NumPy and Pandas

1. Arrays and DataFrames (NumPy Arrays, Pandas DataFrame)

- NumPy Arrays: Arrays are the core of NumPy, which are collections of elements of the same type. They allow for mathematical operations on large datasets and come with a wide array of methods for manipulating them, such as reshaping, slicing, and performing vectorized operations.
- Pandas DataFrames: Pandas builds on NumPy, but adds a tabular structure. DataFrames allow for more complex operations on data with multiple types (like integers, strings, etc.), resembling Excel-like tables with rows and columns. Each column can have a different data type, which makes it more versatile.

2. Data Types and Conversion (NumPy dtypes, Pandas astype)

- NumPy dtypes: NumPy has a fixed type for each array (dtype), such as int, float, or string. This improves efficiency, but careful attention is needed to ensure compatibility between types during operations.
- Pandas astype: In Pandas, each column in a DataFrame has a specific dtype, but it's easy to convert them using the .astype() method. For example, converting strings to integers or floats, or vice versa, to allow operations like addition or comparisons.

3. Indexing, Selection, and Slicing (Across both libraries)

- NumPy: NumPy allows for flexible selection and slicing of arrays. You can select rows, columns, or elements with conditions, such as using integer indices or boolean masks.
- Pandas: Pandas also provides powerful indexing through labels or conditions, like using .loc[], .iloc[], or boolean indexing. This is useful for retrieving or updating specific subsets of data, like extracting rows where a certain condition holds true.

4. Data Cleaning (Handling Missing Data in Pandas)

- Missing data is a common issue in real-world datasets. Pandas provides functions like .isnull(), .fillna(), and .dropna() to detect, replace, or remove missing values. This step is crucial for accurate analysis because many algorithms cannot handle NaN values.

5. Mathematical Operations (NumPy Array operations, Pandas Statistical Methods)

- NumPy: NumPy supports efficient mathematical operations, such as addition, subtraction, and more advanced functions like dot products and matrix multiplications.
- Pandas: Pandas also provides built-in statistical methods, such as .mean(), .sum(), .min(), and .max(), allowing for fast computations over entire columns or rows.

6. Broadcasting (NumPy) and Data Alignment (Pandas)

- NumPy Broadcasting: Broadcasting allows NumPy to perform operations on arrays of different shapes, automatically expanding the smaller array to match the size of the larger one.
- Pandas Data Alignment: When performing operations on DataFrames, Pandas automatically aligns data based on their index or column labels. This feature ensures that operations between two DataFrames only apply to matching rows or columns.

7. Grouping and Aggregation (Pandas GroupBy)

- GroupBy: This function allows data to be grouped by one or more columns, after which aggregation functions (like sum, mean, count, etc.) can be applied. This is often used for summarizing data, like getting the average sales per region or counting occurrences of different values.

8. Merging, Joining, and Concatenating Data (Pandas)

- Merging: Similar to SQL joins, merging in Pandas allows you to combine two DataFrames based on a common column (or index).
- Concatenation: Concatenation stacks DataFrames either vertically (adding rows) or horizontally (adding columns) to combine multiple

datasets.
- Joining: Pandas joins help combine DataFrames with similar indices or columns, providing flexibility to include all or only common data (like inner or outer joins).

9. Pivot Tables and Cross-tabulation (Pandas)

- Pivot Tables: Pivot tables allow for reformatting data into a summary table by applying aggregation functions. This is useful for transforming data into different views for analysis.
- Cross-tabulation: This is a simple table of counts of occurrences between two categorical variables, often used in contingency analysis or comparing different categories.

10. Time Series Analysis (Pandas Time Series Features)

- Time Series: Pandas has extensive support for handling and manipulating time series data, including converting columns to DateTime format, resampling data to different time intervals, and calculating rolling statistics. Time series features include datetime indices, date offsets, and period ranges for efficient handling of time-based data.

11. Linear Algebra Functions (NumPy)

- NumPy includes a comprehensive suite of linear algebra functions like matrix multiplication (np.dot()), solving linear systems, finding eigenvalues and eigenvectors, performing matrix decompositions (like QR, LU), and more.

12. Statistical Functions (NumPy and Pandas)

- NumPy: It provides a wide range of statistical functions like mean, median, standard deviation, variance, and correlations, which are used for analyzing the data distributions.
- Pandas: Pandas extends this by enabling quick application of these functions to DataFrame columns, helping summarize key aspects of datasets with one line of code.

13. Visualization Integration (Pandas with Matplotlib/Seaborn)

- Pandas has built-in support for plotting data, but often works better with external libraries like Matplotlib and Seaborn. You can quickly generate plots such as line graphs, bar charts, histograms, and scatter plots to visualize trends, distributions, and relationships in your data.

14. Performance Tuning (Efficient Operations in Pandas)

- Performance optimization is critical when working with large datasets. Pandas offers several ways to make data operations faster, such as vectorized operations, avoiding loops, optimizing memory usage, and using efficient file formats like .parquet or .feather for saving data.

15. Advanced Data Manipulation (NumPy Advanced Indexing, Pandas MultiIndex and Querying)

- NumPy Advanced Indexing: NumPy supports various forms of indexing beyond basic slicing, including boolean indexing, fancy indexing (using arrays of indices), and multidimensional selections.
- Pandas MultiIndex: Pandas allows for hierarchical indexing (MultiIndex), where you can have multiple levels of indexing, useful for dealing with complex data structures like time series data with categories.
- Querying: Pandas .query() function allows querying of DataFrames using a SQL-like syntax to filter data more efficiently than with boolean conditions alone.

NumPy and Pandas

NumPy and Pandas, including code examples for each concept and detailed explanations of the code.

1. Arrays and DataFrames (NumPy Arrays, Pandas DataFrame)

Code Example:

```
import numpy as np

import pandas as pd
```

```
# NumPy Array

array = np.array([1, 2, 3, 4])

print("NumPy Array:", array)



# Pandas DataFrame

data = {'Name': ['John', 'Anna', 'Peter'], 'Age': [28, 24, 35]}

df = pd.DataFrame(data)

print("\nPandas DataFrame:\n", df)
```

Explanation:

- NumPy Array: The array contains integers [1, 2, 3, 4]. Arrays are efficient for numerical operations.
- Pandas DataFrame: A DataFrame is created using a dictionary, where 'Name' and 'Age' are columns. The DataFrame resembles a table with rows and columns, similar to an Excel spreadsheet.

2. Data Types and Conversion (NumPy dtypes, Pandas astype)

Code Example:

```
# NumPy Data Type Conversion

array = np.array([1.2, 2.5, 3.8])

int_array = array.astype(int)

print("Converted NumPy Array to Integer:", int_array)



# Pandas DataFrame Type Conversion

df['Age'] = df['Age'].astype(float)

print("\nConverted Pandas Age Column to Float:\n", df)
```

Explanation:

- NumPy astype: The array of floats is converted to integers using .astype(), truncating the decimal part.
- Pandas astype: The 'Age' column is converted from integers to floats. This is useful when a column's data type needs to match specific operations.

3. Indexing, Selection, and Slicing (Across both libraries)

Code Example:

```
# NumPy Slicing

array = np.array([10, 20, 30, 40, 50])

print("Sliced NumPy Array:", array[1:4])



# Pandas DataFrame Selection

print("\nPandas DataFrame Selection:")

print(df.loc[1])  # Select row by index

print(df['Name'])  # Select column by name
```

Explanation:

- NumPy Slicing: The array is sliced from index 1 to 3 (inclusive), returning [20, 30, 40].
- Pandas Selection: .loc[] selects the row at index 1, and df['Name'] selects the entire 'Name' column.

4. Data Cleaning (Handling Missing Data in Pandas)

Code Example:

```
# Create a DataFrame with missing data

data = {'Name': ['John', 'Anna', None], 'Age': [28, None, 35]}

df = pd.DataFrame(data)



# Handle missing data

df_filled = df.fillna({'Name': 'Unknown', 'Age': 0})
```

```
print("DataFrame after handling missing values:\n", df_filled)
```

Explanation:

- Handling Missing Data: fillna() replaces missing values with specified values: 'Unknown' for missing names and 0 for missing ages. This prevents errors when performing operations on the dataset.

5. Mathematical Operations (NumPy Array operations, Pandas Statistical Methods)

Code Example:

```
# NumPy Mathematical Operations

array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

sum_array = np.add(array1, array2)

print("Sum of NumPy Arrays:", sum_array)



# Pandas Statistical Methods

mean_age = df_filled['Age'].mean()

print("\nMean Age in DataFrame:", mean_age)
```

Explanation:

- NumPy Mathematical Operations: The sum of two arrays is calculated element-wise using np.add().
- Pandas Statistical Methods: The mean() function computes the average of the 'Age' column.

6. Broadcasting (NumPy) and Data Alignment (Pandas)

Code Example:

```
# NumPy Broadcasting

array = np.array([1, 2, 3])

broadcast_result = array + 5  # Adds 5 to each element in the array
```

```
print("Broadcasting Result:", broadcast_result)


# Pandas Data Alignment

df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]}, index=[1, 2])

sum_df = df1 + df2  # Aligns by index before adding

print("\nDataFrame Alignment Result:\n", sum_df)
```

Explanation:

- NumPy Broadcasting: The scalar value 5 is added to every element in the array due to broadcasting.
- Pandas Data Alignment: The DataFrames df1 and df2 are aligned by their index, and the values are added.

7. Grouping and Aggregation (Pandas GroupBy)

Code Example:

```
# Grouping by 'Age' and Aggregating the count of 'Name'

grouped = df_filled.groupby('Age')['Name'].count()

print("\nGrouped DataFrame by Age:\n", grouped)
```

Explanation:

- GroupBy: Groups the data by 'Age' and counts the number of occurrences in the 'Name' column. This is useful for summarizing the dataset, such as understanding the distribution of ages.

8. Merging, Joining, and Concatenating Data (Pandas)

Code Example:

```
# Merging two DataFrames

df_left = pd.DataFrame({'ID': [1, 2], 'Name': ['John', 'Anna']})

df_right = pd.DataFrame({'ID': [1, 2], 'Salary': [50000, 60000]})

merged_df = pd.merge(df_left, df_right, on='ID')
```

```
print("\nMerged DataFrame:\n", merged_df)
```

Explanation:

- Merging: pd.merge() combines the two DataFrames on the 'ID' column, similar to SQL joins.

9. Pivot Tables and Cross-tabulation (Pandas)

Code Example:

```
# Pivot Table in Pandas

pivot_table = pd.pivot_table(df_filled, values='Age', index='Name', aggfunc='mean')

print("\nPivot Table:\n", pivot_table)
```

Explanation:

- Pivot Table: The pivot table computes the average 'Age' for each 'Name'. Pivot tables are helpful for summarizing and reshaping data.

10. Time Series Analysis (Pandas Time Series Features)

Code Example:

```
# Creating a Time Series DataFrame

date_range = pd.date_range(start='2023-01-01', periods=5, freq='D')

ts_df = pd.DataFrame({'Date': date_range, 'Value': [10, 20, 30, 40, 50]})

ts_df.set_index('Date', inplace=True)


# Resampling the time series to weekly data

weekly_data = ts_df.resample('W').sum()

print("\nResampled Weekly Time Series Data:\n", weekly_data)
```

Explanation:

- Time Series: A time series DataFrame is created with daily data, and it is resampled into weekly data using .resample(). This is useful for summarizing time-based data.

11. Linear Algebra Functions (NumPy)

Code Example:

```
# Matrix multiplication

matrix1 = np.array([[1, 2], [3, 4]])

matrix2 = np.array([[5, 6], [7, 8]])

result = np.dot(matrix1, matrix2)

print("\nMatrix Multiplication Result:\n", result)
```

Explanation:

- Linear Algebra: np.dot() performs matrix multiplication, a key operation in linear algebra.

12. Statistical Functions (NumPy and Pandas)

Code Example:

```
# NumPy Standard Deviation

array = np.array([1, 2, 3, 4, 5])

std_dev = np.std(array)

print("NumPy Standard Deviation:", std_dev)



# Pandas Median

median_age = df_filled['Age'].median()

print("\nMedian Age in DataFrame:", median_age)
```

Explanation:

- Standard Deviation: np.std() computes the standard deviation in NumPy, a common measure of data dispersion.
- Pandas Median: median() calculates the middle value in the 'Age' column.

13. Visualization Integration (Pandas with Matplotlib/Seaborn)

Code Example:

```
import matplotlib.pyplot as plt


# Plotting with Pandas and Matplotlib

df_filled['Age'].plot(kind='bar', title="Age Distribution")

plt.show()
```

Explanation:

- Visualization: Pandas integrates with Matplotlib to generate plots directly from DataFrames, such as bar charts. This helps visualize data trends easily.

14. Performance Tuning (Efficient Operations in Pandas)

Code Example

:

```
# Apply a function across a DataFrame column

df_filled['Age_Squared'] = df_filled['Age'].apply(lambda x: x**2)

print("\nDataFrame with Age Squared:\n", df_filled)
```

Explanation:

- Performance: Using .apply() with a lambda function allows for efficient transformation of DataFrame columns.