1. Need and Overview of Pandas:

What is Pandas?

Pandas is a Python library for data manipulation and analysis. It provides data structures like **Series** (1D) and **DataFrame** (2D), making it easy to work with structured data.

Why is Pandas Needed?

- Efficiently handles large datasets.
- Simplifies data cleaning, transformation, and analysis.
- Integrates with libraries like NumPy and Matplotlib.
- Supports various file formats: CSV, Excel, JSON, SQL, etc.

2. Setup for Pandas:

Step 1: Install Pandas

Pandas can be installed using `pip`, Python's package manager.run:

```
pip install pandas
```

For Jupyter Notebook/ Google Colab users, install Pandas using the following command to ensure compatibility:

```
!pip install pandas
```

Step 2: Import Pandas

To use Pandas in your Python script or notebook, import it using the standard alias:

```
import pandas as pd
```

3. Pandas Data Structures: Series and DataFrame

Pandas provides two main data structures to handle and manipulate data efficiently: **Series** and **DataFrame**.

i. Series

A **Series** is a one-dimensional labeled array that can hold data of any type (e.g., integers, floats, strings). It is similar to a column in a spreadsheet or a Python list with an index.

Key Features

- **Indexing:** Each element has a unique label (index).
- **Homogeneous:** Holds data of a single type (e.g., all integers or all strings).

Code Example:

1. **Creating a Series**

```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data, index=['A', 'B', 'C', 'D'])
print(series)
```

```
A    10
B    20
C    30
D    40
dtype: int64
```

2. **Accessing Data in a Series**

```
# Accessing by index
print(series['A'])  # Output: 10

# Accessing by position
print(series[1])    # Output: 20
```

ii. DataFrame

- **DataFrame:** A 2D labeled data structure similar to a table.

Key Features

- **Labeled Rows and Columns:** Each row and column has a unique label (index and column names).
- **Heterogeneous:** Columns can hold data of different types.

Note : Difference Between DataFrames and 2D Arrays:

DataFrames have labeled rows and columns, whereas arrays rely solely on numerical indices.

Code Example:

1. **Creating a DataFrame**

```
# Creating a DataFrame
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 88]
}
df = pd.DataFrame(data)
print(df)
```

```
      Name  Age  Score
0    Alice   25     85
1      Bob   30     90
2  Charlie   35     88
```

2. **Accessing Data in a DataFrame**

```
# Access a column
print(df["Name"])  # Output: Series with names

# Access a row
print(df.loc[1])   # Output: Row with index 1

# Access a specific value
print(df.at[1, "Age"])  # Output: 30
```

Common File Formats for Datasets:

Format	Command	File Example
CSV	`pd.read_csv("data.csv")`	`data.csv`
Excel	`pd.read_excel("data.xlsx", sheet_name="Sheet1")`	`data.xlsx`
JSON	`pd.read_json("data.json")`	`data.json`
SQL	`pd.read_sql_query("SELECT * FROM table", conn)`	`database.db` (SQLite DB)
Parquet	`pd.read_parquet("data.parquet")`	`data.parquet`
Feather	`pd.read_feather("data.feather")`	`data.feather`

Note:
Parquet and **Feather** file formats, which are optimized for fast reading and writing of large datasets. These formats are commonly used in data engineering and analytics for efficient storage and processing.

Common Methods for Inspecting Data in Pandas:

Method	Description	Default	Example
`.head()`	Returns the first `n` rows of a DataFrame	5	`df.head(3)`
`.tail()`	Returns the last `n` rows of a DataFrame	5	`df.tail(3)`
`.sample()`	Returns a random sample of rows	1	`df.sample(2)`

These methods are particularly helpful for inspecting large datasets by viewing a small subset at the beginning, end, or randomly.

Details of DataFrames:

Labels (Columns, Index), Shape, Size, Info, and Describe:

Pandas provides several methods to quickly understand and summarize the structure and content of a **DataFrame**.

Method	Description	Example
`.columns`	Returns the column labels of the DataFrame	`df.columns`
`.index`	Returns the index labels (row labels) of the DataFrame	`df.index`
`.shape`	Returns the dimensions of the DataFrame (rows, columns)	`df.shape`
`.size`	Returns the total number of elements (rows × columns)	`df.size`
`.info()`	Provides a concise summary of the DataFrame (non-null counts, dtypes, memory usage)	`df.info()`
`.describe()`	Generates descriptive statistics (mean, count, etc.) for numerical columns	`df.describe()`

Accessing Data Using `.loc[]` and `.iloc[]`

In pandas, `.loc[]` and `.iloc[]` are powerful indexers used to access and manipulate data in a **DataFrame**.

1. **.loc[]**

   `.loc[]` is primarily label-based indexing. It is used to access rows and columns by their labels (names).

   - It can accept a **row label** and **column label** to return a specific value or subset of data.
   - You can use **boolean conditions** with `.loc[]` as well.

Code Example:

```python
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "Age": [25, 30, 35, 40, 45],
    "City": ["New York", "Los Angeles", "Chicago", "Houston", "Phoenix"]
})

# Access the value in the 3rd row and the 'Name' column
print(df.loc[2, 'Name'])  # Output: Charlie

# Access all rows for the 'City' column
print(df.loc[:, 'City'])

# Access the 2nd and 4th rows with 'Name' and 'Age' columns
print(df.loc[[1, 3], ['Name', 'Age']])
```

2. `.iloc[]`

`.iloc[]` is primarily **integer position-based** indexing. It is used to access rows and columns by their **integer index positions**.

- It works with **integer-based** indexing, so you can provide the position of the rows and columns.
- It does not include the last index (like Python's usual behavior with slicing).

Code Example:

```python
# Access the value in the 3rd row and 2nd column (zero-indexed)
print(df.iloc[2, 1])  # Output: 35

# Access the 1st and 3rd rows for the 'City' column (zero-indexed)
print(df.iloc[[0, 2], 2])

# Access the first 3 rows and first 2 columns
print(df.iloc[:3, :2])
```

Comparison

Method	Access Type	Indexing Type
`.loc[]`	Label-based indexing	Uses row/column labels
`.iloc[]`	Position-based indexing (integers)	Uses row/column integer positions

When to Use:

- `.loc[]` is useful when you need to access data by **names** (labels).
- `.iloc[]` is best when you need to access data by **integer position** (index numbers).

Accessing Single Values Using `.at[]` and `.iat[]`

`.at[]` is used to access a single value in a DataFrame **by label**.

Example:
```
df.at[row_label, column_label]
```

`.iat[]` is used to access a single value in a DataFrame **by integer position**.

Example:
```
df.iat[row_position, column_position]
```

Accessing Columns: Shorthand and Dot Notation

Shorthand Notation: Access a column in a DataFrame by **label** using square brackets.
Example:
```
df['column_name']
```

Dot Notation: Access a column in a DataFrame by label using dot notation.

Example:

```
df.column_name
```

Filtering Data Based on Conditions

You can filter data by applying conditions to one or more columns to return rows that meet the specified criteria.

Syntax:

```
df[condition]
```

condition: Boolean condition applied to one or more columns.

Example:

1. **Filter rows based on a single condition**:

   Condition: Age > 30)

   ```
   df[df['Age'] > 30]
   ```

2. **Filter rows based on multiple conditions (AND)**:

   Condition: Age > 30 and City is "Chicago"

   ```
   df[(df['Age'] > 30) & (df['City'] == 'Chicago')]
   ```

3. **Filter rows based on multiple conditions (OR)**:

   Condition: Age > 30 or City is "New York"

   ```
   df[(df['Age'] > 30) | (df['City'] == 'New York')]
   ```

4. **Filter rows using** `isin()` **for multiple values**:

   Condition: City is either "Chicago" or "Houston"

   ```
   df[df['City'].isin(['Chicago', 'Houston'])]
   ```

Note: You can apply conditions based on numerical comparisons, string matching, and more, using **&** (AND) and **|** (OR) for combining multiple conditions.

Regular Expressions (Regex) in Pandas

Regular expressions allow you to filter, match, and manipulate string data in pandas columns based on patterns.

Common Syntax:

1. **Filter rows containing a pattern**:

   Syntax:

   ```
   df[df['column_name'].str.contains('pattern', regex=True)]
   ```

2. **Filter rows not containing a pattern**:

   Syntax:

   ```
   df[~df['column_name'].str.contains('pattern', regex=True)]
   ```

3. **Filter rows starting with a specific pattern**:

   Syntax:

   ```
   df[df['column_name'].str.match('^pattern')]
   ```

4. **Replace values using regex**:

   Syntax:

   ```
   df['column_name'] = df['column_name'].str.replace('pattern',
   'replacement', regex=True)
   ```

General patterns which widely used with regex:

Part 1: Anchors and Basic Patterns

Pattern	Definition	Example	Matches
`^pattern`	Matches strings **starting with** the pattern.	`'^A'`	"Apple", "Apricot"
`pattern$`	Matches strings **ending with** the pattern.	`'\.com$'`	"example.com", "test.com"
`pattern1	pattern2`	Matches strings containing **either pattern1 or pattern2**.	`'A
`[abc]`	Matches any **one character** in the set.	`'[abc]'`	"apple", "banana", "carrot"
`[^abc]`	Matches any **character not** in the set.	`'[^abc]'`	"dog", "elephant"

Part 2: Special Characters and Quantifiers

Pattern	Definition	Example	Matches
`\d`	Matches any **digit** (0–9).	`'\d'`	"123", "42"
`\D`	Matches any **non-digit** character.	`'\D'`	"abc", "#"
`\w`	Matches any **alphanumeric** character or underscore.	`'\w'`	"cat", "dog_42"
`\W`	Matches any **non-alphanumeric** character.	`'\W'`	"@", "#"
`\s`	Matches any **whitespace** character.	`'\s'`	" " (space), "\t" (tab)
`\S`	Matches any **non-whitespace** character.	`'\S'`	"Hello", "World"

Part 3: Quantifiers and Escapes

Pattern	Definition	Example	Matches
`{n}`	Matches **exactly n occurrences** of the preceding item.	`'a{3}'`	"aaa"
`{n,}`	Matches **n or more occurrences** of the preceding item.	`'a{2,}'`	"aa", "aaa", "aaaa"
`{n,m}`	Matches **between n and m occurrences**.	`'a{2,4}'`	"aa", "aaa", "aaaa"
`.`	Matches any **single character** except newline.	`'.'`	"a", "1", "@"
`\b`	Matches a **word boundary**.	`'\bcat\b'`	"cat" (not "category")
`.*`	Matches **zero or more of any character**.	`'.*'`	Any text
`\\`	Escapes a special character to treat it literally.	`'\\.'`	"." (literal dot)

Transforming Data Using `apply()`

The `apply()` method in pandas is used to apply a **custom function** or a predefined operation along the rows (`axis=1`) or columns (`axis=0`) of a **DataFrame** or on a **Series**.

Syntax:

For Series: `Series.apply(func)`

For DataFrame: `DataFrame.apply(func, axis=0/1)`

Example 1: Applying a Function to a Series

```
import pandas as pd

data = pd.Series([1, 2, 3, 4, 5])
result = data.apply(lambda x: x**2)  # Square each value
print(result)
```

```
0     1
1     4
2     9
3    16
4    25
dtype: int64
```

Example 2: Applying a Function to a Series

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
result = df.apply(lambda x: x.sum(), axis=0)  # Sum each column
print(result)
```

```
A     6
B    15
dtype: int64
```

Example 3: Applying a Function Along DataFrame Rows

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
result = df.apply(lambda x: x.sum(), axis=1)  # Sum each row
print(result)
```

```
0    5
1    7
2    9
dtype: int64
```

Transforming or Adding Data Using `where()`

The `where()` method in pandas is used to conditionally transform data. It retains values that meet a given condition and replaces others with a specified value (default is `NaN`).

Syntax:

For Series: `Series.where(cond, other=np.nan) # np -> numpy alias`

For DataFrame: `DataFrame.where(cond, other=np.nan, axis=0)`

Example:

```python
data = pd.Series([10, 20, 30, 40, 50])

df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [10, 20, 30, 40]
})
```

Let's use `where()` on the above data and dataFrame

Example	Code	Condition	Output
1. Series Filtering	`data.where(data > 30)`	Keep values greater than 30	`[NaN, NaN, NaN, 40.0, 50.0]`
2. DataFrame Filtering	`df.where(df > 15)`	Keep values greater than 15	`[[NaN, NaN], [NaN, NaN], [NaN, 30.0], [NaN, 40.0]]`
3. Replace Values	`df.where(df > 15, other=0)`	Replace values ≤ 15 with `0`	`[[0, 0], [0, 0], [0, 30], [0, 40]]`
4. Adding Data	`df['C'] = df['A'].where(df['A'] % 2 == 0, other='Odd')`	Mark even numbers, else 'Odd'	`A: [1, 2, 3, 4]`, `B: [10, 20, 30, 40]`, `C: ['Odd', 2, 'Odd', 4]`

Inserting Columns:

Syntax : `df.insert(position, new_column_name, column_data)`

Example: `df.insert(1, 'Gender', ['F', 'M'])`

Dropping Columns

Syntax : `df.drop(column_name, axis=1, inplace=True)`

Example: `df.drop('Gender', axis=1, inplace=True)`

Renaming Columns

Syntax :

```
df.rename(columns={'old_column_name': 'new_column_name'},
inplace=True)
```

Example:

```
df.rename(columns={'name': 'FullName'}, inplace=True))
```

Merging DataFrames: Inner, Outer, Left, Right Joins

Merging combines two DataFrames using a common key (or keys). Joins control how the DataFrames are merged based on the relationship of their keys.

Join Types and Syntax

Join Type	Definition	Syntax
Inner	Keeps only the rows with matching keys in both DataFrames.	`pd.merge(df1, df2, how='inner', on='key')`
Outer	Keeps all rows from both DataFrames, filling missing values with `NaN` for non-matching keys.	`pd.merge(df1, df2, how='outer', on='key')`
Left	Keeps all rows from the left DataFrame, adding matching rows from the right DataFrame.	`pd.merge(df1, df2, how='left', on='key')`
Right	Keeps all rows from the right DataFrame, adding matching rows from the left DataFrame.	`pd.merge(df1, df2, how='right', on='key')`

Code Examples:

```python
import pandas as pd

# Example DataFrames
df1 = pd.DataFrame({
    'key': [1, 2, 3],
    'value1': ['A', 'B', 'C']
})

df2 = pd.DataFrame({
    'key': [2, 3, 4],
    'value2': ['X', 'Y', 'Z']
})
```

1. **Inner Join**

```python
#Inner Join
result = pd.merge(df1, df2, how='inner', on='key')
print(result)
```

```
   key value1 value2
0    2      B      X
1    3      C      Y
```

2. **Outer Join**

```python
#Outer Join
result = pd.merge(df1, df2, how='outer', on='key')
print(result)
```

```
   key value1 value2
0    1      A    NaN
1    2      B      X
2    3      C      Y
3    4    NaN      Z
```

3. **Left Join**

```
#Left Join
result = pd.merge(df1, df2, how='left', on='key')
print(result)
```

```
   key value1 value2
0    1      A    NaN
1    2      B      X
2    3      C      Y
```

4. Right Join

```
#Right Join
result = pd.merge(df1, df2, how='right', on='key')
print(result)
```

```
   key value1 value2
0    2      B      X
1    3      C      Y
2    4    NaN      Z
```

Concatenating DataFrames

Concatenation in pandas refers to combining two or more DataFrames along a particular axis (either rows or columns). The `concat()` function is used to join DataFrames either vertically (stacking rows) or horizontally (joining columns).

Syntax:

```
pd.concat([df1, df2, ...], axis=0, join='outer', ignore_index=False)
```

Note:

`axis`: Determines whether to concatenate along rows (`axis=0`, default) or columns (`axis=1`).
`join`: Specifies how to handle columns that are not present in both DataFrames:

- **`'outer'`** (default): Includes all columns (union of columns).
- **`'inner'`**: Includes only columns common to all DataFrames.

`ignore_index`: If `True`, the index is reset. If `False`, keeps the original index from each DataFrame.

Code Example

1. **Concatenate Vertically (Stacking Rows)**

```
import pandas as pd

# Example DataFrames
df1 = pd.DataFrame({
    'key': [1, 2],
    'value': ['A', 'B']
})

df2 = pd.DataFrame({
    'key': [3, 4],
    'value': ['C', 'D']
})

# Concatenate vertically (stack rows)
result = pd.concat([df1, df2], axis=0, ignore_index=True)
print(result)
```

```
   key value
0    1     A
1    2     B
2    3     C
3    4     D
```

2. **Concatenate Horizontally (Joining Columns)**

```
# Concatenate horizontally (join columns)
df3 = pd.DataFrame({
    'extra': ['X', 'Y', 'Z', 'W']
})

result = pd.concat([df1, df2, df3], axis=1)
print(result)
```

```
   key value  key value extra
0  1.0     A  3.0     C     X
1  2.0     B  4.0     D     Y
2  NaN   NaN  NaN   NaN     Z
3  NaN   NaN  NaN   NaN     W
```

Handling Null (Missing) Values in Pandas

Null values are represented as `NaN` in pandas. Handling them efficiently is essential for data cleaning and preparation. Pandas provides several methods to detect, fill, or drop missing data.

Methods for Handling Null Values

Method	Description	Syntax
isnull()	Detects missing values and returns a boolean DataFrame indicating `True` for `NaN`.	`df.isnull()`
notnull()	Returns the opposite of `isnull()`, indicating `True` for non-null values.	`df.notnull()`
dropna()	Drops rows or columns with missing values.	`df.dropna(axis=0, how='any', inplace=False)`
fillna()	Fills missing values with a specified value or method.	`df.fillna(value=None, method=None, inplace=False)`
replace()	Replaces specific values (including `NaN`) with another value.	`df.replace(to_replace=None, value=None, inplace=False)`
interpolate()	Fills missing values using interpolation methods.	`df.interpolate(method='linear', inplace=False)`

Grouping Data Using `groupby()`

`groupby()` in pandas is a powerful tool for grouping data based on one or more columns, followed by applying aggregation or transformation operations to each group. It is commonly used for summarizing, aggregating, and transforming data.

Syntax:

```
df.groupby(by, axis=0, level=None, as_index=True, sort=True,
group_keys=True)
```

by: Column(s) or index level(s) to group by.
`axis`: Axis to group along (default is 0 for rows).
`level`: Group by a particular level (useful for MultiIndex).
`as_index`: If `True` (default), the group labels become the index.

sort: If `True` (default), the groups are sorted.

group_keys: If `True` (default), it includes group keys in the result.

Common Operations with `groupby()`

1. **Aggregation** (e.g., sum, mean)
2. **Transformation** (e.g., normalization, filling missing values)
3. **Iteration** (e.g., iterating over groups)

Code Example:

1. **Grouping and Aggregating with** `sum()`

```
import pandas as pd

# Example DataFrame
df = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [10, 20, 30, 40, 50]
})

# Group by 'Category' and calculate the sum of 'Value'
grouped_sum = df.groupby('Category')['Value'].sum()
print(grouped_sum)
```

```
Category
A    90
B    60
Name: Value, dtype: int64
```

2. **Grouping and Aggregating with multiple functions**

```
# Group by 'Category' and apply multiple aggregation functions
grouped_agg = df.groupby('Category')['Value'].agg(['sum', 'mean', 'max'])
print(grouped_agg)
```

```
          sum  mean  max
Category
A          90  30.0   50
B          60  30.0   40
```

3. **Grouping and Iterating Over Groups**

```python
# Iterate over groups
for name, group in df.groupby('Category'):
    print(f"Group: {name}")
    print(group)
```

```
Group: A
  Category  Value
0        A     10
2        A     30
4        A     50
Group: B
  Category  Value
1        B     20
3        B     40
```

4. **Grouping by Multiple Columns**

```python
# Group by 'Category' and 'Value' and calculate the sum
df['Value'] = [10, 20, 30, 40, 50]
grouped_multiple = df.groupby(['Category', 'Value']).size()
print(grouped_multiple)
```

```
Category  Value
A         10       1
          30       1
          50       1
B         20       1
          40       1
dtype: int64
```

5. **Transforming Data Within Groups Using** `transform()`

```python
# Normalize data by subtracting group mean
df['Normalized'] = df.groupby('Category')['Value'].transform(lambda x: x - x.mean())
print(df)
```

```
  Category  Value  Normalized
0        A     10       -20.0
1        B     20       -10.0
2        A     30         0.0
3        B     40        10.0
4        A     50        20.0
```