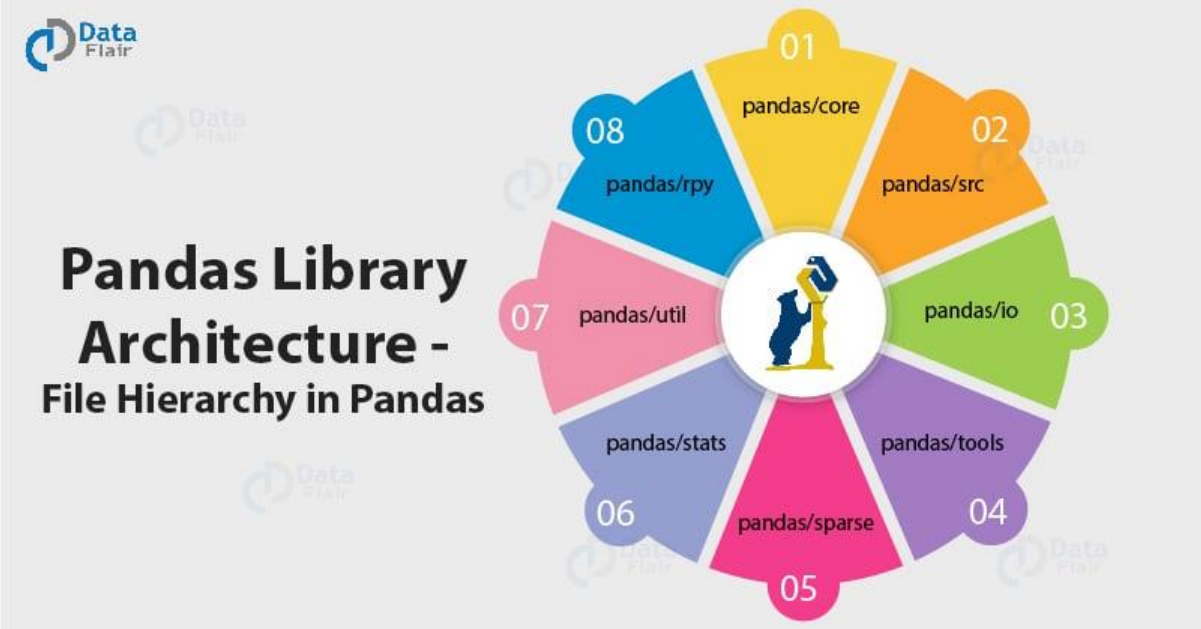
UNIT 5:

1a) In detail explain the architecture of the pandas library?

However, there are 8 types of files present in Pandas. The hierarchy of files is important to know the architecture of Pandas, so before starting with the architecture, let's explore the hierarchy.



Pandas Library Architecture

The following list gives us an idea about the hierarchy of the files within Pandas Library Architecture:

1. pandas/core

In Pandas library architecture, this part consists of basic files about the data structures present within the library. For examples, data structures – Series and DataFrames. There are various Python files within the core. The most important of them being:

- **api.py:** Important key modules which will be used later are imported using these files.
- **base.py:** This will provides the base for all the other classes present, like PandasObject and StringMIxin.
- **common.py:** It controls the common utility methods which help in handling various data structures.
- **config.py:** This helps to handle configurable objects found throughout the package.

These are the **essential python classes** which handle most of the working in the core of Pandas.

2. pandas/src

This contains algorithms which provide basic functionality to the library. The code here is usually written in C or Cython.

3. pandas/io

pandas/io, an essential part of the Pandas library architecture. This contains input and output tools which help Pandas handle files of various file formats. Essential modules found here are:

- **api.py:** This module handles various imports needed for input and output functions.
- **auth.py:** This module handles authentications and the methods dealing with it.
- **common.py:** Common functionality of input and output functions are taken care of by this module.
- **data.py:** This module helps to handle data with is input or output.

4. pandas/tools

The algorithms of pandas/tools are for auxiliary data. These help various functions like pivot, merge, join, concatenation, and other such functions for manipulating the data sets.

5. pandas/sparse

This part consists of sparse versions of various data structures like DataFrames and Series. A sparse version means that the data is mostly missing or unavailable.

6. pandas/stats

This part of the Pandas library architecture consists of a panel and linear regression and also contains moving window regression. Various statistics-related functions can be found in this portion.

7. pandas/util

Various utilities, testing tools, development can be found here. In pandas/util, classes are used to make testing and debugging any part of the library.

8. pandas/rpy

It consists of an interface to connect to R programming, called RPy2. Using Pandas with both R and Python can help you to have a much better grasp over data analysis.

1b) Discuss about features and application of pandas?

Pandas is a powerful library in Python that offers a wide range of features for data manipulation, analysis, and exploration. Its rich set of functionalities makes it suitable for various applications across different domains. Here are some key features and applications of pandas:

Features of Pandas:

1. Data Structures: Pandas provides two main data structures - Series and DataFrame. Series is a one-dimensional labeled array capable of holding any data type, while DataFrame is a two-dimensional labeled data structure with columns of potentially different types. These structures enable efficient handling and manipulation of structured data.

2. Data Cleaning and Preprocessing: Pandas offers numerous functions for data cleaning tasks, including handling missing data, removing duplicates, transforming data types, handling outliers, and dealing with inconsistent values. It allows for flexible data reshaping and reformatting, enabling users to transform and prepare their data for further analysis.

3. Data Manipulation: Pandas provides powerful tools for data manipulation, such as indexing, slicing, filtering, merging, joining, grouping, pivoting, and reshaping data. It allows for seamless data aggregation, transformation, and computation using functions like sorting, ranking, and applying custom functions across rows or columns.

4. Data Exploration and Analysis: Pandas facilitates exploratory data analysis by providing functions for statistical summary, descriptive statistics, correlation analysis, and data profiling. It enables efficient data slicing and filtering based on conditions, as well as advanced querying using SQL-like syntax. Pandas also supports time series analysis, allowing for time-based indexing and analysis of temporal data.

5. Data Visualization: While pandas itself does not provide visualization capabilities, it integrates well with other libraries like Matplotlib and Seaborn. It allows users to easily generate various types of plots, charts, and graphs to visually analyze and communicate data patterns and insights.

6. Input/Output Operations: Pandas supports reading and writing data in various formats, including CSV, Excel, SQL databases, JSON, HTML, and more. It simplifies the process of importing data from external sources and exporting data for further use or sharing.

Applications of Pandas:

1. Data Analysis and Research: Pandas is extensively used in data analysis and research projects, allowing analysts and researchers to explore, clean, manipulate, and analyze data efficiently. It provides a convenient framework for performing statistical analysis, hypothesis testing, and building predictive models.

2. Financial Analysis: Pandas is widely adopted in finance for tasks like portfolio management, risk analysis, time series analysis, and backtesting trading strategies. It enables professionals to efficiently analyze and manipulate financial data, perform calculations, and generate reports.

3. Business Analytics: Pandas finds applications in business analytics for tasks such as data exploration, customer segmentation, market research, and decision making. It enables businesses to gain insights from large datasets, identify trends, and make data-driven decisions.

4. Machine Learning: Pandas plays a crucial role in machine learning workflows. It helps with data preprocessing, feature engineering, and dataset preparation. It allows for data cleaning, transformation, and selection to ensure the quality and relevance of input data for machine learning models.

5. Data Visualization: Although pandas itself does not provide visualization capabilities, it integrates well with libraries like Matplotlib and Seaborn. Pandas' data manipulation capabilities, combined with visualization libraries, enable users to create informative plots, charts, and graphs for effective data visualization.

These are just a few examples of the features and applications of pandas. Its versatility and ease of use make it a valuable tool for data analysis and manipulation across various domains, including finance, business, research, and machine learning.

2a) What is Series? Write a program to demonstrate the process of creating a series using pandas?

The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types. We can easily convert the list, tuple, and dictionary into series using "**series**' method. The row labels of series are called the index. A Series cannot contain multiple columns. It has the following parameter:

- **data:** It can be any list, dictionary, or scalar value.
- **index:** The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default **np.arrange(n)** will be used.
- **dtype:** It refers to the data type of series.
- **copy:** It is used for copying the data.

Creating a Series:

We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

Create an Empty Series:

We can easily create an empty series in Pandas which means it will not have any value.

The syntax that is used for creating an Empty Series:

1. <series object> = pandas.Series()

The below example creates an Empty Series type object that has no values and having default datatype, i.e., **float64**.

Example

```
import pandas as pd
x = pd.Series()
print (x)
```

Output

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

Creating a Series using inputs:

We can create Series by using various inputs:

- Array
- Dict
- Scalar value

Creating Series from Array:

Before creating a Series, firstly, we have to import the **numpy** module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of **range(n)** is being passed where n defines the length of an array, i.e., [0,1,2,....**range(len(array))-1**].

Example

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
```

4. a = pd.Series(info)
5. print(a)

Output

```
0    P
1    a
2    n
3    d
4    a
5    s
dtype: object
```

Create a Series from dict

We can also create a Series from dict. **If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index**.

If index is passed, then values correspond to a particular label in the index will be extracted from the **dictionary**.

1. #**import** the pandas library
2. **import** pandas as pd
3. **import** numpy as np
4. info = {'x' : 0., 'y' : 1., 'z' : 2.}
5. a = pd.Series(info)
6. print (a)

Output

```
x    0.0
y    1.0
z    2.0
dtype: float64
```

Create a Series using Scalar:

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

1. #**import** pandas library
2. **import** pandas as pd
3. **import** numpy as np
4. x = pd.Series(4, index=[0, 1, 2, 3])
5. print (x)

Output

```
0       4
1       4
2       4
3       4
dtype: int64
```

2b) With suitable examples explain the process of indexing and selecting data in Series?

In pandas, a Series is a one-dimensional labeled array that can hold any data type. Indexing and selecting data in a Series involves accessing specific elements or subsets of the Series based on various criteria. Let's explore the process with suitable examples:

Indexing by Label:

Series can be indexed using labels associated with each element. The label can be a string, integer, or any other hashable object. Here's an example:

import pandas as pd

Create a Series

s = pd.Series([10, 20, 30, 40], index=['A', 'B', 'C', 'D'])

Accessing a single element by label

print(s['B']) # Output: 20

Accessing multiple elements by labels

print(s[['A', 'C']]) # Output:

A 10

C 30

dtype: int64

2.Indexing by Position:

Series can also be indexed using integer positions, starting from 0. This is similar to indexing in Python lists. Here's an example:

import pandas as pd

Create a Series

```
s = pd.Series([10, 20, 30, 40])

# Accessing a single element by position

print(s[1])  # Output: 20

# Accessing multiple elements by positions

print(s[[0, 2]])  # Output:

# 0    10

# 2    30

# dtype: int64
```

3.Indexing by Boolean Conditions:

Boolean indexing allows selecting elements based on a condition. Here's an example:

```
import pandas as pd

# Create a Series

s = pd.Series([10, 20, 30, 40])

# Select elements greater than 20

print(s[s > 20])  # Output:

# 2    30

# 3    40

# dtype: int64
```

4.Slicing:

Similar to Python lists, Series support slicing to select a range of elements. Here's an example:

```
import pandas as pd

# Create a Series

s = pd.Series([10, 20, 30, 40])

# Select a slice of elements
```

```
print(s[1:3])  # Output:
# 1    20
# 2    30
# dtype: int64
```

These are some of the ways to index and select data in a Series in pandas. Remember that the examples provided can be applied to more complex Series with different data types and indexes, and can also be combined to perform more advanced operations.

3a) What is a Data Frame? How is Series different from Data Frame?

A DataFrame is a two-dimensional labeled data structure in pandas that is similar to a table in a relational database or a spreadsheet. It consists of rows and columns, where each column can have a different data type. In other words, a DataFrame is a collection of Series objects that share the same index.

A Series, on the other hand, is a one-dimensional labeled array in pandas. It can be thought of as a single column of a DataFrame. While a DataFrame represents a table-like structure with rows and columns, a Series represents a single column or a single row of data.

Here are some key differences between a Series and a DataFrame:

1. Dimensions: A Series is one-dimensional, whereas a DataFrame is two-dimensional.

2. Data Structure: A Series is essentially a single column of data with an associated index, while a DataFrame is a collection of Series objects organized into columns, forming a tabular structure.

3. Data Type: Each column in a DataFrame can have a different data type, allowing for heterogeneous data. In a Series, all elements within the column have the same data type.

4. Flexibility: DataFrames provide flexibility in handling and manipulating structured data due to their tabular nature. They allow for easy addition, deletion, and modification of columns, as well as operations across multiple

columns. Series, on the other hand, are more focused on handling a single column or row of data.

5. Indexing: Series have their own index, which provides labels for each element. In a DataFrame, there are both row and column indices, enabling efficient data access, selection, and alignment.

6. Functionality: DataFrames offer a wider range of functionalities compared to Series. While Series provide basic data manipulation and analysis operations, DataFrames provide additional features such as merging, joining, grouping, pivoting, and statistical summaries. DataFrames also integrate well with other pandas functionalities, such as data cleaning, preprocessing, and visualization.

In summary, a DataFrame is a two-dimensional data structure with rows and columns, while a Series is a one-dimensional data structure representing a single column or row. DataFrames are more versatile and provide a comprehensive set of operations for data manipulation and analysis, while Series are more focused on handling individual columns or rows of data.

3b)Write a program to create a DataFrame from a csv file? Explain about various features of the Data Frame?

Certainly! To create a DataFrame from a CSV file, you can use the `read_csv()` function provided by the pandas library. This function reads the contents of a CSV file and converts it into a DataFrame. Here's an example program:

```python
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the DataFrame
print(df)
```

In the above program, `data.csv` is the name of the CSV file you want to read. Make sure that the file is in the same directory as your Python script or notebook.

After executing the program, the contents of the CSV file will be displayed as a DataFrame.

Now, let's discuss various features of a DataFrame:

1. Data Structure: A DataFrame is a two-dimensional labeled data structure with columns and rows. It provides a tabular representation of data, similar to a spreadsheet or a SQL table.

2. Columns and Index: DataFrames have column names that represent different variables or attributes of the data. Each column is a Series within the DataFrame. DataFrames also have an index that labels the rows, allowing for efficient data access and alignment.

3. Data Manipulation: DataFrames offer a wide range of operations for data manipulation. You can add, delete, or modify columns, rename columns, and apply functions to columns or rows. DataFrames allow for filtering, sorting, grouping, merging, and reshaping data, enabling you to perform complex data transformations.

4. Data Cleaning: DataFrames provide functions to handle missing values, duplicate records, and inconsistent data. You can drop or fill missing values, remove duplicates, and perform data validation and transformation to ensure data quality.

5. Indexing and Selection: DataFrames support various indexing and selection techniques to access specific rows or columns. You can use integer-based indexing, label-based indexing, or boolean indexing to retrieve subsets of data based on specific criteria.

6. Statistical Summary: DataFrames offer functions to compute descriptive statistics for numerical columns, such as mean, median, standard deviation, and quantiles. You can generate summary statistics for the entire DataFrame or specific columns.

7. Data Visualization: DataFrames can be easily integrated with visualization libraries like Matplotlib and Seaborn. You can create various plots, charts, and graphs to visualize and analyze data patterns.

8. Input/Output Operations: DataFrames support reading and writing data in various formats, including CSV, Excel, SQL databases, JSON, and more. You can import data from external sources into a DataFrame and export DataFrame data for further analysis or sharing.

These are some of the key features of a DataFrame. With these features, DataFrames provide a powerful tool for data manipulation, analysis, and exploration in various domains such as data science, finance, business analytics, and more.

4a) How dropna() and fillna() are useful in handling missing data of DataFrame?

The `dropna()` and `fillna()` functions are useful methods in pandas for handling missing data (NaN values) in a DataFrame. Here's an explanation of how they work and their usefulness:

1. dropna():

The `dropna()` function is used to remove rows or columns from a DataFrame that contain missing values. It provides flexibility in handling missing data by allowing you to specify different conditions for dropping.

Syntax: `DataFrame.dropna(axis=0, how='any', inplace=False)`

Parameters:

- `axis`: Specifies the axis along which missing values should be dropped. Use `0` for rows and `1` for columns.

- `how`: Specifies the condition for dropping. It can take the values `'any'` or `'all'`. `'any'` drops a row or column if it contains at least one missing value, while `'all'` drops a row or column only if all values are missing.

- `inplace`: Specifies whether to modify the DataFrame in-place or return a new DataFrame. By default, it is set to `False`, which returns a new DataFrame.

Example:

```python
import pandas as pd
# Create a DataFrame with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, None, 28, 36],
    'Country': ['USA', None, 'UK', 'Australia'],
    'Salary': [5000, None, None, 8000]
}
df = pd.DataFrame(data)
# Drop rows with any missing values
df_dropped = df.dropna(axis=0, how='any')
print("DataFrame after dropping rows with missing values:")
print(df_dropped)
```

Output:

```
DataFrame after dropping rows with missing values:
    Name   Age    Country  Salary
0  Alice  25.0        USA  5000.0
3  David  36.0  Australia  8000.0
```

In the above example, the `dropna()` function is used to drop rows with any missing values (`how='any'`) from the DataFrame. As a result, rows with missing values are removed, and a new DataFrame (`df_dropped`) without those rows is returned.

2. fillna():

The `fillna()` function is used to fill missing values in a DataFrame with specified values or with computed values. It allows you to replace NaN values with a constant, a specific value, the mean, median, or any other calculated value.

Syntax: `DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None)`

Parameters:

- `value`: Specifies the value to use for filling the missing values.

- `method`: Specifies the method to use for filling missing values. It can take values like `'ffill'` (forward fill) or `'bfill'` (backward fill).

- `axis`: Specifies the axis along which missing values should be filled. Use `0` for rows and `1` for columns.

- `inplace`: Specifies whether to modify the DataFrame in-place or return a new DataFrame. By default, it is set to `False`, which returns a new DataFrame.

- `limit`: Specifies the maximum number of consecutive missing values to fill.

Example:

```python

import pandas as pd

# Create a DataFrame with missing values

data = {

    'Name': ['Alice', 'Bob', 'Charlie', 'David'],

    'Age': [25, None, 28, None],

    'Country': ['USA', None, 'UK', 'Australia'],

    'Salary': [5000, None, None,

8000]
```

}

df = pd.DataFrame(data)

Fill missing values with a specific value

df_filled = df.fillna(value=0)

print("DataFrame after filling missing values with 0:")

print(df_filled)

```

Output:

```

DataFrame after filling missing values with 0:

```
      Name   Age    Country  Salary
0   Alice  25.0        USA  5000.0
1     Bob   0.0       None     0.0
2  Charlie  28.0         UK     0.0
3    David   0.0  Australia  8000.0
```

In the above example, the `fillna()` function is used to replace missing values with `0` in the DataFrame. The resulting DataFrame (`df_filled`) contains the original values where they were present and `0` where missing values were filled.

4b) How to add a new column to the existing DataFrame? Explain with a suitable code?

Method #1: By declaring a new list as a column.

- Python3

```
# Import pandas package
import pandas as pd

# Define a dictionary containing Students data
data = {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],
```

```
        'Height': [5.1, 6.2, 5.1, 5.2],
        'Qualification': ['Msc', 'MA', 'Msc', 'Msc']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)

# Declare a list that is to be converted into a column
address = ['Delhi', 'Bangalore', 'Chennai', 'Patna']

# Using 'Address' as the column name
# and equating it to the list
df['Address'] = address

# Observe the result
print(df)
```

Output:

	Name	Height	Qualification	Address
0	Jai	5.1	Msc	Delhi
1	Princi	6.2	MA	Bangalore
2	Gaurav	5.1	Msc	Chennai
3	Anuj	5.2	Msc	Patna

Note that the length of your list should match the length of the index column otherwise it will show an error.

5a) How to access rows and columns of DataFrame? Write a program to drop an existing column from the DataFrame?

To access rows and columns of a DataFrame in pandas, you can use various indexing and selection techniques. Here's an example program that demonstrates how to access rows and columns, as well as how to drop an existing column from a DataFrame:

```python

import pandas as pd

# Create a DataFrame
```

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [5000, 6000, 7000]
}
df = pd.DataFrame(data)
# Accessing rows by index
row = df.loc[1]  # Access the second row (index 1)
print("Row at index 1:")
print(row)
print()
# Accessing columns by name
column = df['Age']  # Access the 'Age' column
print("Age column:")
print(column)
print()
# Dropping an existing column
df_dropped = df.drop('Salary', axis=1)  # Drop the 'Salary' column
print("DataFrame after dropping the 'Salary' column:")
print(df_dropped)
```

Output:

```
Row at index 1:
Name      Bob
Age        30
Salary    6000
```

Name: 1, dtype: object

Age column:

0 25

1 30

2 35

Name: Age, dtype: int64

DataFrame after dropping the 'Salary' column:

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

In the above example, we first create a DataFrame `df` with three columns: 'Name', 'Age', and 'Salary'. To access a specific row, we use the `loc` indexer and provide the row index as an argument (`df.loc[1]`). This returns a Series object representing the second row of the DataFrame.

To access a specific column, we use the column name enclosed in square brackets (`df['Age']`). This returns a Series object representing the 'Age' column of the DataFrame.

To drop an existing column from the DataFrame, we use the `drop()` function. We specify the column name as the first argument and set the `axis` parameter to `1` to indicate that we want to drop a column. The result is a new DataFrame `df_dropped` without the dropped column.

Note that the original DataFrame `df` remains unchanged unless you set the `inplace` parameter to `True` in the `drop()` function.

5b) Write a program to calculate aggregate values of a numpy array?

To calculate aggregate values of a NumPy array, you can utilize various functions provided by NumPy. Here's an example program that demonstrates the calculation of aggregate values:

```python
import numpy as np
```

```
# Create a NumPy array
arr = np.array([5, 2, 8, 3, 1, 7, 6])
# Calculate aggregate values
minimum = np.min(arr)
maximum = np.max(arr)
mean = np.mean(arr)
median = np.median(arr)
sum = np.sum(arr)
# Display the aggregate values
print("Minimum:", minimum)
print("Maximum:", maximum)
print("Mean:", mean)
print("Median:", median)
print("Sum:", sum)
```

Output:

```
Minimum: 1
Maximum: 8
Mean: 4.571428571428571
Median: 5.0
Sum: 32
```

In the above example, we first create a NumPy array `arr` containing a set of numbers. We then use the following functions to calculate aggregate values:

- `np.min(arr)`: Calculates the minimum value in the array.
- `np.max(arr)`: Calculates the maximum value in the array.

- `np.mean(arr)`: Calculates the mean (average) value of the array.

- `np.median(arr)`: Calculates the median value of the array.

- `np.sum(arr)`: Calculates the sum of all elements in the array.

These functions allow you to compute various aggregate values based on the elements in the NumPy array. Depending on your requirements, you can use additional functions such as `np.std()` for standard deviation, `np.var()` for variance, `np.percentile()` for percentiles, and more.

By utilizing these aggregate functions, you can gain insights into the statistical properties of the data contained in a NumPy array.

6a) Give the syntax to create a DataFrame using i) List of lists ii) List of Dictionaries

To create a DataFrame in pandas, you can use the following syntax for different data structures:

i) List of lists:

```python
import pandas as pd

data = [['Alice', 25], ['Bob', 30], ['Charlie', 35]]

df = pd.DataFrame(data, columns=['Name', 'Age'])
```

In this case, you provide a list of lists `data` where each inner list represents a row of data. The `columns` parameter specifies the column names for the DataFrame.

ii) List of Dictionaries:

```python
import pandas as pd

data = [{'Name': 'Alice', 'Age': 25}, {'Name': 'Bob', 'Age': 30}, {'Name': 'Charlie', 'Age': 35}]

df = pd.DataFrame(data)
```

Here, you provide a list of dictionaries `data` where each dictionary represents a row of data. The keys of the dictionaries correspond to the column names of the DataFrame.

Both approaches allow you to create a DataFrame from structured data. Choose the method that best fits your data format and ease of use.

6b) How to add columns names to the existing DataFrame which was created using list of lists

To add column names to an existing DataFrame created using a list of lists, you can use the `columns` attribute of the DataFrame and assign a list of column names to it. Here's an example:

```python
import pandas as pd

data = [['Alice', 25], ['Bob', 30], ['Charlie', 35]]

df = pd.DataFrame(data)  # Create DataFrame without column names

column_names = ['Name', 'Age']

df.columns = column_names  # Assign column names to the DataFrame

print(df)
```

Output:

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

In the above example, we first create a DataFrame `df` using a list of lists `data` without providing column names. Then, we define a list of column names `column_names`. Finally, we assign the `column_names` list to the `columns` attribute of the DataFrame `df` using the assignment operator (`=`). The DataFrame is then printed, showing the column names along with the data.

By assigning column names to an existing DataFrame, you can provide meaningful labels to the columns and enhance the readability and interpretability of your data.

7) Create a sample DataFrame for student report which contains atleast 6 columns.Display the aggregate values of each column using python code?

Certainly! Here's an example of creating a sample DataFrame for a student report with six columns and displaying the aggregate values using Python code:

```python
import pandas as pd
import numpy as np
# Create a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
    'Age': [20, 21, 19, 22, 20],
    'Gender': ['Female', 'Male', 'Male', 'Male', 'Female'],
    'Math': [85, 90, 75, 95, 80],
    'Science': [92, 88, 78, 85, 90],
    'English': [87, 85, 90, 92, 88]
}
df = pd.DataFrame(data)
# Display the DataFrame
print("Student Report:")
print(df)
print()
# Calculate aggregate values of each column
aggregate_values = {
    'Column': df.columns,
    'Minimum': df.min(),
```

```
    'Maximum': df.max(),
    'Mean': df.mean(),
    'Median': df.median(),
    'Sum': df.sum()
}
aggregate_df = pd.DataFrame(aggregate_values)
# Display the aggregate values
print("Aggregate Values:")
print(aggregate_df)
```

Output:

```
Student Report:
      Name  Age  Gender  Math  Science  English
0    Alice   20  Female    85       92       87
1      Bob   21    Male    90       88       85
2  Charlie   19    Male    75       78       90
3    David   22    Male    95       85       92
4    Emily   20  Female    80       90       88
Aggregate Values:
    Column  Minimum Maximum   Mean  Median   Sum
0     Name    Alice   Emily    NaN     NaN   NaN
1      Age       19      22  20.40    20.0   102
2   Gender   Female    Male    NaN     NaN   NaN
3     Math       75      95  85.00    85.0   425
4  Science       78      92  86.60    88.0   433
5  English       85      92  88.40    88.0   442
```

In the above code, we first create a sample DataFrame `df` with six columns: 'Name', 'Age', 'Gender', 'Math', 'Science', and 'English'. We then display the DataFrame to see the student report.

Next, we calculate the aggregate values for each column using functions like `min()`, `max()`, `mean()`, `median()`, and `sum()` on the DataFrame. We store these values in a dictionary `aggregate_values` and create a new DataFrame `aggregate_df` from it.

Finally, we display the aggregate values DataFrame, which shows the minimum, maximum, mean, median, and sum for each column.

This code allows you to create a sample DataFrame for a student report and calculate aggregate values for each column, providing insights into the data.

8a) How Panel is different from the DataFrame? In detail explain about creating a Panel?

In pandas, a Panel is a three-dimensional data structure that can be seen as a container of multiple DataFrames. It is designed to handle data with three dimensions: items, major_axis, and minor_axis. The major use case for Panels is when you have data that is naturally three-dimensional, such as financial data with multiple assets (items), dates (major_axis), and attributes (minor_axis).

Here's a detailed explanation of the differences between a Panel and a DataFrame:

1. Dimensions:

- DataFrame: A DataFrame is a two-dimensional data structure with rows and columns. It can be seen as a table where each column represents a variable, and each row represents an observation.

- Panel: A Panel is a three-dimensional data structure. It adds an additional dimension, called items, to the rows and columns of a DataFrame. This allows for storing and manipulating multiple DataFrames within a single object.

2. Structure:

- DataFrame: In a DataFrame, the data is aligned in a tabular format, with rows and columns. Each column has a unique name, and data is accessed using column names or row indices.

- Panel: A Panel is a collection of multiple DataFrames stacked along the items axis. It consists of a dictionary-like structure, where each item

represents a DataFrame. Each DataFrame in a Panel has its own row and column structure, and they are aligned based on their major_axis and minor_axis values.

3. Flexibility:

- DataFrame: DataFrames are widely used and versatile data structures. They can handle various types of data and are suitable for most analytical tasks. DataFrames allow for easy data manipulation, filtering, and analysis.

- Panel: Panels are less commonly used compared to DataFrames and are specifically designed for handling three-dimensional data. They provide an additional level of organization and can be useful for certain applications, such as multi-dimensional time series analysis or multi-attribute data analysis.

Creating a Panel:

To create a Panel, you can use the `pd.Panel()` constructor by passing a dictionary of DataFrames or a three-dimensional ndarray. Here's an example:

```python
import pandas as pd
import numpy as np
# Create a Panel using a dictionary of DataFrames
data = {
    'Item1': pd.DataFrame(np.random.randn(4, 3)),
    'Item2': pd.DataFrame(np.random.randn(4, 2))
}
panel = pd.Panel(data)
# Display the Panel
print(panel)
```

Output:

```
```

<class 'pandas.core.panel.Panel'>

Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)

Items axis: Item1 to Item2

Major_axis axis: 0 to 3

Minor_axis axis: 0 to 2

```

In this example, we create a Panel `panel` by passing a dictionary `data` containing two DataFrames, 'Item1' and 'Item2'. Each DataFrame has its own row and column structure. The resulting Panel displays the dimensions of the data, including the number of items, major_axis (rows), and minor_axis (columns).

8b) How to drop a column(s) or row(s) in DataFrame? Explain with a sample code?

In pandas, you can drop columns or rows from a DataFrame using the `drop()` method. The `drop()` method allows you to remove specific columns or rows based on their labels or indices. Here's a sample code that demonstrates how to drop columns or rows from a DataFrame:

```python

import pandas as pd

# Create a sample DataFrame

data = {

    'Name': ['Alice', 'Bob', 'Charlie'],

    'Age': [25, 30, 35],

    'Salary': [5000, 6000, 7000]

}

df = pd.DataFrame(data)

# Drop a column by label

df_dropped_column = df.drop('Salary', axis=1)

# Drop multiple columns by labels

columns_to_drop = ['Age', 'Salary']

df_dropped_columns = df.drop(columns_to_drop, axis=1)

```
# Drop a row by index
df_dropped_row = df.drop(1)
# Drop multiple rows by indices
rows_to_drop = [0, 2]
df_dropped_rows = df.drop(rows_to_drop)
# Print the resulting DataFrames
print("DataFrame after dropping 'Salary' column:")
print(df_dropped_column)
print()
print("DataFrame after dropping 'Age' and 'Salary' columns:")
print(df_dropped_columns)
print()
print("DataFrame after dropping row at index 1:")
print(df_dropped_row)
print()
print("DataFrame after dropping rows at indices 0 and 2:")
print(df_dropped_rows)
```

Output:

```
DataFrame after dropping 'Salary' column:
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
DataFrame after dropping 'Age' and 'Salary' columns:
      Name
```

0   Alice

1   Bob

2 Charlie

DataFrame after dropping row at index 1:

     Name Age Salary

0   Alice   25   5000

2 Charlie   35   7000

DataFrame after dropping rows at indices 0 and 2:

  Name Age Salary

1 Bob   30   6000

```

In the above code, we first create a DataFrame `df` with three columns: 'Name', 'Age', and 'Salary'. Then, we demonstrate different scenarios of dropping columns or rows:

- To drop a column, we use the `drop()` method and provide the column label as the first argument and set `axis=1`. This removes the 'Salary' column from the DataFrame and assigns the result to `df_dropped_column`.

- To drop multiple columns, we pass a list of column labels as the first argument and set `axis=1`. This removes the 'Age' and 'Salary' columns from the DataFrame and assigns the result to `df_dropped_columns`.

- To drop a row, we provide the row index as the argument of `drop()`. This removes the row at index 1 from the DataFrame and assigns the result to `df_dropped_row`.

- To drop multiple rows, we pass a list of row indices as the argument of `drop()`. This removes the rows at indices 0 and 2 from the DataFrame and assigns the result to `df_dropped_rows`.

9) Create a sample DataFame for a company which contains stores all over India? Explain about sorting and ranking using the above company DataFrame?

Certainly! Here's an example of creating a sample DataFrame for a company with stores all over India and an explanation of sorting and ranking using the DataFrame:
```

```python
import pandas as pd
# Create a sample DataFrame for a company
data = {
    'Store ID': ['S001', 'S002', 'S003', 'S004', 'S005'],
    'Store Name': ['Store A', 'Store B', 'Store C', 'Store D', 'Store E'],
    'City': ['Delhi', 'Mumbai', 'Chennai', 'Kolkata', 'Bangalore'],
    'Region': ['North', 'West', 'South', 'East', 'South'],
    'Revenue (in lakhs)': [50, 75, 60, 45, 80]
}
df = pd.DataFrame(data)
# Display the DataFrame
print("Company DataFrame:")
print(df)
print()
# Sorting by a column
df_sorted = df.sort_values('Revenue (in lakhs)', ascending=False)
# Display the sorted DataFrame
print("Sorted DataFrame:")
print(df_sorted)
print()
# Ranking by a column
df['Rank'] = df['Revenue (in lakhs)'].rank(ascending=False)
# Display the DataFrame with rankings
print("Ranked DataFrame:")
print(df)
```

Output:

```
Company DataFrame:
  Store ID Store Name       City Region  Revenue (in lakhs)
0     S001    Store A      Delhi  North                 50
1     S002    Store B     Mumbai   West                  75
2     S003    Store C    Chennai  South                 60
3     S004    Store D    Kolkata   East                 45
4     S005    Store E  Bangalore  South                  80
Sorted DataFrame:
  Store ID Store Name       City Region  Revenue (in lakhs)
4     S005    Store E  Bangalore  South                  80
1     S002    Store B     Mumbai   West                  75
2     S003    Store C    Chennai  South                 60
0     S001    Store A      Delhi  North                 50
3     S004    Store D    Kolkata   East                 45
Ranked DataFrame:
  Store ID Store Name       City Region  Revenue (in lakhs)  Rank
0     S001    Store A      Delhi  North                 50   3.0
1     S002    Store B     Mumbai   West                  75   2.0
2     S003    Store C    Chennai  South                 60   3.0
3     S004    Store D    Kolkata   East                 45   4.0
4     S005    Store E  Bangalore  South                  80   1.0
```

In the above code, we create a sample DataFrame `df` for a company with stores all over India. The DataFrame has columns for 'Store ID', 'Store Name', 'City', 'Region', and 'Revenue (in lakhs)'.

- Sorting: To sort the DataFrame based on a specific column, we use the `sort_values()` method. In the example, we sort the DataFrame by 'Revenue (in lakhs)' in descending order by setting `ascending=False`. The sorted DataFrame is stored in `df_sorted`.

- Ranking: To assign ranks to the DataFrame based on a column, we use the `rank()` method. In the example, we create a new column 'Rank' that contains the rankings based on 'Revenue (in lakhs)'. The `rank()` function assigns a higher rank to a higher value by default. The resulting DataFrame with the rankings is displayed.

By sorting the DataFrame,

you can rearrange the rows based on a specific column in either ascending or descending order. Ranking allows you to assign a rank to each row based on a column's value, which can be useful for analyzing performance or creating leaderboards.

10a) How loc() and iloc() functions are useful in accessing data from DataFrame.

In pandas, the `loc()` and `iloc()` functions are used to access data from a DataFrame by specifying the row(s) and column(s) you want to retrieve. Here's an explanation of how `loc()` and `iloc()` functions are useful in accessing data:

1. `loc()` function:
    - Syntax: `df.loc[row_indexer, column_indexer]`
    - The `loc()` function is primarily label-based, meaning it is used to access data by specifying the row and column labels explicitly.
    - You can provide either a single label or a list of labels to select specific rows or columns.
    - It allows you to slice the DataFrame based on label-based indexing.
    - You can also use boolean indexing with `loc()` to filter rows based on conditions.
    - Example:

```
import pandas as pd

# Create a sample DataFrame

data = {

    'Name': ['Alice', 'Bob', 'Charlie'],

    'Age': [25, 30, 35],

    'Salary': [5000, 6000, 7000]
```

```
}
df = pd.DataFrame(data, index=['A', 'B', 'C'])
# Accessing data using loc()
print(df.loc['A'])  # Access a single row
print(df.loc[['A', 'C']])  # Access multiple rows
print(df.loc['A', 'Salary'])  # Access a single element
print(df.loc['A':'B', 'Age':'Salary'])  # Slicing the DataFrame
#output:
->Name      Alice
Age         25
Salary     5000
Name: A, dtype: object

->Name  Age  Salary
A    Alice   25    5000
C  Charlie   35    7000
->5000
 ->  Age  Salary
A   25    5000
B   30    6000
```

`iloc()` function:

- Syntax: `df.iloc[row_indexer, column_indexer]`
- The `iloc()` function is primarily integer-based, meaning it is used to access data by specifying the integer-based position of rows and columns.
- You can provide either a single integer or a range of integers to select specific rows or columns.
- It allows you to slice the DataFrame based on integer-based indexing.
- Example:

```
import pandas as pd
# Create a sample DataFrame
```

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [5000, 6000, 7000]
}
df = pd.DataFrame(data)
# Accessing data using iloc()
print(df.iloc[0])  # Access a single row
print(df.iloc[[0, 2]])  # Access multiple rows
print(df.iloc[0, 2])  # Access a single element
print(df.iloc[0:2, 1:3])  # Slicing the DataFrame
#output:
->Name      Alice
Age          25
Salary     5000
Name: A, dtype: object
->Name  Age  Salary
A    Alice   25    5000
C  Charlie   35    7000
->5000
 ->  Age  Salary
A   25    5000
B   30    6000
```

In both cases, you can use **`loc()`** and **`iloc()`** functions to access rows, columns, or individual elements of a DataFrame based on label-based or integer-based indexing. It provides flexibility in retrieving data from the DataFrame depending on your specific requirements.

10b) Write a program to demonstrate the concept of reindexing in DataFrame?

Certainly! Here's a program that demonstrates the concept of reindexing in a DataFrame:

```python
import pandas as pd
# Create a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [5000, 6000, 7000]
}
df = pd.DataFrame(data)
# Display the original DataFrame
print("Original DataFrame:")
print(df)
print()
# Reindexing the DataFrame
new_index = ['A', 'B', 'C']
df_reindexed = df.reindex(new_index)
# Display the reindexed DataFrame
print("Reindexed DataFrame:")
print(df_reindexed)
print()
Output:
Original DataFrame:
      Name  Age  Salary
0   Alice   25    5000
1     Bob   30    6000
2  Charlie   35    7000
Reindexed DataFrame:
      Name   Age  Salary
A    Alice  25.0  5000.0
B      Bob  30.0  6000.0
C  Charlie  35.0  7000.0
```

In the above code, we start by creating a sample DataFrame `df` with three columns: 'Name', 'Age', and 'Salary'.

Next, we demonstrate the concept of reindexing by using the `reindex()` method. We define a new index called `new_index` as `['A', 'B', 'C']`. By calling `df.reindex(new_index)`, we create a new DataFrame `df_reindexed` where the rows are rearranged according to the new index. Note that the missing values are filled with NaN (Not a Number) by default.

Finally, we print the original DataFrame and the reindexed DataFrame to observe the changes.

Reindexing allows you to change the row index labels of a DataFrame, which can be useful when you want to align the DataFrame with a new index or create a DataFrame with missing values filled or dropped based on the new index.