Pandas- Answers

Section – A

1. Pandas is an open-source Python library that is built on top of the NumPy library. It is made for working with relational or labelled data. It provides various data structures for manipulating, cleaning and analyzing numerical data. It can easily handle missing data as well. Pandas are fast and have high performance and productivity.

2. The two data structures that are supported by Pandas are **Series** and **DataFrames**.
   Pandas Series is a one-dimensional labelled array that can hold data of any type. It is mostly used to represent a single column or row of data.
   Pandas Dataframe is a two-dimensional heterogeneous data structure. It stores data in a tabular form. Its three main components are **data, rows,** and **columns**.

3. Pandas are used for efficient data analysis. The key features of Pandas are as follows:
   - Fast and efficient data manipulation and analysis
   - Provides time-series functionality
   - Easy missing data handling
   - Faster data merging and joining
   - Flexible reshaping and pivoting of data sets
   - Powerful group by functionality
   - Data from different file objects can be loaded
   - Integrates with NumPy

4. A Series in Pandas is a one-dimensional labelled array. Its columns are like an Excel sheet that can hold any type of data, which can be, an integer, string, or Python objects, etc. Its axis labels are known as the **index**. Series contains homogeneous data and its values can be changed but the size of the series is immutable. A series can be created from a Python tuple, list and dictionary. The syntax for creating a series is as follows:

```
import pandas as pd
series = pd.Series(data)
```

5. In Pandas, a series can be created in many ways. They are as follows:

   Creating an Empty Series
   An empty series can be created by just calling the **pandas.Series()** constructor.

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

  # Creating empty series
  print(pd.Series())
```

Output:

```
Series([], dtype: float64)
```

Creating a Series from an Array

In order to create a series from the NumPy array, we have to import the NumPy module and have to use the **array()** function.

```
# import pandas and numpy
  import pandas as pd
  import numpy as np

  # simple array
  data = np.array(['g', 'e', 'e', 'k', 's'])

  # convert array to Series
  print(pd.Series(data))
```

Output:

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

6. In Pandas, there are two ways to create a copy of the Series. They are as follows:
Shallow Copy is a copy of the series object where the indices and the data of the original object are not copied. It only copies the references to the indices and data. This means any changes made to a series will be reflected in the other. A shallow copy of the series can be created by writing the following syntax:

```
ser.copy(deep=False)
```

Deep Copy is a copy of the series object where it has its own indices and data. This means nay changes made to a copy of the object will not be reflected tot he original series object. A deep copy of the series can be created by writing the following syntax:

```
ser.copy(deep=True)
```

The default value of the deep parameter of the copy() function is set to True.

7. A DataFrame in Panda is a data structure used to store the data in tabular form, that is in the form of rows and columns. It is two-dimensional, size-mutable, and heterogeneous in nature. The main components of a dataframe are data, rows, and columns. A dataframe can be created by loading the dataset from existing storage, such as SQL database, CSV file, Excel file, etc. The syntax for creating a dataframe is as follows:

   ```
   import pandas as pd
   dataframe = pd.DataFrame(data)
   ```

8. In Pandas, a dataframe can be created in many ways. They are as follows:

Creating an Empty DataFrame

An empty dataframe can be created by just calling the **pandas.DataFrame()** constructor.

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

  # Calling DataFrame constructor
  print(pd.DataFrame())
```

Output:

```
Empty DataFrame
Columns: []
Index: []
```

Creating a DataFrame using a List

In order to create a DataFrame from a Python list, just pass the list to the DataFrame() constructor.

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

  # list of strings
  lst = ['Geeks', 'For', 'Geeks', 'is',
  'portal', 'for', 'Geeks']

  # Calling DataFrame constructor on list
  print(pd.DataFrame(lst))
```

Output:

```
      0
0   Geeks
1     For
2   Geeks
3      is
4  portal
5     for
6   Geeks
```

9. We can create a data frame from a CSV file – "Comma Separated Values". This can be done by using the **read_csv()** method which takes the csv file as the parameter.

   ```
   pandas.read_csv(file_name)
   ```

   Another way to do this is by using the **read_table()** method which takes the CSV file and a delimiter value as the parameter.

   ```
   pandas.read_table(file_name, deliniter)
   ```

10. The first few records of a dataframe can be accessed by using the pandas **head()** method. It takes one optional argument **n**, which is the number of rows. By default, it returns the first 5 rows of the dataframe. The head() method has the following syntax:

    ```
    df.head(n)
    ```

    Another way to do it is by using **iloc()** method. It is similar to the Python list-slicing technique. It has the following syntax:

    ```
    df.iloc[:n]
    ```

11. Reindexing in Pandas as the name suggests means changing the index of the rows and columns of a dataframe. It can be done by using the Pandas **reindex()** method. In case of missing values or new values that are not present in the dataframe, the reindex() method assigns it as NaN.

```
df.reindex(new_index)
```

12. There are many ways to Select a single column of a dataframe. They are as follows:
By using the **Dot operator**, we can access any column of a dataframe.

```
Dataframe.column_name
```

Another way to select a column is by using the **square brackets** [].

```
DataFrame[column_name]
```

13. A column of the dataframe can be renamed by using the **rename()** function. We can rename a single as well as multiple columns at the same time using this method.

```
DataFrame.rename(columns={'column1': 'COLUMN_1', 'column2':'COLUMN_2'},
inplace=True)
```

Another way is by using the **set_axis()** function which takes the new column name and axis to be replaced with the new name.

```
DataFrame.set_axis(labels=['COLUMN_1','COLUMN_2'], axis=1, inplace=True)
```

In case we want to add a prefix or suffix to the column names, we can use the **add_prefix()** or **add_suffix()** methods.

```
DataFrame.add_prefix(prefix='PREFIX_')
DataFrame.add_suffix(suffix='_suffix')
```

14. **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 take 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)
```

15. We can delete a row or a column from a dataframe by using 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)
```

16. We can set the index to a Pandas dataframe by using the set_index() method, which is used to set a list, series, or dataframe as the index of a dataframe.

```
DataFrame.set_index('Column_Name')
```

17. The index of Pandas dataframes can be reset by using the reset_index() method. It can be used to simply reset the index to the default integer index beginning at 0.

```
DataFrame.reset_index(inplace = True)
```

18. Pandas dataframe.corr() method is used to find the correlation of all the columns of a dataframe. It automatically ignores any missing or non-numerical values.

```
DataFrame.corr()
```

19. There are various ways to iterate the rows and columns of a dataframe.

Iteration over Rows

In order to iterate over rows, we apply a **iterrows()** function this function returns each index value along with a series containing the data in each row. Another way to iterate over rows is by using **iteritems()** method, which iterates over each column as key-value pairs. We can also use **itertuples()** function which returns a tuple for each row in the DataFrame.The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

Iteration over Columns

To iterate columns of a dataframe, we just need to create a list of dataframe columns by using the list constructor and passing the dataframe to it.

20. 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.

21. Categorical data is a set of predefined data values under some categories. It usually has a limited and fixed range of possible values and can be either numerical or textual in nature. A few examples of categorical data are gender, educational qualifications, blood type, country affiliation, observation time, etc. In Pandas categorical data is often represented by Object datatype.

22. A Pandas dataframe can be converted to an Excel file by using the to_excel() function which takes the file name as the parameter. We can also specify the sheet name in this function.

```
DataFrame.to_excel(file_name)
```

23. Multi-indexing refers to selecting two or more rows or columns in the index. It is a multi-level or hierarchical object for pandas object and deals with data analysis and works with higher dimensional data. Multi-indexing in Pandas can be achieved by using a number of functions, such as **MultiIndex.from_arrays, MultiIndex.from_tuples, MultiIndex.from_product, MultiIndex.from_frame**, etc which helps us to create multiple indexes from arrays, tuples, dataframes, etc.

24. The Pandas select_dtypes() method is used to include or exclude a specific type of data in the dataframe. The datatypes to include or exclude are specified to it as a list or parameters to the function. It has the following syntax:

```
DataFrame.select_dtypes(include=['object','float'], exclude =['int'])
```

25. Pandas Numpy is an inbuilt Python package that is used to perform large numerical computations. It is used for processing multidimensional array elements to perform complicated mathematical operations.
The pandas dataframe can be converted to a NumPy array by using the to_numpy() method. We can also provide the datatype as an optional argument.

```
Dataframe.to_numpy()
```

We can also use .values to convert dataframe values to NumPy array

```
df.values
```

26. Boolean masking is a technique that can be used in Pandas to split a DataFrame depending on a boolean criterion. You may divide different regions of the DataFrame and filter rows depending on a certain criterion using boolean masking.

```
# Define the condition
condition = DataFrame['col_name'] < VALUE
# DataFrame with rows where the condition is True
DataFrame1 = DataFrame[condition]
# DataFrame with rows where the condition is False
DataFrame1 = DataFrame[~condition]
```

27. Time series is a collection of data points with timestamps. It depicts the evolution of quantity over time. Pandas provide various functions to handle time series data efficiently. It is used to work with data timestamps, resampling time series for different time periods, working with missing data, slicing the data using timestamps, etc.

Pandas Built-in Function	**Operation**
`pandas.to_datetime(DataFrame['Date'])`	Convert 'Date' column of DataFrame to datetime dtype
`DataFrame.set_index('Date', inplace=True)`	Set 'Date' as the index
`DataFrame.resample('H').sum()`	Resample time series to a different frequency (e.g., Hourly, daily, weekly, monthly etc)
`DataFrame.interpolate()`	Fill missing values using linear interpolation
`DataFrame.loc[start_date:end_date]`	Slice the data based on

Pandas Built-in Function	Operation
	timestamps

28. 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.
The time delta object can be created by using the **timedelta**() method and providing the number of weeks, days, seconds, milliseconds, etc as the parameter.

```
Duration = pandas.Timedelta(days=7, hours=4, minutes=30, seconds=23)
```

With the help of the Timedelta data type, you can easily perform arithmetic operations, comparisons, and other time-related manipulations. In terms of different units, such as days, hours, minutes, seconds, milliseconds, and microseconds, it can give durations.

```
Duration + pandas.Timedelta('2 days 6 hours')
```

29. 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. The following table shows the difference between merge() and concat():

.merge()	concat()
It is used to join exactly 2 dataframes based on a common column or index	It is used to join 2 or more dataframes along a particular axis i.e rows or columns
Perform different types of joins such as inner join, outer join, left join, and right join.	Performs concatenation by appending the dataframes one below the other (along the rows) or side by side (along the columns).

.merge()	concat()
Join types and column names have to be specified.	By default, performs row-wise concatenation (i.e. axis=0). To perform column-wise concatenation (i.e. axis=1)
Multiple columns can be merged if needed	Does not perform any sort of matching or joining based on column values
Used when we want to combine data based on a shared column or index.	Commonly used when you want to combine dataframes vertically or horizontally without any matching criteria.

31. The map(), applymap(), and apply() methods are used in pandas for applying functions or transformations to elements in a DataFrame or Series. The following table shows the difference between map(), applymap() and apply():

map()	applymap()	apply()
Defined only in Series	Defined only in Dataframe	Defined in both Series and DataFrame
Used to apply a function or a dictionary to each element of the Series.	Used to apply a function to each element of the DataFrame.	Used to apply a function along a specific axis of the DataFrame or Series.
Series.map() works element-wise and can be used to perform element-wise transformations or mappings.	DataFrame.applymap() works element-wise, applying the provided function to each element in the DataFrame.	DataFrame.apply() works on either entire rows or columns element-wise of a Dataframe or Series
Used when we want to apply a simple transformation or mapping operation to each element of a series	Used when we want to apply a function to each individual element of a Dataframe	Used when we want to apply a function that aggregates or transforms data across rows or columns.

32. Both pivot_table() and groupby() are powerful methods in pandas used for aggregating and summarizing data. The following table shows the difference between pivot_table() and groupby():

pivot_table()	**groupby()**
It summarizes and aggregates data in a tabular format	It performs aggregation on grouped data of one or more columns
Used to transform data by reshaping it based on column values.	Used to group data based on categorical variables then we can apply various aggregation functions to the grouped data.
It can handle multiple levels of grouping and aggregation, providing flexibility in summarizing data.	It performs grouping based on column values and creates a GroupBy object then aggregation functions, such as sum, mean, count, etc., can be applied to the grouped data.
It is used when we want to compare the data across multiple dimensions	It is used to summarize data within groups

33. We can pivot the dataframe in Pandas by using the pivot_table() method. To unpivot the dataframe to its original form we can melt the dataframe by using the melt() method.

34. A Python string can be converted to a DateTime object by using the to_datetime() function or **strptime()** method of datetime. It returns a DateTime object corresponding to date_string, parsed according to the format string given by the user.

Using Pandas.to_datetime()

```
import pandas as pd

  # Convert a string to a datetime object
  date_string = '2023-07-17'
  dateTime = pd.to_datetime(date_string)
  print(dateTime)
```

Output:

```
2023-07-17 00:00:00
```

Using datetime.strptime

```
from datetime import datetime

 # Convert a string to a datetime object
   date_string = '2023-07-17'
   dateTime = datetime.strptime(date_string, '%Y-%m-%d')
   print(dateTime)
```

Output:

```
2023-07-17 00:00:00
```

35. Pandas describe() is used to view some basic statistical details of a data frame or a series of numeric values. It can give a different output when it is applied to a series of strings. It can get details like percentile, mean, standard deviation, etc.

```
DataFrame.describe()
```

36. The mean, median, mode, Variance, Standard Deviation, and Quantile range can be computed using the following commands in Python.

- DataFrame.mean(): To calculate the mean
- DataFrame.median(): To calculate median
- DataFrame.mode(): To calculate the mode
- DataFrame.var(): To calculate variance
- DataFrame.std(): To calculate the standard deviation
- DataFrame.quantile(): To calculate quantile range, with range value as a parameter

37. 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.

38. One-hot encoding is a technique for representing categorical data as numerical values in a machine-learning model. It works by creating a separate binary variable for each category in the data. The value of the binary variable is 1 if the observation belongs to that category and 0 otherwise. It can improve the performance of the model. To apply one hot encoding, we greater a dummy column for our dataframe by using get_dummies() method.

39. A Boxplot is a visual representation of grouped data. It is used for detecting outliers in the data set. We can create a boxplot using the Pandas dataframe by using the boxplot() method and providing the parameter based on which we

want the boxplot to be created.

```
DataFrame.boxplot(column='Col_Name', grid=False)
```

40. A distribution plot is a graphical representation of the distribution of data. It is a type of histogram that shows the frequency of each value in a dataset. To create a distribution plot using Pandas, you can use the plot.hist() method. This method takes a DataFrame as input and creates a histogram for each column in the DataFrame.

```
DataFrame['Numerical_Col_Name'].plot.hist()
```

41. A dataframe in pandas can be sorted in ascending or descending order according to a particular column. We can do so by using the **sort_values()** method. and providing the column name according to which we want to sort the dataframe. we can also sort it by multiple columns. To sort it in descending order, we pass an additional parameter 'ascending' and set it to False.

```
DataFrame.sort_values(by='Age',ascending=True)
```

42. In pandas, duplicate values can be checked by using the duplicated() method.

```
DataFrame.duplicated()
```

To remove the duplicated values we can use the drop_duplicates() method.

```
DataFrame.drop_duplicates()
```

43. We can create a column from an existing column in a DataFrame by using the df.apply() and df.map() functions.

44. Generally 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.

45. The groupby() function is used to group or aggregate the data according to a category. It makes the task of splitting the Dataframe over some criteria really easy and efficient. It has the following syntax:

```
DataFrame.groupby(by=['Col_name'])
```

46. Pandas Subset Selection is also known as Pandas Indexing. It means selecting a particular row or column from a dataframe. We can also select a number of rows or columns as well. Pandas support the following types of indexing:

- **Dataframe.[ ]:** This function is also known as the indexing operator
- **Dataframe.loc[ ]:** This function is used for label-based indexing.
- **Dataframe.iloc[ ]:** This function is used for positions or integer-based indexing.

47. In pandas, we can combine two dataframes using the pandas.merge() method which takes 2 dataframes as the parameters.

```
import pandas as pd
  # Create two DataFrames
  df1 = pd.DataFrame({'A': [1, 2, 3],
  'B': [4, 5, 6]},
  index=[10, 20, 30])

  df2 = pd.DataFrame({'C': [7, 8, 9],
  'D': [10, 11, 12]},
  index=[20, 30, 40])

  # Merge both dataframe
  result = pd.merge(df1, df2, left_index=True, right_index=True)
  print(result)
```

Output:

```
    A  B  C   D
20  2  5  7  10
30  3  6  8  11
```

48. The iloc() and loc() functions of pandas are used for accessing data from a DataFrame.The following table shows the difference between iloc() and loc():

iloc()	loc()
It is an indexed-based selection method	It is labelled based selection method
It allows you to access rows and columns of a DataFrame by their integer positions	It allows you to access rows and columns of a DataFrame using their labels or names.
The indexing starts from 0 for both rows and columns.	The indexing can be based on row labels, column labels, or a combination of both.
Used for integer-based slicing, which can be single integers, lists or arrays of integers for specific rows or columns.	Used for label-based slicing, the labels can be single labels, lists or arrays of labels for specific rows or columns
Syntax: `DataFrame.iloc[row_index, column_index]`	**Syntax**: `DataFrame.loc[row_label, column_label]`

49. Both join() and merge() functions in pandas are used to combine data from multiple DataFrames. The following table shows the difference between join and merge():

join()	merge()
Combines dataframes on their indexes	Combines dataframes by specifying the columns as a merge key
Joining is performed on the DataFrame's index and not on any specified columns.	Joining is performed based on the values in the specified columns or indexes.
Does not support merging based on column values or multiple columns.	Supports merging based on one or more columns or indexes, allowing for more flexibility in combining DataFrames.

50. The interpolate() and fillna() methods in pandas are used to handle missing or NaN (Not a Number) values in a DataFrame or Series. The following table shows the difference between interpolate() and fillna():

interpolate()	fillna()
Fill in the missing values based on the interpolation or estimate values based on the	Fill missing values with specified values that can be based on some

interpolate()	fillna()
existing data.	strategies.
Performs interpolation based on various methods such as linear interpolation, polynomial interpolation, and time-based interpolation.	Replaces NaN values with a constant like zero, mean, median, mode, or any other custom value computed from the existing data.
Applied to both numerical and DateTime data when dealing with time series data or when there is a logical relationship between the missing values and the existing data.	Can be applied to both numerical and categorical data.

Section- B

1. Pandas is an open-source Python library with powerful and built-in methods to efficiently clean, analyze, and manipulate datasets. Developed by Wes McKinney in 2008, this powerful package can easily blend with various other data science modules in Python.

   Pandas is built on top of the NumPy library, i.e., its data structures Series and DataFrame are the upgraded versions of NumPy arrays.

2. The `head()` method in pandas is used to access the initial rows of a DataFrame, and tail() method is used to access the last rows.

   To access the top 6 rows: `dataframe_name.head(6)`

   To access the last 7 rows: `dataframe_name.tail(7)`

3. In pandas, `shape` is an attribute and not a method. So, you should access it without parentheses.

   `DataFrame.shape` outputs a tuple with the number of rows and columns in a DataFrame.

4. **DataFrame:** The pandas DataFrame will be in tabular format with multiple rows and columns where each column can be of different data types.

Series: The Series is a one-dimensional labeled array that can store any data type, but all of its values should be of the same data type. The Series data structure is more like single column of a DataFrame.

The Series data structure consumes less memory than a DataFrame. So, certain data manipulation tasks are faster on it.

However, DataFrame can store large and complex datasets, while Series can handle only homogeneous data. So, the set of operations you can perform on DataFrame is significantly higher than on Series data structure.

5. The index is a series of labels that can uniquely identify each row of a DataFrame. The index can be of any datatype like integer, string, hash, etc.,

   `df.index` prints the current row indexes of the DataFrame df.

6. Index in pandas uniquely specifies each row of a DataFrame. We usually pick the column that can uniquely identify each row of a DataFrame and set it as the index. But what if you don't have a single column that can do this?

   For example, you have the columns "name", "age", "address", and "marks" in a DataFrame. Any of the above columns may not have unique values for all the different rows and are unfit as indexes.

   However, the columns "name" and "address" together may uniquely identify each row of the DataFrame. So you can set both columns as the index. Your DataFrame now has a multi-index or hierarchical index.

7. Reindexing in pandas lets us create a new DataFrame object from the existing DataFrame with the updated row indexes and column labels.

   You can provide a set of new indexes to the function DataFrame.reindex() and it will create a new DataFrame object with given indexes and take values from the actual DataFrame.

   If values for these new indexes were not present in the original DataFrame, the function fills those positions with the default nulls. However, we can alter the default value NaN to whatever value we want them to fill with.

   Here is the sample code:

   Create a DataFrame df with indexes:

```
import pandas as pd

data = [['John', 50, 'Austin', 70],
        ['Cataline', 45 , 'San Francisco', 80],
    ['Matt', 30, 'Boston' , 95]]

columns = ['Name', 'Age', 'City', 'Marks']

#row indexes
idx = ['x', 'y', 'z']

df = pd.DataFrame(data, columns=columns, index=idx)

print(df)
```

Reindex with new set of indexes:

```
new_idx = ['a', 'y', 'z']



new_df = df.reindex(new_idx)



print(new_df)
```

The `new_df` has values from the `df` for common indexes (‘y’ and ‘z’), and the new index ‘a’ is filled with the default NaN.

8. Both loc and the iloc methods in pandas are used to select subsets of a DataFrame. Practically, these are widely used for filtering DataFrame based on conditions.

   We should use the loc method to select data using actual labels of rows and columns, while the iloc method is used to extract data based on integer indices of rows and columns.

9. **Using Python Dictionary:**

```
import pandas as pd



data = {'Name': ['John', 'Cataline', 'Matt'],

    'Age': [50, 45, 30],

    'City': ['Austin', 'San Francisco', 'Boston'],

    'Marks' : [70, 80, 95]}
```

```
df = pd.DataFrame(data)
```

Using Python Lists:

```
import pandas as pd


data = [['John', 25, 'Austin',70],
        ['Cataline', 30, 'San Francisco',80],
        ['Matt', 35, 'Boston',90]]


columns = ['Name', 'Age', 'City', 'Marks']


df = pd.DataFrame(data, columns=columns)
```

10. The function `Series.value_counts()` returns the count of each unique value of a series or a column.

Example:

We have created a DataFrame df that contains a categorical column named ‘Sex’, and ran `value_counts()` function to see the count of each unique value in that column.

```
import pandas as pd


data = [['John', 50, 'Male', 'Austin', 70],
        ['Cataline', 45 ,'Female', 'San Francisco', 80],
        ['Matt', 30 ,'Male','Boston', 95]]


# Column labels of the DataFrame

columns = ['Name','Age','Sex', 'City', 'Marks']
```

```
# Create a DataFrame df

df = pd.DataFrame(data, columns=columns)



df['Sex'].value_counts()
```

11. **Load less data:** While reading data using `pd.read_csv()`, choose only the columns you need with the "usecols" parameter to avoid loading unnecessary data. Plus, specifying the "chunksize" parameter splits the data into different chunks and processes them sequentially.

    Avoid loops: Loops and iterations are expensive, especially when working with large datasets. Instead, opt for vectorized operations, as they are applied on an entire column at once, making them faster than row-wise iterations.

    Use data aggregation: Try aggregating data and perform statistical operations because operations on aggregated data are more efficient than on the entire dataset.

    Use the right data types: The default data types in pandas are not memory efficient. For example, integer values take the default datatype of int64, but if your values can fit in int32, adjusting the datatype to int32 can optimize the memory usage.

    Parallel processing: Dask is a pandas-like API to work with large datasets. It utilizes multiple processes of your system to parallely execute different data tasks.

12. **Join:** Joins two DataFrames based on their index. However, there is an optional argument 'on' to explicitly specify if you want to join based on columns. By default, this function performs left join.

    Syntax: `df1.join(df2)`

    Merge: The merge function is more versatile, allowing you to specify the columns on which you want to join the DataFrames. It applies inner join by default, but can be customized to use different join types like left, right, outer, inner, and cross.

    Syntax: `pd.merge(df1, df2, on="column_names")`

13. Timedelta represents the duration ie., the difference between two dates or times, measured in units as days, hours, minutes, and seconds.

14.

(A) We can use the `concat` method to combine DataFrames either along rows or columns. Similarly, append is also used to combine DataFrames, but only along the rows.

(B) With the concat function, you have the flexibility to modify the original DataFrame using the “inplace” parameter, while the append function can't modify the actual DataFrame, instead it creates a new one with the combined data.

15. First, we should use the read_excel() function to pull in the Excel data to a variable. Then, just apply the to_csv() function for a seamless conversion.

Here is the sample code:

```
import pandas as pd


#input your excel file path into the read_excel() function.

excel_data = pd.read_excel("/content/sample_data/california_housing_test.xlsx")


excel_data.to_csv("CSV_data.csv", index = None, header=True)
```

16. We have the sort_values() method to sort the DataFrame based on a single column or multiple columns.

Syntax: `df.sort_values(by=[“column_names”])`

Example code:

```
import pandas as pd


data = [['John', 50, 'Male', 'Austin', 70],

['Cataline', 45 ,'Female', 'San Francisco', 80],

['Matt', 30 ,'Male', 'Boston', 95],

['Oliver',35,'Male', 'New york', 65]]
```

```
# Column labels of the DataFrame

columns = ['Name','Age','Sex', 'City', 'Marks']



# Create a DataFrame df

df = pd.DataFrame(data, columns=columns)



# Sort values based on ‘Age’ column

df.sort_values(by=['Age'])



df.head()`
```

17. To create a DataFrame:

```
import pandas as pd



data = {'Name': ['John', 'Cataline', 'Matt'],

       'Age': [50, 45, 30],

       'City': ['Austin', 'San Francisco', 'Boston'],

       'Marks' : [70, 80, 95]}



# Create a DataFrame df

df = pd.DataFrame(data)
```

Method 1: Based on conditions

```
new_df = df[(df.Name == "John") | (df.Marks > 90)]

print (new_df)
```

Method 2: Using query function

```
df.query('Name == "John" or Marks > 90')

print (new_df)
```

18. The `groupby` function lets you aggregate data based on certain columns and perform operations on the grouped data. In the following code, the data is grouped on the column 'Name' and the mean 'Marks' of each group is calculated.

```
import pandas as pd



# Create a DataFrame

data = {

    'Name': ['John', 'Matt', 'John', 'Matt', 'Matt', 'Matt'],

    'Marks': [10, 20, 30, 15, 25, 18]

}



# Create a DataFrame df

df = pd.DataFrame(data)



# mean marks of John and Matt

print(df.groupby('Name').mean())
```

19. We can use `apply()` method to derive a new column by performing some operations on existing columns.

The following code adds a new column named 'total' to the DataFrame. This new column holds the sum of values from the other two columns.

Example code:

```
import pandas as pd



# Create a DataFrame

data = {

    'Name': ['John', 'Matt', 'John', 'Cateline'],

    'math_Marks': [18, 20, 19, 15],

    'science_Marks': [10, 20, 15, 12]

}



# Create a DataFrame df

df = pd.DataFrame(data)



df['total'] = df.apply(lambda row : row["math_Marks"] + row["science_Marks"], axis=1)





print(df)
```

20. You can use any of the following three methods to handle missing values in pandas:

`dropna()` – the function removes the missing rows or columns from the DataFrame.

`fillna()` – fill nulls with a specific value using this function.

`interpolate()` – this method fills the missing values with computed interpolation values. The interpolation technique can be linear, polynomial, spline, time, etc.,

21. **fillna() –**

    `fillna()` fills the missing values with the given constant. Plus, you can give forward-filling or backward-filling inputs to its 'method' parameter.

    interpolate() –

    By default, this function fills the missing or NaN values with the linear interpolated values. However, you can customize the interpolation technique to polynomial, time, index, spline, etc., using its 'method' parameter.

    The interpolation method is highly suitable for time series data, whereas fillna is a more generic approach.

22. Resampling is used to change the frequency at which time series data is reported. Imagine you have monthly time series data and want to convert it into weekly data or yearly, this is where resampling is used.

    Converting monthly to weekly or daily data is nothing but upsampling. Interpolation techniques are used to increase the frequencies here.

    In contrast, converting monthly to yearly data is termed as downsampling, where data aggregation techniques are applied.

23. We perform one hot encoding to convert categorical values to numeric ones so that can be fed to the machine learning algorithm.

```
import pandas as pd



data = {'Name': ['John', 'Cateline', 'Matt', 'Oliver'],

        'ID': [1, 22, 23, 36]}



df = pd.DataFrame(data)



#one hot encoding
```

```
new_df = pd.get_dummies(df.Name)

new_df.head()
```

24. To draw a line plot, we have a plot function in pandas.

```
import pandas as pd




data = {'units': [1, 2, 3, 4, 5],

        'price': [7, 12, 8, 13, 16]}

# Create a DataFrame df

df = pd.DataFrame(data)



df.plot(x='units', y='price')
```

25. `df.describe()`

    This method returns stats like mean, percentile values, min, max, etc., of each column in the DataFrame.

26. Rolling mean is also referred to as moving average because the idea here is to compute the mean of data points for a specified window and slide the window throughout the data. This will lessen the fluctuations and highlight the long-term trends in time series data.

    Syntax: `df['column_name'].rolling(window=n).mean()`

Section- C

1. Python Pandas is a data analysis and manipulation software library built by **Wes McKinney**. It is an open-source, cross-platform library. It provides data

structures and procedures for numerical and time series data manipulation. It makes machine learning algorithms easy to implement.

2. It is used for data analysis, time series manipulation, and table management. It is specially designed for the Python programming language.

3. It is a one-dimensional array of objects of any data type. Using the 'series' method, you can convert any list, tuple, and dictionary into a series. A series cannot have a column. The row labels of the series are called indexes.

4. Pandas provides two types of data structures built on top of NumPy. These are

   - series and DataFrames.

   - Series are one-dimensional, whereas DataFrames are two-dimensional data types.

5. The features of Pandas library are:

- Time Series
- Data Alignment
- Merge and Join
- Reshaping
- Memory efficient

6. In pandas, you can calculate the standard deviation of a Series using the .std() method. For example:

```
import pandas as pd
data = pd.Series([1, 2, 3, 4, 5])
std_deviation = data.std()
```

   Here, std_deviation will contain the standard deviation of the data in the Series.

7. A DataFrame is an extensively used data structure in Pandas and works with 2-D arrays with labelled axes. It is a standard storing data with row and column indices. The columns can store heterogeneous data such as int and bool. It can be viewed as a dictionary of series data structures.

8. Time series is an organised collection of data points showing a quantity's evolution over time. Pandas are extremely capable and have the tools to work with time series data from various fields.
   Functions provided by Pandas:

- Create date and time sequences using preset frequencies
- Date and time manipulation supported by timezone feature
- Conversion of time series to a given frequency or to resample
- Analysing time series data from several sources
- Calculating date and time in absolute or relative terms

9. Reindexing allows the assignment of new indices and has configurable filling logic. It injects NA/NaN in the areas where the elements are missing from the last index. It returns an object unless the new index is equivalent to the current one, and the value of the copy becomes false. It is used to alter the index of the rows and columns of the DataFrame.

10. MultiIndexing in Pandas allows us to have multi-levels of row and column labels which provide a way to analyze and represent data. With the help of MultiIndexing, one can organize the data in a tabular format with multiple features.

11. TimeDelta is a data type in Python. It represents the duration or difference between two points in time. TimeDelta is mainly used to perform arithmetic operations involving dates and times. It can be positive or negative and can store values for days, seconds, minutes, hours, and weeks.

12. The Series() method is used without the index parameter to create a series.

13. **Scatter_matrix** is used for this purpose.

14. A pandas Series or DataFrame can be shifted or offset by using a time offset, which is a relative period of time. The representation of time spans like days, weeks, months, and years can be done using time offsets.

    The pandas.offsets module can be used to produce time offsets. A range of pre-defined time offsets, including Day(), Week(), Month(), and Year(), are available in the pandas.offsets module. By mixing the pre-defined time offsets, you can also produce custom time offsets.

15. A Categorical data is a Pandas data type corresponding to a categorical variable in statistics. A categorical variable usually takes a limited and fixed number of values. All values of categorical data are in categories, and np. Nan.

16. To create a copy of the series, use the following code snippet:
pandas.Series.copy

Series.copy(deep=True)
The above code creates a deep copy that includes a copy of data and indices. It will not copy data or indices if deep is set to false.

17. While creating a DataFrame, you can add inputs to the **index** argument. It will ensure that you have the required index. If you don't specify inputs, the DataFrame, by default, contains a numerical index that starts with zero and ends on the last row of the DataFrame.

18. First, reset the index of DataFrame and then execute the following command to remove the index name.

del df.index.name

Remove duplicate index values and drop the identical values from the index column.

19. Use the drop() method to eliminate indices, rows, or columns from a Pandas DataFrame. The DataFrame's related indices, rows, or columns are eliminated by the drop() method, which accepts a list of labels as an input.

20. We can use the **.rename** method to change the index and columns name.

21. You can add new columns to the existing DataFrame. Follow the code snippet below to add a column :

```
# importing the pandas library as pd
import pandas as pd
info = {'one' : pd.Series([21, 12, 33, 14, 51], index=['a', 'b', 'c', 'd', 'e']),
        'two' : pd.Series([18, 32, 39, 48, 45, 56], index=['a', 'b', 'c', 'd', 'e', 'f'])}
info = pd.DataFrame(info)
print ("Passing a series to add new column")
info['three']=pd.Series([89,65,67],index=['a','b','c'])
print (info)
print ("Add new column using previous columns")
info['four']=info['one']+info['three']
print (info)
```

22. You can use **.loc, iloc and ix** to add new rows to a DataFrame.
loc work for labels of the index, iloc works for the position and ix requires a label to be passed to it if it is integer based.

23. To iterate over the rows of the DataFrame, use loop with the **iterrows()** method.

24. NumPy extends to Numerical Python. Calculations in NumPy arrays are faster than in regular Python arrays. It is a Python package to perform various analyses and process single-dimensional and multidimensional array elements.

25. DataFrames can be converted to NumPy arrays to perform high-level mathematical computations. You can use **DataFrame.to_numpy()** method for conversion. This function will return a NumPy array.

26. Below are some statistical functions in Python Pandas.

- **mean():** This function computes the arithmetic mean along a specified axis.
- **median():** This function calculates the median along a specified axis.
- **mode():** This function calculates the mode() along a specified axis.
- **sum():** This function finds the sum of values along a specified axis.
- **var():** It calculates the variance along a specified axis.

27. You can use the steps below to find the series A items that are missing from series B:

- Make a collection of the series B items. The set() function can be used to accomplish this.
- Add the series B set of items to the series A group of things. To accomplish this, use the - operator.
- The items in series A that are missing from series B will make up the resulting set.

28. We can use the to_excel() function to convert a DataFrame to an excel file using Pandas in Python. To use this function, you can simply provide the DataFrame and the desired file name as an argument.

29. The main work of data aggregation is to apply some assembly to one or more columns. It uses **sum** to return the sum of the values, **min** to return the minimum, and **max** to return the maximum value for the requested axis.

30. A method for indexing a Pandas DataFrame with numerous layers is multiple indexing, commonly referred to as hierarchical indexing. As a result, you can design indexes with numerous dimensions, including those for data with time series, locations, or categories.

31. The .concat() method stacks multiple DataFrames vertically or connects them horizontally after aligning them on an index.

32. The **GroupBy()** functions' main task is to split the data into various groups. It allows rearranging the data by utilising them in real-world data sets.

33. To sort the DataFrame use the **DataFrame.sort_values()** function. It sorts the DataFrame row or column-wise. The important parameters of the sort function are:

- **axis**: specifies whether to sort for rows (0) or columns (1)
- **by**: specifies which column or rows determine sorting
- **ascending**: specifies whether to sort the DataFrame in ascending or descending order

34. You can use the pd.DataFrame() function in Pandas without giving any arguments to build a blank DataFrame. A DataFrame without any rows or columns will result from this.

Here is an example of how to make a blank Python DataFrame:

```
import pandas as pd
# Create an empty DataFrame
df = pd.DataFrame()
# Print the DataFrame
print(df)
```

35. To split the DataFrame, first create a mask to separate the data frame and then use the (~) inverse operator to take the complement of the mask.

36. The percentiles describe the data distribution we are working on. The median is represented by **50,** whereas the lower and upper borders are at **25** and **75,** respectively. Using this, we can get a clearer idea of how **skewed** is our data.

37. To merge, use the **.merge()** method which is similar to database-style joins. We have the **inner, outer, left and right** merge operations. An inner merge merges left and right data frames keeping only the common values. Left and right merge operations keep all the rows from their side and add empty / Nan values on the missing opposite side. An **outer** merge returns all the rows from the left and right sides.

38. You will have to use the **to_sql** module, create an **SQLAlchemy** engine, and then write DataFrame to the SQL table.

39. You will have to use either **cut()** or **qcut()** functions:

- **cut()** bins the data on values. We use it when we need evenly spaced values in bins. This function will use values rather than frequencies to sort the data.
- **qcut()** bins the data based on sample quantities. We use it to study data by quantities. It will divide an equal number of data in each bin.

40. The major difference between iloc() and loc() is that the iloc() function is used for selecting data based on integer-based indexing. While loc() is used to select data based on label-based indexing.

41. The major difference between join() and merge() in Pandas is below.

join(): It is a method for combining DataFrames based on their indexes. Left join is the default join and it is a convenient way to merge DataFrames.

merge(): This allows merging DataFrames based on specified column values. It supports inner, outer left, and right joins. It can merge DataFrames on one or more columns based on common values to combine the data.

42. The major difference between merge() and concat() in Pandas is below.

merge(): It combines DataFrames based on common columns and performs various joins such as inner, outer, right, and left.

concat(): This function concatenates DataFrames along with a particular axis. It provide no relationship between the data in the DataFrames.

43. The major difference between interpolate() and fillna() in Pandas is below.

interpolate(): It is the method that is used to fill missing values in DataFrame by estimating values based on existing data.

fillna(): Maily fillna() is used to replace missing data or values with the appropriate values.

44. The conversion methods are:

- **to_numeric()** - converts non numeric to numeric type
- **astype()** - converts any type to any other type, it can also convert to categorical types
- **convert_dtypes()** - converts DataFrames to best dtype
- **infer_objects()** - a utility method to convert object columns holding Python objects to a pandas type if possible

Section- D

1. **Pandas** is a powerful Python library for data analysis. In a nutshell, it's designed to make the manipulation and analysis of structured data intuitive and efficient.

Key Features

- **Data Structures:** Offers two primary data structures: `Series` for one-dimensional data and `DataFrame` for two-dimensional tabular data.
- **Data Munging Tools:** Provides rich toolsets for data cleaning, transformation, and merging.
- **Time Series Support:** Extensive functionality for working with time-series data, including date range generation and frequency conversion.
- **Data Input/Output:** Facilitates effortless interaction with a variety of data sources, such as CSV, Excel, SQL databases, and REST APIs.

- **Flexible Indexing:** Dynamically alters data alignments and joins based on row/column index labeling.

Ecosystem Integration

Pandas works collaboratively with several other Python libraries like:

- **Visualization Libraries**: Seamlessly integrates with Matplotlib and Seaborn for data visualization.
- **Statistical Libraries**: Works in tandem with statsmodels and SciPy for advanced data analysis and statistics.

Performance and Scalability

Pandas is optimized for fast execution, making it reliable for small to medium-sized datasets. For large datasets, it provides tools to optimize or work with the data in chunks.

Common Data Operations

- **Loading Data**: Read data from files like CSV, Excel, or databases using the built-in functions.
- **Data Exploration**: Get a quick overview of the data using methods like `describe`, `head`, and `tail`.
- **Filtering and Sorting**: Use logical indexing to filter data or the `sort_values` method to order the data.
- **Missing Data**: Offers methods like `isnull`, `fillna`, and `dropna` to handle missing data efficiently.
- **Grouping and Aggregating**: Group data by specific variables and apply aggregations like sum, mean, or count.
- **Merging and Joining**: Provide several merge or join methods to combine datasets, similar to SQL.
- **Pivoting**: Reshape data, often for easier visualization or reporting.
- **Time Series Operations**: Includes functionality for date manipulations, resampling, and time-based queries.
- **Data Export**: Save processed data back to files or databases.

Code Example

Here is the Python code:

```
import pandas as pd

# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Age': [25, 30, 35, 40],
    'Department': ['HR', 'Finance', 'IT', 'Marketing']
}
df = pd.DataFrame(data)

# Explore the data
print(df)
print(df.describe())  # Numerical summary

# Filter and sort the data
filtered_df = df[df['Department'].isin(['HR', 'IT'])]
sorted_df = df.sort_values(by='Age', ascending=False)

# Handle missing data
df.at[2, 'Age'] = None  # Simulate missing age for 'Charlie'
df.dropna(inplace=True)  # Drop rows with any missing data

# Group, aggregate, and visualize
grouped_df = df.groupby('Department')['Age'].mean()
grouped_df.plot(kind='bar')

# Export the processed data
df.to_csv('processed_data.csv', index=False)
```

2. **Pandas**, a popular data manipulation and analysis library, primarily operates on two data structures: **Series**, for one-dimensional data, and **DataFrame**, for two-dimensional data.

Series Structure

- **Data Model**: Each Series consists of a one-dimensional array and an associated array of labels, known as the index.
- **Memory Representation**: Data is stored in a single ndarray.
- **Indexing**: Series offers simple, labeled indexing.
- **Homogeneity**: Data is homogeneous, meaning it's of a consistent data type.

DataFrame Structure

- **Data Model**: A DataFrame is composed of data, which is in a 2D tabular structure, and an index, which can be a row or column header or both. Extending its table-like structure, a DataFrame can also contain an index for both its rows and columns.
- **Memory Representation**: Internally, a DataFrame is made up of one or more Series structures, in one or two dimensions. Data is 2D structured.
- **Indexing**: Data can be accessed via row index or column name, favoring loc and iloc for multi-axes indexing.
- **Columnar Data**: Columns can be of different, heterogeneous data types.
- **Missing or NaN Values**: DataFrames can accommodate missing or NaN entries.

Common Features of Series and DataFrame

Both Series and DataFrame share some common characteristics:

- **Mutability**: They are both mutable in content, but not in length of the structure.
- **Size and Shape Changes**: Both the Series and DataFrame can change in size. For the Series, you can add, delete, or modify elements. For the DataFrame, you can add or remove columns or rows.
- **Sliceability**: Both structures support slicing operations and allow slicing with different styles of indexers.

Practical Distinctions

- **Initial Construction**: Series can be built from a scalar, list, tuple, or dictionary. DataFrame, on the other hand, is commonly constructed using a dictionary of lists or another DataFrame.
- **External Data Source Interaction**: DataFrames can more naturally interact with external data sources, like CSV or Excel files, due to their tabular nature.

3. **Pandas** makes reading from and writing to **CSV files** straightforward.

Reading a CSV File

You can read a CSV file into a DataFrame using the `.read_csv()` method. Here is the code:

```
import pandas as pd
```

```
df = pd.read_csv('filename.csv')
```

Configuring the Read Operation

- **Header**: By default, the first row of the file is used as column names. If your file doesn't have a header, you should set `header=None`.
- **Index Column**: Select a column to be used as the row index. Pass the column name or position using the `index_col` parameter.
- **Data Types**: Let Pandas infer data types or specify them explicitly through `dtype` parameter.
- **Text Parsing**: Handle non-standard delimiters or separators using `sep` and `delimiter`.
- **Date Parsing**: Specify date formats for more efficient parsing using the `parse_dates` parameter.

Writing to a CSV File

You can export your DataFrame to a CSV file using the `.to_csv()` method.

```
df.to_csv('output.csv', index=False)
```

- **Index**: If you don't want to save the index, set `index` to `False`.
- **Specifying Delimiters**: If you need to use a different delimiter, e.g., tabs, use `sep` parameter.
- **Handling Missing Values**: Choose a representation for missing values, such as `na_rep='NA'`.
- **Encoding**: Use the `encoding` parameter to specify the file encoding, such as 'utf-8' or 'latin1'.
- **Date Format**: When writing dates to the file, choose 'ISO8601', 'epoch', or specify your custom format with `date_format`.
- **Compression**: If your data is large, use the `compression` parameter to save disk space, e.g., `compression='gzip'` for compressed files.

Example: Writing a Dataframe to CSV

Here is a code example:

```
import pandas as pd

data = {
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 22, 28]
}

df = pd.DataFrame(data)

df.to_csv('people.csv', index=False, header=True)
```

4. In **Pandas**, an index serves as a unique identifier for each row in a DataFrame, making it powerful for data retrieval, alignment, and manipulation.

Types of Indexes

1. **Default, Implicit Index**: Automatically generated for each row (e.g., 0, 1, 2, ...).
2. **Explicit Index**: A user-defined index where labels don't need to be unique.
3. **Unique Index**: All labels are unique, typically seen in primary key columns of databases.
4. **Non-Unique Index**: Labels don't have to be unique, can have duplicate values.
5. **Hierarchical (MultiLevel) Index**: Uses multiple columns to form a unique identifier for each row.

Key Functions for Indexing

- **Loc**: Uses labels to retrieve rows and columns.
- **Iloc**: Uses integer indices for row and column selection.

Operations with Indexes

- **Reindexing**: Changing the index while maintaining data integrity.
- **Set operations**: Union, intersection, difference, etc.
- **Alignment**: Matches rows based on index labels.

Code Example: Index Types

```
# Creating DataFrames with different types of indexes
import pandas as pd

# Implicit index
df_implicit = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2,
3]})

# Explicit index
df_explicit = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2,
3]}, index=['X', 'Y', 'Z'])
```

```
# Unique index
df_unique = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2, 3]},
index=['1st', '2nd', '3rd'])

# Non-unique index
df_non_unique = pd.DataFrame({'A': ['a', 'b', 'c', 'd'], 'B':
[1, 2, 3, 4]}, index=['E', 'E', 'F', 'G'])

# Hierarchical index
df_hierarchical = df_explicit.set_index('A')

print(df_implicit, df_explicit, df_unique, df_non_unique,
df_hierarchical, sep='\n\n')
```

Code Example: Key Operations

```
# Setting up a DataFrame for operation examples
import pandas as pd

data = {
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': [100, 200, 300, 400, 500]
}

df = pd.DataFrame(data, index=['a', 'b', 'c', 'd', 'e'])

# Reindexing
new_index = ['a', 'e', 'c', 'b', 'd']  # Reordering index labels
df_reindexed = df.reindex(new_index)

# Set operations
index1 = pd.Index(['a', 'b', 'c', 'f'])
index2 = pd.Index(['b', 'd', 'c'])
print(index1.intersection(index2))  # Intersection
print(index1.difference(index2))  # Difference

# Data alignment
data2 = {'A': [6, 7], 'B': [60, 70], 'C': [600, 700]}
df2 = pd.DataFrame(data2, index=['a', 'c'])
print(df + df2)  # Aligned based on index labels
```

5. Dealing with **missing data** is a common challenge in data analysis. **Pandas** offers flexible tools for handling missing data in DataFrames.

Common Techniques for Handling Missing Data in Pandas

Dropping Missing Values

This is the most straightforward option, but it might lead to data loss.

- **Drop NaN Rows**: `df.dropna()`
- **Drop NaN Columns**: `df.dropna(axis=1)`

Filling Missing Values

Instead of dropping missing data, you can choose to fill it with a certain value.

- **Fill with a Constant Value**: `df.fillna(value)`
- **Forward Filling**: `df.fillna(method='ffill')`
- **Backward Filling**: `df.fillna(method='bfill')`

Interpolation

Pandas gives you the option to use various interpolation methods like linear, time-based, or polynomial.

- **Linear Interpolation**: `df.interpolate(method='linear')`

Masking

You can create a mask to locate and replace missing data.

- **Create a Mask**: `mask = df.isna()`
- **Replace with a value if NA**: `df.mask(mask, value=value_to_replace_with)`

Tagging

Categorize or "tag" missing data to handle it separately.

- **Select Missing Data**: `missing_data = df[df['column'].isna()]`

Advanced Techniques

Pandas supports more refined strategies:

- **Apply a Function**: `df.apply()`
- **Use External Libraries for Advanced Imputation**: Libraries like `scikit-learn` offer sophisticated imputation techniques.

6. **GroupBy** in Pandas lets you **aggregate** and **analyze** data based on specific features or categories. This allows for powerful split-apply-combine operations, especially when combined with **`agg`** to define multiple aggregations at once.

Core Functions in GroupBy

- **Split-Apply-Combine**: This technique segments data into groups, applies a designated function to each group, and then merges the results.
- **Lazy Evaluation**: GroupBy operations are not instantly computed. Instead, they are tracked and implemented only when required. This guarantees efficient memory utilization.
- **In-Memory Caching**: Once an operation is computed for a distinctive group, its outcome is saved in memory. When a recurring computation for the same group is demanded, the cached result is utilized.

  Beneath this sleek surface, GroupBy functionalities meld simplicity with robustness, furnishing swift and accurate outcomes.

Key Methods

- **`groupby()`**: Divides a dataframe into groups based on specified keys, often column names. This provides a `groupby` object which can be combined with additional aggregations or functions.
- **Aggregation Functions**: These functions generate aggregated summaries once the data has been divided into groups. Commonly used aggregation functions include `sum`, `mean`, `median`, `count`, and `std`.
- **Chaining GroupBy Methods**: `.filter()`, `.apply()`, `.transform()`.

Practical Applications

- **Statistical Summary by Category**: Quickly compute metrics such as average or median for segregated data.
- **Data Quality**: Pinpoint categories with certain characteristics, like groups with more missing values.
- **Splitting by Predicate**: Employ `.filter` to focus on particular categories that match user-specified criteria.
- **Normalized Data**: Deploy `.transform()` to standardize or normalize data within group partitions.

Code Example: GroupBy & Aggregation

Consider this dataset of car sales:

Car	Category	Price	Units Sold
Honda	Sedan	25000	120
Honda	SUV	30000	100
Toyota	Sedan	23000	150
Toyota	SUV	28000	90
Ford	Sedan	24000	110
Ford	Pickup	35000	80

We can compute the following using **GroupBy** and **Aggregation**:

1. **Category-Wise Sales**:
   - Sum of "Units Sold"
   - Average "Price"
2. General computations:
   - Total car sales
   - Maximum car price.

Visualizing Missing Data

The `missingno` library provides a quick way to visualize missing data in pandas dataframes using heatmaps. This can help to identify patterns in missing data that might not be immediately obvious from summary statistics. Here is a code snippet to do that:

7. **Data alignment** and **broadcasting** are two mechanisms that enable pandas to manipulate datasets with differing shapes efficiently.

Data Alignment

Pandas operations, such as addition, are designed to generate new series that adhere to the index of both source series, avoiding any NaN values. This process, where the data aligns based on the index labels, is known as **data alignment**.

This behavior is particularly useful when handling data where values may be missing.

How It Works

Take two DataFrames, `df1` and `df2`, each with different shapes and sharing only partial indices:

- `df1`:
  - Index: A, B, C
  - Column: X
  - Values: 1, 2, 3
- `df2`:
  - Index: B, C, D
  - Column: Y
  - Values: 4, 5, 6

When you perform an addition (`df1 + df2`), pandas join values that have the same index. The resulting DataFrame has:

- Index: A, B, C, D
- Columns: X, Y
- Values: NaN, 6, 8, NaN

Broadcasting

Pandas efficiently manages operations between objects of different dimensions or shapes through **broadcasting**.

It employs a set of rules that allow operations to be performed on datasets even if they don't perfectly align in shape or size.

Key broadcasting rules:

1. **Scalar**: Any scalar value can operate on an entire Series or DataFrame.
2. **Vector-Vector**: Operations occur pairwise—each element in one dataset aligns with the element in the same position in the other dataset.
3. **Vector-Scalar**: Each element in a vector gets the operation with the scalar.

Example: Add a Scalar to a Series

Consider a Series `s`:

```
        A
        -
Index: 0, 1, 2
Value: 3, 4, 5
```

Now, perform `s + 2`. This adds 2 to each element of the Series, resulting in:

```
        A
        -
Index: 0, 1, 2
Value: 5, 6, 7
```

8. **Data slicing** and **filtering** are distinct techniques used for extracting subsets of data in **Pandas**.

Key Distinctions

- **Slicing** entails the selection of contiguous rows and columns based on their order or the position within the DataFrame. This is more about locational reference.
- **Filtering**, however, involves selecting rows and columns conditionally based on specific criteria or labels.

Code Example: Slicing vs Filtering

Here is the Python code:

```
import pandas as pd

# Create a simple DataFrame
data = {'A': [1, 2, 3, 4, 5],
        'B': [10, 20, 30, 40, 50],
        'C': ['foo', 'bar', 'baz', 'qux', 'quux']}
df = pd.DataFrame(data)

# Slicing
sliced_df = df.iloc[1:4, 0:2]
print("Sliced DataFrame:")
print(sliced_df)
# 'iloc' is a positional selection method.

# Filtering
filtered_df = df[df['A'] > 2]
print("\nFiltered DataFrame:")
print(filtered_df)
# Here, we use a conditional expression within the brackets to
filter rows.
```

9. **Pandas** provides versatile methods for combining and linking datasets, with the two main approaches being `join` and `merge`. Let's explore how they operate.

Join: Relating DataFrames on Index or Column

The `join` method is a convenient way to link DataFrames based on specific columns or their indices, aligning them either through intersection or union.

Types of Joins

- **Inner Join**: Retains only the common entries between the DataFrames.
- **Outer Join**: Maintains all entries, merging based on where keys exist in either DataFrame.

These types of joins can be performed both along the index (`df1.join(df2)`) as well as specified columns (`df1.join(df2, on='column_key')`). The default join type is an Inner Join.

Example: Inner Join on Index

```
# Inner join on default indices
result_inner_index = df1.join(df2, lsuffix='_left')

# Inner join on specified column and index
result_inner_col_index = df1.join(df2, on='key', how='inner',
lsuffix='_left', rsuffix='_right')
```

Merge: Handling More Complex Join Scenarios

The `merge` function in pandas provides greater flexibility than `join`, accommodating a range of keys to combine DataFrames.

Join Types

- **Left Merge**: All entries from the left DataFrame are kept, with matching entries from the right DataFrame. Unmatched entries from the right get `NaN` values.
- **Right Merge**: Correspondingly serves the right DataFrame.
- **Outer Merge**: Unites all entries from both DataFrames, addressing any mismatches with `NaN` values.
- **Inner Merge**: Selects only entries with matching keys in both DataFrames.

These merge types can be assigned based on the data requirement. You can use the `how` parameter to specify the type of merge.

Code Example

```
# Perform a left merge aligning on 'key' column
left_merge = df1.merge(df2, on='key', how='left')
```

```
# Perform an outer merge on 'key' and 'another_key' columns
outer_merge = df1.merge(df2, left_on='key',
right_on='another_key', how='outer')
```

10. You can apply a function to all elements in a DataFrame column using **Pandas' `.apply()`** method along with a **lambda function** for quick transformations.

Using .apply() and Lambda Functions

The `.apply()` method works as a vectorized alternative for element-wise operations on a column. This mechanism is especially useful for complex transformation and calculation steps.

Here is the generic structure:

```
# Assuming 'df' is your DataFrame and 'col_name' is the name of
your column
df['col_name'] = df['col_name'].apply(lambda x:
your_function(x))
```

You can tailor this approach to your particular transformation function.

Example: Doubling Values in a Column

Let's say you want to double all values in a `scores` column of your DataFrame:

```
import pandas as pd

# Sample DataFrame
data = {'names': ['Alice', 'Bob', 'Charlie'], 'scores': [80, 90,
85]}
df = pd.DataFrame(data)

# Doubling the scores using .apply() and a lambda function
df['scores'] = df['scores'].apply(lambda x: x*2)

# Verify the changes
print(df)
```

Considerations

- **Efficiency**: Depending on the nature of your function, a traditional `for` loop might be faster, especially for simple operations. However, in many scenarios, using vectorized operations such as `.apply()` can lead to improved efficiency.
- **In-Place vs. Non In-Place**: Specifying `inplace=True` in the `.apply()` method directly modifies the DataFrame, to save you from the need for reassignment.

11. Dealing with **duplicate rows** in a DataFrame is a common data cleaning task. The Pandas library provides simple, yet powerful, methods to identify and handle this issue.

Identifying Duplicate Rows

You can use the `duplicated()` method to **identify rows that are duplicated**.

```
import pandas as pd

# Create a sample DataFrame
data = {'A': [1, 1, 2, 2, 3, 3], 'B': ['a', 'a', 'b', 'b', 'c',
'c']}
df = pd.DataFrame(data)

# Identify duplicate rows
print(df.duplicated())  # Output: [False, True, False, True,
False, True]
```

Dropping Duplicate Rows

You can use the `drop_duplicates()` method to **remove duplicated rows**.

```
# Drop duplicates
unique_df = df.drop_duplicates()

# Alternatively, you can keep the last occurrence
last_occurrence_df = df.drop_duplicates(keep='last')

# To drop in place, use the `inplace` parameter
df.drop_duplicates(inplace=True)
```

Carrying Out Aggregations

For numerical data, you can **aggregate** using functions such as mean or sum. This is beneficial when duplicates may have varying values in other columns.

```
# Aggregate using mean
mean_df = df.groupby('A').mean()
```

Counting Duplicates

To **count the occurrences of duplicates**, use the `duplicated()` method in conjunction with `sum()`. This provides the number of duplicates for each row.

```
# Count duplicates
num_duplicates = df.duplicated().sum()
```

Keeping the First or Last Occurrence

By default, `drop_duplicates()` keeps the **first occurrence** of a duplicated row.

If you prefer to keep the **last occurrence**, you can use the `keep` parameter.

```
# Keep the last occurrence
df_last = df.drop_duplicates(keep='last')
```

Leverage Unique Identifiers

Identifying duplicates might require considering a subset of columns. For instance, in an orders dataset, two orders with the same order date might still be distinct because they involve different products. **Set** the subset of columns to consider with the `subset` parameter.

```
# Consider only the 'A' column to identify duplicates
df_unique_A = df.drop_duplicates(subset=['A'])
```

12. **Converting categorical data to a numeric format**, a process also known as **data pre-processing**, is fundamental for many machine learning algorithms that can only handle numerical inputs.

There are **two common approaches** to achieve this: **Label Encoding** and **One-Hot Encoding**.

Label Encoding

Label Encoding replaces each category with a unique numerical label. This method is often used with **ordinal data**, where the categories have an inherent order.

For instance, the categories "Low", "Medium", and "High" can be encoded as 1, 2, and 3.

Here's the Python code using scikit-learn's `LabelEncoder`:

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd

data = {'Size': ['S', 'M', 'L', 'XL', 'M']}
df = pd.DataFrame(data)

le = LabelEncoder()
df['Size_LabelEncoded'] = le.fit_transform(df['Size'])
print(df)
```

One-Hot Encoding

One-Hot Encoding creates a new binary column for each category in the dataset. For every row, only one of these columns will have a value of 1, indicating the presence of that category.

This method is ideal for **nominal data** that doesn't indicate any order.

Here's the Python code using scikit-learn's `OneHotEncoder` and pandas:

```
from sklearn.preprocessing import OneHotEncoder

# Avoids SettingWithCopyWarning
df = pd.get_dummies(df, prefix=['Size'], columns=['Size'])

print(df)
```

13. In **Pandas**, you can pivot data using the `pivot` or `pivot_table` methods. These functions restructure data in various ways, such as converting unique values into column headers or aggregating values where duplicates exist for specific combinations of row and column indices.

Key Methods

- **`pivot`**: It works well with a simple DataFrame but cannot handle duplicate entries for the same set of index and column labels.
- **`pivot_table`**: Provides more flexibility and robustness. It can deal with data duplicates and perform varying levels of aggregation.

Code Example: Pivoting with `pivot`

Here is the Python code:

```
import pandas as pd

# Sample data
data = {
    'Date': ['2020-01-01', '2020-01-01', '2020-01-02', '2020-01-
02'],
    'Category': ['A', 'B', 'A', 'B'],
    'Value': [10, 20, 30, 40]
}

df = pd.DataFrame(data)

# Pivot the DataFrame
pivot_df = df.pivot(index='Date', columns='Category',
values='Value')
print(pivot_df)
```

Output

The pivot DataFrame is a transformation of the original one:

Date	A	B
2020-01-01	10	20
2020-01-02	30	40

Code Example: Pivoting with `pivot_table`

Here is the Python code:

```
import pandas as pd

# Sample data
data = {
    'Date': ['2020-01-01', '2020-01-01', '2020-01-02', '2020-01-
02'],
    'Category': ['A', 'B', 'A', 'B'],
    'Value': [10, 20, 30, 40]
}
```

```
df = pd.DataFrame(data)

# Pivoting the DataFrame using pivot_table
pivot_table_df = df.pivot_table(index='Date',
columns='Category', values='Value', aggfunc='sum')
print(pivot_table_df)
```

Output

The pivot table presents the sum of values for unique combinations of 'Date' and 'Category'. **Notice how duplicate entries for some combinations have been automatically aggregated**.

Date	A	B
2020-01-01	10	20
2020-01-02	30	40

14. The `where()` method in **Pandas** enables conditional logic on columns, providing a more streamlined alternative to `loc` or `if-else` statements.

The method essentially replaces values in a **DataFrame** or **Series** based on defined conditions.

`where()` Basics

Here are some important key points:

- **Import**: It works directly on data obtained via Pandas modules (`import pandas as pd`).
- **Parameters**: Specify conditions like `cond` for when to apply replacements and `other` for the values to replace when the condition is False.

Code Example: `where()`

Here is the Python code:

```
# Importing Pandas
import pandas as pd

# Sample Data
data = {'A': [1, 2, 3, 4, 5], 'B': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)

# Applying 'where'
result = df.where(df > 2, df * 10)

# Output
print(result)

# Output:   A     B
#           0   NaN   NaN
#           1   NaN   NaN
#           2   3.0  30.0
#           3   4.0  40.0
#           4   5.0  50.0
```

In this example:

- Values less than or equal to 2 are replaced with `original_value * 10`.
- `NaN` values are returned for all the cells that did not meet the condition.

15. The **`apply()`** function in **Pandas** is utilized to apply a given function along either the rows or columns of a DataFrame for advanced data manipulation tasks.

Practical Applications

- **Row or Column Wise Operations**: Suitable for applying element-wise or aggregative functions, like sum, mean, or custom-defined operations, across rows or columns.
- **Aggregations**: Ideal for multiple computations, for example calculating totals and averages simultaneously.
- **Custom Operations**: Provides versatility for applying custom functions across data; for example, calculating the interquantile range of two

columns.

- **Scalability**: Offers performance improvements over element-wise iterations, particularly in scenarios with significantly large datasets.

Syntax

The `apply()` function typically requires specification regarding row or column iteration:

```
# Row-wise function application
df.apply(func, axis=1)

# Column-wise function application (default; '0' or
'index' yields the same behavior)
df.apply(func, axis='columns')
```

Here `func` denotes the function to apply, which can be a built-in function, lamba function, or user-constructed function.

Code Example: `apply()`

Let's look at a simple code example to show how `apply()` can be used to calculate the difference between two columns.

```
import pandas as pd

# Sample data
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define the function to calculate the difference
def diff_calc(row):
    return row['B'] - row['A']

# Apply the custom function along the columns
df['Diff'] = df.apply(diff_calc, axis='columns')

print(df)
```

The output will be:

```
   A  B  Diff
0  1  4     3
```

```
                    Date                                   A     B
1   2   5       3
2   3   6       3
```

Section – E

1. Pandas is a widely-used Python library designed for data manipulation and analysis. It offers two primary data structures:

- **DataFrame**, which is akin to a table with rows and columns, similar to an Excel spreadsheet or a SQL table
- **Series**, which is like a single column from that table

These structures allow users to analyze large datasets with ease.

2. To start using Pandas in their Python script, developers need to import it with an import statement: `import pandas as pd`.

The pd is an alias, a common shorthand that makes it easier to call Pandas functions without typing the full library name each time. This step is key for using any of Pandas' features for data manipulation and analysis.

3. Creating a DataFrame from a dictionary in Pandas is straightforward. Candidates should explain they'd use the `pd.DataFrame()` function, passing in their dictionary as an argument.

Each key in the dictionary becomes a column in the DataFrame, and the corresponding values form the rows. This method is efficient for converting structured data, like records or tables, into a format that is easy to manipulate and analyze in Pandas.

4. Reading a CSV file into a Pandas DataFrame is simple with the `pd.read_csv()` function, which provides a path to the CSV file.

Pandas then parses the CSV file and loads its content into a DataFrame. This function handles various file formats and delimiters, offering flexibility for different types of CSV files.

5. Knowledgeable candidates will explain that to see the first five rows of a DataFrame, they'd use the `.head()` method. This method is very useful for quickly inspecting the beginning of a dataset, enabling the user to check the data and column headers.

By default, `.head()` returns the first five rows, but they can pass a different number as an argument if they want to see more or fewer rows.

6. For this, developers would need to use the `.dtypes` attribute.

This attribute returns a series with the data type of each column, helping them understand the kind of data they're working with. This is crucial for effective data cleaning and analysis.

7. Expect candidates to explain they'd use square brackets with the column name.

For instance, if their DataFrame is named df and you want to select the column Age, you write `df['Age']`. This returns a Pandas Series containing all the values in the Age column. It's a straightforward way to access data in a single column for analysis or manipulation.

8. Experienced candidates will explain they'd use the DataFrame's ability to perform boolean indexing.

They'd write a condition inside the square brackets, such as `df[df['Age'] > 30]` to get all rows where the Age column has values greater than 30. This method returns a new DataFrame with only the rows that meet the specified condition, making it easy to analyze subsets of your data.

9. To handle missing values in a DataFrame, candidates could use:

- The `dropna()` method, which removes any rows or columns with missing values
- The `fillna()` method, which replaces missing values with a specified value
- The `interpolate()` method, which fills in missing values using interpolation

These methods provide flexible options to clean data and handle gaps due to missing entries.

10. For this, developers could use the `fillna()` function, which enables them to specify a value that will replace all the missing entries in the DataFrame.

For example, if they want to replace all NaN values with 0, they'd call `df.fillna(0)`. This function helps ensure their dataset is complete and ready for analysis by filling in the gaps with a meaningful value.

11. Experienced candidates will know that the `describe()` function in Pandas provides a summary of the basic statistics for a DataFrame.

It includes measures such as:

- Count
- Mean
- Standard deviation
- Minimum and maximum values,
- The 25th, 50th, and 75th percentiles

12. This function is particularly useful for getting a quick overview of the data's distribution and identifying any potential outliers or anomalies.

To count the number of unique values in a column, the user can recruit the `nunique()` function.

When applied to a DataFrame column, `nunique()` returns the number of distinct values present in that column. This function is helpful for understanding the diversity or variability within a column, such as counting the number of unique categories in a categorical variable.

13. Expect candidates to outline a process where they would:

- Ensure they have Matplotlib installed, as Pandas uses it for plotting
- Use the `plot.scatter()` method on their DataFrame
- Specify the columns for the x and y axes, like this: `df.plot.scatter(x='column1', y='column2')`

This generates a scatter plot, enabling users to visualize the relationship between the two variables.

Our Matplotlib test enables you to assess candidates' skills in solving situational tasks using the functionalities of this Python library.

14. Candidates should explain they'd use the `groupby()` method, which enables them to split the data into groups based on the values in one or more columns.

After grouping, they can apply aggregate functions like `mean()`, `sum()`, or `count()` to each group. For example, `df.groupby('Category').mean()` will calculate the mean of each numerical column for each category.

15. The `apply()` function enables data analysts to apply a function along an axis of the DataFrame. This means they can apply a function to each row or each column of the DataFrame; it's useful for performing complex operations that are not built into Pandas.

16. The `groupby()` function in Pandas is used to split the data into groups based on some criteria. It's often followed by an aggregation function to summarize the data.

The purpose of `groupby` is to enable users to perform operations on subsets of their data, like calculating the sum, mean, or count for each group. This is particularly useful for exploratory data analysis and for understanding patterns within your data.

17. Hierarchical indexing, or multi-level indexing, enables users to have multiple levels of indices in a DataFrame. They can create a hierarchical index by passing a list of columns to the `set_index()` method. This enables them to work with higher-dimensional data in a lower-dimensional DataFrame.

It's useful for complex data manipulations and for performing more advanced data slicing, grouping, and analysis.

18. Vectorization is the process of performing operations on entire arrays or series without using explicit loops. In Pandas, vectorized operations are performed using optimized C and Fortran libraries, making them much faster than traditional loops in Python.

The benefits of vectorization include:

- Improved performance
- Cleaner, more readable code
- Efficient data processing, especially with large datasets, by leveraging the power of NumPy

19. Expect candidates to explain they'd use the `to_excel()` method and specify the file name as an argument.

For example, `df.to_excel('output.xlsx')` saves the DataFrame df to an Excel file named 'output.xlsx'. This method also enables users to customize the sheet name and other parameters if needed.

20. For this, developers would need to use the `read_json()` function and provide the path to the JSON file as an argument.

Skilled applicants would explain that, for example, `pd.read_json('data.json')` would read the JSON file 'data.json' into a DataFrame. This function can handle different JSON formats and structures, making it easy to import JSON data for analysis and manipulation in Pandas.

21. To read and write data into a HDF5 file using Pandas, users need to use the HDFStore class and methods like `to_hdf()` and `read_hdf()`:

- **To write data**, they'd need to use `df.to_hdf('data.h5', key='df', mode='w')`, which would save the DataFrame df to an HDF5 file named 'data.h5'
- **To read data**, they'd need to use `pd.read_hdf('data.h5', 'df')` to load the DataFrame

HDF5 is particularly useful for handling large datasets efficiently.

22. Pandas is a top choice for data cleaning when dealing with datasets with inconsistencies, missing values, or which need reformatting.

For instance, if you have a CSV file with customer information that includes missing ages, incorrect date formats, and duplicate entries, you can use Pandas to handle all those issues.

Functions like `dropna()`, `fillna()`, `astype()`, and `drop_duplicates()` help clean and standardize the data for further analysis.

23. Candidates should explain that preprocessing data involves the following steps:

- Load their dataset into a DataFrame
- Handle missing values using methods like fillna() or dropna()
- Convert categorical variables into numeric using techniques like one-hot encoding (`get_dummies()`)
- Normalize or standardize numerical features

- Remove irrelevant or redundant features
- Split data into training and testing sets

24. In financial data analysis, Pandas is used to manage and analyze time-series data, such as stock prices, trading volumes, and financial ratios.

For instance, you can use Pandas to load historical stock price data, calculate moving averages, and identify trends or anomalies. With functions like `resample()` and `rolling()`, you can aggregate and smooth data to better understand market behaviors.

25. To verify the integrity of a DataFrame, candidates would:

- Start by checking for missing values using `isnull().sum()`
- Ensure data types are correct with `dtypes`
- Use `describe()` to review basic statistics and identify anomalies
- Check for duplicate rows with `duplicated()`
- Validate data ranges and consistency with logical checks and custom conditions
- Use domain knowledge to inspect sample records