to pivot a level of the column labels, returning a DataFrame with a new level of row labels. The `unstack()` function is used to pivot a level of the row index labels, returning a DataFrame with a new level of column labels.

45. How can you efficiently iterate over rows in a DataFrame in Pandas?
To efficiently iterate over rows in a DataFrame in Pandas, you can use techniques like using vectorized operations instead of loops, or using functions like `iterrows()` or `apply()` with appropriate parameters.

46. Explain the purpose of the `nlargest()` and `nsmallest()` functions in Pandas.
The **`nlargest()`** and **`nsmallest()`** functions in Pandas are used to return the n largest or smallest elements from a DataFrame or Series, respectively. They are often used to quickly identify top or bottom records based on specified criteria.

47. What is the purpose of the `pipe()` function in Pandas?
The **`pipe()`** function in Pandas is used to apply a function to a DataFrame or Series in a chainable manner. It allows for more readable and concise code by enabling function chaining.

48. How to get the items not common to both series A and series B?
To get the items not common to both series A and series B in Python, use the symmetric difference operation:

```
result = series_A.symmetric_difference(series_B)
```

49. How to get the minimum, 25th percentile, median, 75th, and max of a numeric series?
directly use the *quantile* method to get the desired percentiles