1. Introduction to Pandas

Definition: **Pandas** is a powerful data analysis and manipulation library in Python, built on top of **NumPy**. It provides data structures like **Series** and **DataFrames** to efficiently handle and analyze large datasets.

Functionality: Pandas allows:

- Handling missing data.
- Data reshaping and aggregation.
- Merge and join operations.
- Time series functionality.
- Grouping, sorting, and filtering data.
- Easy reading/writing of data to/from different file formats (CSV, Excel, SQL, etc.).

2. Fundamentals of Pandas

Definition: Pandas' core functionality revolves around two main data structures:

- **Series**: A one-dimensional array-like object with labeled axes (indexes).
- **DataFrame**: A two-dimensional table where data is arranged in rows and columns with labels for both axes.

Functionality:

- **Series** allows you to work with a single list of data (e.g., temperatures, ages).
- **DataFrame** handles structured data (tables) with multiple variables (e.g., columns with names and types).

---

3. Importing Pandas

Definition: To use **Pandas** in Python, you need to import it using import pandas as pd.

Functionality: The import statement allows access to Pandas functions and methods, enabling data manipulation and analysis.

---

4. Aliasing

Definition: **Aliasing** is giving a library or module a shorthand name for easier access.

Functionality: Common alias for **Pandas** is pd to reduce typing. For example:

- import pandas as pd is the standard alias for Pandas.
- Similarly, import numpy as np is used for **NumPy**.

SERIES – INTRODUCTION

Definition: A **Series** is a one-dimensional array-like object in Pandas with labeled axes (index). It is similar to a list or a dictionary but with labeled indices.

Functionality: A Series can hold any data type and is often used to represent a single column in a DataFrame. The key feature is that each data element has an index (label).

Creating Series Object

Definition: A **Series object** can be created from various data structures such as lists, dictionaries, or arrays.

Functionality: This allows you to create a labeled dataset with a custom index, making it easy to perform operations like indexing, slicing, and mathematical operations.

Empty Series Object

Definition: An **empty Series** is a Series with no data but with an index structure.

Functionality: It can be useful when initializing a Series that will later be filled with data or as a placeholder.

Create Series from List/Array/Column from DataFrame

Definition: A Series can be created from various sources such as lists, NumPy arrays, or columns from a **DataFrame**.

Functionality: This allows easy conversion between different data structures. For instance, if a DataFrame contains multiple columns, you can extract one column as a Series for further analysis.

Index in Series

Definition: The **index** in a Series is an array of labels that correspond to the data. It works like the keys in a dictionary.

Functionality: It allows for efficient data retrieval by label (index) and supports operations like alignment when combining Series.

Accessing values in Series

Definition: Values in a Series can be accessed by their **index**.

Functionality: You can retrieve data using integer-based indexing or label-based indexing, enabling both positional and label-based access.

NaN Value

Definition: **NaN** (Not a Number) represents missing or undefined values in a dataset.

Functionality: In Pandas, **NaN** is used to denote missing data and is handled using methods like **isna()** or **fillna()** to check or fill missing values.

1. **df.isna()**: Check for missing (NaN) values.
2. **df.fillna()**: Replace NaN with a specified value.
3. **df.dropna()**: Remove rows with NaN values.

Series – Attributes (Values, index, dtypes, size)

Definition: Attributes of a Series include:

- values: The actual data of the Series.
- index: The labels corresponding to the data.
- dtype: The data type of the Series.
- size: The number of elements in the Series.

Functionality: These attributes provide essential metadata that helps understand the structure and characteristics of the Series.

Series – Methods (head(), tail(), sum(), count(), nunique() etc.)

Definition: Methods in Series allow performing various operations:

- head(): Returns the first few elements.
- tail(): Returns the last few elements.
- sum(): Sums all elements.
- count(): Counts non-null values.
- nunique(): Returns the number of unique values.

Functionality: These methods are useful for inspecting and performing basic statistical operations on Series.

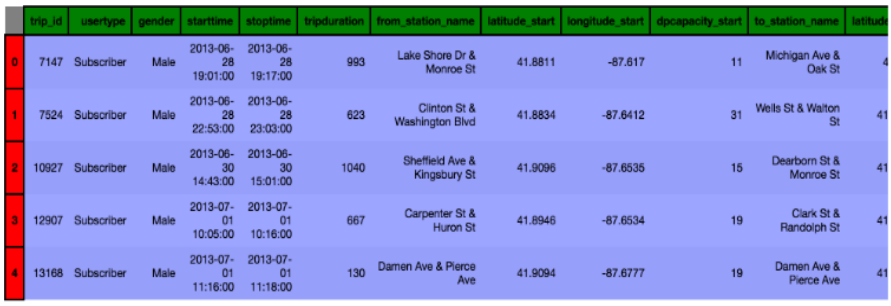
DataFrame

Components of a DataFrame

The Columns, Index, and Data

Columns

Index



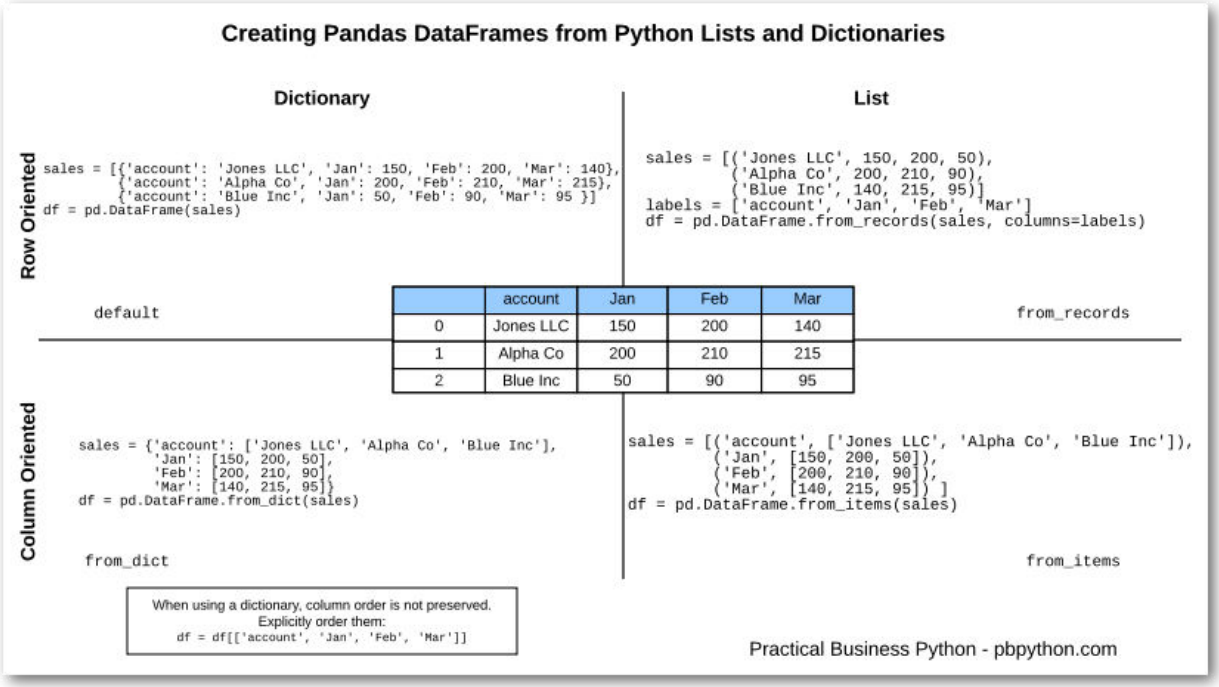
	trip_id	usertype	gender	starttime	stoptime	tripduration	from_station_name	latitude_start	longitude_start	dpcapacity_start	to_station_name	latitud
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	41.8811	-87.617	11	Michigan Ave & Oak St	4
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	41.8834	-87.6412	31	Wells St & Walton St	41
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	41.9096	-87.6535	15	Dearborn St & Monroe St	41
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	41.8946	-87.6534	19	Clark St & Randolph St	41
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130	Damen Ave & Pierce Ave	41.9094	-87.6777	19	Damen Ave & Pierce Ave	41

Data

Description	Alternative Names	Axis Number
• Columns - label each column • Index - label each row • Data - actual values in DataFrame	• Columns - column names/labels, column index • Index - index names/labels, row names/labels • Data - values	• Columns: 1 • Index: 0

Definition: A **DataFrame** is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

Functionality: DataFrames are the most commonly used Pandas data structure and allow complex data manipulation, filtering, grouping, merging, etc.



Loading Different Files

Definition: Pandas can read data from multiple file formats like **CSV**, **Excel**, **SQL**, **JSON**, etc., and convert them into DataFrames.

1. **CSV**: pd.read_csv()
2. **Excel**: pd.read_excel()
3. **SQL**: pd.read_sql()
4. **JSON**: pd.read_json()
5. **HDF5**: pd.read_hdf()

Functionality: This enables data import from various sources for analysis, and it supports writing data back to these formats for saving results.

DataFrame Attributes

Definition: Key attributes of DataFrame include:

- shape: The number of rows and columns.
- columns: The names of columns.
- index: The row labels.
- dtypes: Data types of the columns.

Functionality: These attributes give insight into the structure of the DataFrame, allowing for better data management and inspection.

DataFrame Methods

Definition: Methods in DataFrames are functions used to manipulate and analyze data:

1. **head()**: First 5 rows of the DataFrame.
2. **tail()**: Last 5 rows of the DataFrame.
3. **info()**: Summary of the DataFrame including number of non-null values and data types.
4. **describe()**: Statistical summary of the DataFrame (mean, std, min, etc.).

Functionality: These methods provide quick summaries, allow data inspection, and assist in understanding the structure and content of the DataFrame.

Rename Column & Index

Definition: The **rename()** method allows changing the column names or row indices.

Functionality: This is useful when you need to adjust the labels for clarity or consistency.

Inplace Parameter

Definition: The inplace=True parameter modifies the DataFrame directly without returning a new object.

Functionality: This allows for more efficient operations, especially when dealing with large datasets, as it modifies the data in place without creating additional copies.

Handling Missing or NaN values

Definition: Missing data (NaN) can be handled by either filling it with a default value or dropping rows/columns containing NaN values.

Functionality: Methods like **fillna()** and **dropna()** are used to address missing data by filling it with specific values or removing it entirely.

---

iLoc and Loc

Definition:

- iloc[]: Integer-location based indexing for selecting rows and columns by position.
- loc[]: Label-based indexing for selecting data by row/column labels.

Functionality: These indexing methods enable access to specific rows and columns in a DataFrame.

DataFrame – Filtering

Definition: Filtering involves selecting specific rows based on conditions (e.g., df[df['column'] > 50]).

Functionality: This allows for subsetting the data and focusing on rows of interest.

DataFrame – Sorting

Definition: Sorting refers to arranging rows based on column values or index.

Functionality: Methods like **sort_values()** and **sort_index()** are used for organizing the data in ascending or descending order.

DataFrame – GroupBy

Definition: Grouping involves splitting data into groups based on some criteria and then applying a function (like sum, mean, etc.) to each group.

Functionality: This is useful for aggregation, summarization, or applying specific operations to subsets of data.

Merging or Joining

Definition: Merging or joining involves combining multiple DataFrames based on a common column or index.

Functionality: The **merge()** method is used to perform database-style joins (inner, outer, left, right), combining data from different sources.

DataFrame – Concat

Definition: **Concatenation** combines multiple DataFrames along a particular axis (rows or columns).

Functionality: The **concat()** method is used to append or stack DataFrames vertically or horizontally.

DataFrame - Adding, Dropping Columns & Rows

Definition: This involves adding or removing columns or rows in a DataFrame.

Functionality: Methods like **drop()** and direct assignment (df['new_col'] = values) are used for adding or dropping elements.

DataFrame - Date and Time

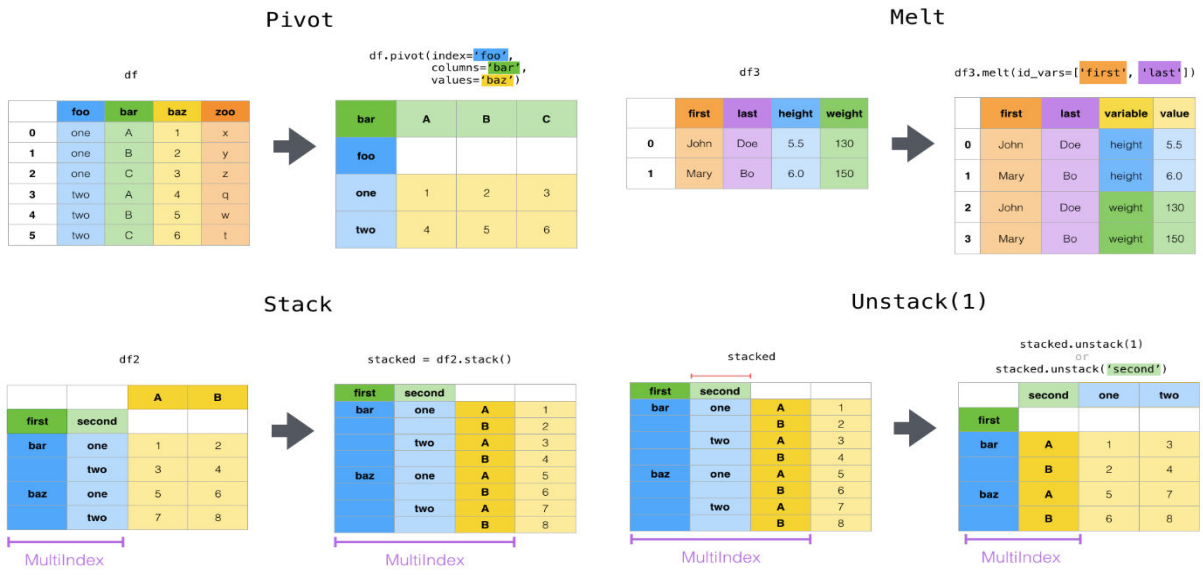
Definition: Pandas provides powerful functionality for working with **date and time** data.

Functionality: Methods like **pd.to_datetime()** allow conversion of date strings into datetime objects. Operations like extracting year, month, and day are also supported.

DataFrame - Concatenate Multiple CSV Files

Definition: Concatenating multiple CSV files refers to combining data from several CSV files into one DataFrame.

Functionality: This is commonly done using **concat()**, making it easy to work with datasets split across multiple files.



Pandas Interview Questions

1. What is Pandas in Python?

Pandas is an open-source Python package that is most commonly used for data science, data analysis, and machine learning tasks. It is built on top of another library named **Numpy**. It provides various data structures and operations for manipulating numerical data and time series and is very efficient in performing various functions like data visualization, data manipulation, data analysis, etc.

2. Mention the different types of Data Structures in Pandas?

Pandas have three different types of data structures. It is due to these simple and flexible data structures that it is fast and efficient.

- **Series** - It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as **integers**, **floats**, and **strings** and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed.
- **DataFrame** - It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner. Both size and values of DataFrame are mutable.
- **Panel** - The Pandas have a third type of data structure known as Panel, which is a 3D data structure capable of storing heterogeneous data but it isn't that widely used.

3. What are the significant features of the pandas Library?

Pandas library is known for its efficient data analysis and state-of-the-art data visualization.
The key features of the panda's library are as follows:

- Fast and efficient DataFrame object with default and customized indexing.
- High-performance merging and joining of data.
- Data alignment and integrated handling of missing data.
- Label-based slicing, indexing, and subsetting of large data sets.
- Reshaping and pivoting of data sets.
- Tools for loading data into in-memory data objects from different file formats.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- Time Series functionality.

4. Define Series in Pandas?

It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as **integers**, **floats**, and **strings** and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed. By using a 'series' method, we can easily convert the list, tuple, and dictionary into a series. A Series cannot contain multiple columns.

5. Define DataFrame in Pandas?

It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner i.e. in rows and columns and the indexes with respect

to these are called row index and column index respectively. Both size and values of DataFrame are mutable. The columns can be heterogeneous types like int and bool. It can also be defined as a dictionary of Series.

6. What are the different ways in which a series can be created?

i. **From a List** – Passing a Python list to pd.Series().
ii. **From a NumPy Array** – Using a NumPy array as input.
iii. **From a Dictionary** – Creating a Series with dictionary keys as indices and values as data.
iv. **From a Scalar Value** – Creating a Series with the same value repeated for a given index.
v. **From Another Series** – Copying an existing Series.
vi. **From a DataFrame Column** – Extracting a single column from a DataFrame.

7. What are the different ways in which a dataframe can be created?

i. **From a Dictionary of Lists** – Using a dictionary where keys are column names and values are lists.
ii. **From a Dictionary of Series** – Using Pandas Series as column values.
iii. **From a Dictionary of Dictionaries** – Keys as column names and nested dictionaries as row values.
iv. **From a List of Dictionaries** – Each dictionary represents a row with column names as keys.
v. **From a NumPy Array** – Converting a structured NumPy array into a DataFrame.
vi. **From Another DataFrame** – Copying or modifying an existing DataFrame.
vii. **From an External File** – Reading data from CSV, Excel, SQL, JSON, etc.

8. How can we create a copy of the series in Pandas?

In Pandas, a copy of a **Series** can be created using the `.copy()` method:

```python
new_series = original_series.copy()
```

9. Explain Categorical data in Pandas?

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

10. Explain Reindexing in pandas along with its parameters?

Reindexing as the name suggests is used to alter the rows and columns in a DataFrame. It is also defined as the process of conforming a dataframe to a new index with optional filling logic. For missing values in a dataframe, the reindex() method assigns NA/NaN as the value. A new object is returned unless a new index is produced that is equivalent to the current one. The copy value is set to **False**. This is also used for changing the index of rows and columns in the dataframe

11. How can we convert Series to DataFrame?

```python
import pandas as pd
series = pd.Series([1, 2, 3, 4])
df = series.to_frame(name="Column1")
```

12. How can we convert DataFrame to Numpy Array?

```python
import pandas as pd
import numpy as np

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

# Convert to NumPy array
numpy_array = df.to_numpy()
```

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

In order to convert DataFrame to an excel file we need to use the to_excel() function. There are various parameters to be considered. But initially, all you need is to mention the DataFrame name and the name of the excel sheet.

14. List some statistical functions in Python Pandas?

Some of the major statistical functions in Python Pandas are:

- **sum()** – It returns the sum of the values.
- **min()** – It returns the minimum value.
- **max()** – It returns the maximum value.
- **abs()** – It returns the absolute value.
- **mean()** – It returns the mean which is the average of the values.
- **std()** – It returns the standard deviation of the numerical columns.
- **prod()** – It returns the product of the values.

15. How to Read Text Files with Pandas?

There are multiple ways in which we read a text file using Pandas.

- **Using read_csv():** CSV is a comma-separated file i.e. any text file that uses commas as a delimiter to separate the record values for each field. Therefore, in order to load data from a text file we use pandas.read_csv() method.
- **Using read_table():** This function is very much like the read_csv() function, the major difference being that in read_table the delimiter value is '\t' and not a comma which is the default value for read_csv(). We will read data with the read_table function making the separator equal to a single space(' ').
- **Using read_fwf():** It stands for fixed-width lines. This function is used to load DataFrames from files. Another very interesting feature is that it supports optionally iterating or breaking the file into chunks. Since the columns in the text file were separated with a fixed width, this read_fwf() read the contents effectively into separate columns.

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

- **DataFrame.iloc():** It is a method used to retrieve data from a Data frame, and it is an integer position-based locator (from 0 to length-1 of the axis), but may also be used with a boolean array and this is the major difference factor between iloc() and loc(). It takes input as **integers**, arrays of integers, an object, boolean arrays, and functions.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
```

```
df = pd.DataFrame(student_dict)
print(df)
print(df.iloc[[0, 2]])
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
2  Sheila   12    91
```

- **DataFrame.loc():** It gets rows or columns with particular labels as input. It takes input as a single label, a list of arrays, and objects with labels. It does not work with boolean arrays or values.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

print(df.loc[(df.Name=='Kate')])
```

Output:-

```
    Name  Age  Marks
0   Kate   10    85
1  Harry   14    77
2  Sheila   12    91
   Name  Age  Marks
```

```
0  Kate   10    85
```

17. How will you sort a DataFrame?

The function used for sorting in pandas is called DataFrame.sort_values(). It is used to sort a DataFrame by its column or row values. The function comes with a lot of parameters, but the most important ones to consider for sort are:

- **by:** It is used to specify the column/row(s) which are used to determine the sorted order. It is an optional parameter.
- **axis:** It specifies whether the sorting is to be performed for a row or column and the value is 0 and 1 respectively.
- **ascending:** It specifies whether to sort the dataframe in ascending or descending order. The default value is set to ascending. If the value is set as **ascending=False** it will sort in descending order.

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

Depending on the problem, continuous values can be discretized using the cut() or qcut() function:

- **cut()** It bins the data based on values. We use it when we need to segment and sort data values into bins that are evenly spaced. cut() will choose the bins to be evenly spaced based on the values themselves and not the frequency of those values. For example, cut could convert ages to groups of age ranges.
- **qcut()** bins the data based on sample quantiles. We use it when we want to have the same number of records in each bin or simply study the data by quantiles. For example, if in a data we have 30 records, and we want to compute the quintiles, qcut() will divide the data such that we have 6 records in each bin.

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

Both join and merge functions are used to combine two dataframes. The major difference is that the join method combines two dataframes on the basis of their indexes whereas the merge method is more flexible and allows us to specify columns along with the index to combine the two dataframes.

These are the main differences between df.join() and df.merge():

- **lookup on right table:** When performing a lookup on the right table, the join() method will always use the index of df2 to perform the join operation. However, if you use the merge() method, you can choose to join based on one or more columns of df2 by default, or even the index of df2 if you specify the right_index=True parameter.
- **lookup on left table:** When performing a lookup on the left table, df1.join(df2) method will use the index of df1 by default, while df1.merge(df2) method will use the column(s) of df1 for the join operation. However, you can override this behavior by specifying the on=key_or_keys parameter in df1.join(df2) or by setting the left_index=True parameter in df1.merge(df2).
- **left vs inner join:** By default, the df1.join(df2) method performs a left join (retains all rows of df1), while the df1.merge(df2) method performs an inner join (returns only the matching rows of df1 and df2).

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

Both concat and merge functions are used to combine dataframes. There are three major key differences between these two functions.

- **The Way of Combining: concat()** function concatenates dataframes along rows or columns. It is nothing but stacking up of multiple dataframes whereas merge() combines dataframes based on values in shared columns thus it is more flexible compared to concat() as the combination can happen based on the given condition.
- **Axis parameter:** concat() function has axis parameter. Since merge() function combines dataframes on the basis of shared columns side by side it does not really need an axis parameter. The value of the axis parameter decides in what direction will the concatenation happen. For it to happen row-wise the value of the axis parameter will be '0' and for it to happen side-by-side it will be '1'. The default value is 1.
- **Join vs How:** Join is a parameter of concat() function and how is a parameter of merge() function. Join can take two values outer and inner whereas how can take four values inner, outer, left, and right.

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

- **fillna():** It fills the NaN values with a given number with which you want to substitute. It gives you the option to fill according to the index of rows of a pd.DataFrame or on the name of the columns in the form of a python dict.

Pandas Coding Interview Questions

1. How to reset the index in a Python Pandas DataFrame?

Inorder to reset the index of the DataFrame we use the Dataframe.reset_index() command. If the DataFrame has a MultiIndex, this method can also remove one or more levels.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)

student_df.reset_index(drop=True, inplace=True)
print(student_df)
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
       Age  Marks
Name
```

```
Kate    10    85
Harry   11    77
Sheila  12    91
   Age  Marks
0   10    85
1   11    77
2   12    91
```

2. How to rename the index in a Pandas DataFrame?

In order to rename a DataFrame we use
the Dataframe.set_index() method to give different values to the columns or the index values of DataFrame. Like in this example we will change the index label from ‘Name’ to ‘FirstName’.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)
student_df.index.names = ['FirstName']
print(student_df)
```

Output:-

```
      Age  Marks
```

```
Name
Kate     10     85
Harry    11     77
Sheila   12     91
          Age  Marks
FirstName
Kate       10     85
Harry      11     77
Sheila     12     91
```

3. How to get frequency count of unique items in a Pandas DataFrame?

In order to get the frequency count of unique items in a Pandas DataFrame we can use the Series.value_counts() method.

Code Example :

```
# importing the module
import pandas as pd

# creating the series
s = pd.Series(data = [1,2,3,4,3,5,3,7,1])

# displaying the series
print(s)

# finding the unique count
print(s.value_counts())
```

Output:-

```
0   1
1   2
```

```
2    3
3    4
4    3
5    5
6    3
7    7
8    1
dtype: int64
3    3
1    2
2    1
4    1
5    1
7    1
dtype: int64
```

4. How to delete a column in Pandas DataFrame?

The drop() method is used to delete a column in a DataFrame. If we set the value of the axis parameter as '1' it will work for a column if we set the value to '0' it will delete the rows in the DataFrame.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
student_df.drop(["Age"], axis = 1, inplace = True)
print(student_df)
```

Output:-

```
    Name  Marks
0    Kate     85
1   Harry     77
2  Sheila     91
```

5. How to delete a row in Pandas DataFrame?

The drop() method is used to delete a row in a DataFrame. If we set the value of the axis parameter as ‘0’ or do not mention it at all it will work for rows as the default value for the axis parameter is set to ‘0’, if we set the value to ‘1’ it will delete the column in the DataFrame.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)


# set index using column
student_df = student_df.set_index('Name')
print(student_df)

student_df.drop(["Harry"], inplace = True)
print(student_df)
```

Output:-

```
      Age  Marks
Name
Kate    10    85
Harry   11    77
Sheila  12    91
      Age  Marks
Name
Kate    10    85
Sheila  12    91
```

6. How can we convert NumPy array into a DataFrame?

In order to convert a Numpy array into a DataFrame we first need to create a numpy array and then use the pandas.DataFrame the method along with specifying the/labels for rows and columns.

Code Example :

```
# Python program to Create a
# Pandas DataFrame from a Numpy
# array and specify the index
# column and column headers

# import required libraries
importnumpyas np
import pandas as pd

# creating a numpy array
numpyArray = np.array([[115, 222, 343],
                       [323, 242, 356]])

# generating the Pandas dataframe
# from the Numpy array and specifying
```

```
# name of index and columns
dataframe = pd.DataFrame(data = numpyArray,
                                        index = ["Row1", "Row2"],
                                        columns = ["Column1",
                                                   "Column2",
"Column3"])

# printing the dataframe
print(dataframe)
```

Output:-

```
    Column1  Column2  Column3
Row1     115     222     343
Row2     323     242     356
```

7. How do you split a DataFrame according to a boolean criterion?

We can create a mask to separate the dataframe and then use the inverse operator (~) to take the complement of the mask.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
df1 = df[df['Age'] >10]

# printing df1
```

```
df1
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   14    77
2 Sheila   12    91
Name Age    Marks
1       Harry 14      77
2       Sheila 12      91
```

8. How to create Timedelta objects in Pandas?

String: In order to create a timedelta object using a string argument we pass a string literal.

Code Example :

```
#importing necessary libraries
import pandas as pd
importnumpyas np

#Conversion from string format to date format takes place using Timedelta method.
print (pd.Timedelta('20 days 12 hours 45 minutes 3 seconds'))
```

Output:

```
20 days 12:45:03
```

**Integer*: What differs from string, in this case, is we just need to pass an integer value and the object will be created.

Code Example :

```
#importing necessary libraries
import pandas as pd
importnumpyas np

print (pd.Timedelta(16,unit='h'))#h here is used for hours.
```

Output:

```
0 days 16:00:00
```

Data Offsets: In order to first learn how to create a timedelta object using data offset we first need to understand what data offset actually is. Data offsets are parameters like weeks, days, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds. This when passed as an argument helps in the creation of the timedelta object.

Code Example :

```
#importing necessary libraries
import pandas as pd
importnumpyas np

print (pd.Timedelta(days=2, hours = 16))
```

Output:-

```
2 days 16:00:00
```

Code Example :

```
#importing necessary libraries
import pandas as pd
importnumpyas np

print (pd.Timedelta(days=2, hours = 6, minutes = 23))
```

Output:-

```
2 days 06:23:00
```

9. How will you compute the percentile of a numerical series in Pandas?

In order to compute percentile we use numpy.percentile() method.

Syntax:

```
numpy.percentile(a, q, axis=None, out=None, overwrite_input=False, method='linear', keepdims=False, *, interpolation=None)
```

It will calculate the q-th percentile of the given the data along the mentioned axis.

Parameters:

- **a:** It is an input array or object that can be converted to an array.
- **q:** It is the percentile or sequence of percentiles to be calculated. The value must be between 0 and 100 both inclusive.

Code Example :

```
import pandas as pd
import random

A = [ random.randint(0,100) foriinrange(10) ]
B = [ random.randint(0,100) foriinrange(10) ]

df = pd.DataFrame({ 'field_A': A, 'field_B': B })
df
```

```
print(df.field_A.quantile(0.1)) # 10th percentile


print(df.field_A.quantile(0.5)) # same as median

print(df.field_A.quantile(0.9)) # 90th percentile
```

Output:-

```
12.1
52.0
92.6
```

10. How can we convert DataFrame into a NumPy array?

In order to convert a dataframe into a NumPy array we use DataFrame.to_numpy() method.

Syntax:

```
DataFrame.to_numpy(dtype=None, copy=False,
na_value=_NoDefault.no_default)
```

Parameters:

- **dtype:** It accepts string or numpy.dtype value. It is an optional parameter.
- **copy:** It accepts a boolean value. The default value is set to False. It ensures that the returned value is not a view on another array. Setting the value of copy=False does not ensure that to_numpy() is no-copy. Whereas if copy=True it does ensure that a copy is made.

- **na_value:** It accepts the parameter of any datatype and it is an optional parameter. It specifies the value to be used for missing values. The default value is of the same data type as the object.

Code Example :

```
import pandas as pd

# initialize a dataframe
df = pd.DataFrame(
        [[10, 12, 33],
        [41, 53, 66],
        [17, 81, 19],
        [10, 11, 12]],
        columns=['X', 'Y', 'Z'])

# convert dataframe to numpy array
arr = df.to_numpy()

print('\nNumpy Array\n----------\n', arr)
print(type(arr))
```

Output:

```
Numpy Array
----------
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
<class 'numpy.ndarray'>
```

11. How to add a column to a Pandas DataFrame?

We first create the dataframe and then look into the various methods one by one.

Code Example :

```
# Add column
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df['Address'] = address

# Observe the result
print(student_df)
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
    Name  Age  Marks  Address
0   Kate   10    85  Chicago
1  Harry   11    77   London
2 Sheila   12    91   Berlin
```

- **By declaring a new list as a column.**

Code Example :

```
# Add column
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df['Address'] = address

# Observe the result
print(student_df)
```

Output:-

```
    Name  Age  Marks
0    Kate   10    85
1   Harry   11    77
2  Sheila   12    91
    Name  Age  Marks  Address
0    Kate   10    85  Chicago
1   Harry   11    77   London
2  Sheila   12    91   Berlin
```

- **By using DataFrame.insert():** It gives the freedom to add a column at any position we like and not just at the end. It also provides different options for inserting the column values.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df.insert(2, "Address", ['Chicago', 'London', 'Berlin'], True)
# Observe the result
print(student_df)
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
    Name  Age  Address  Marks
0   Kate   10  Chicago    85
1  Harry   11   London    77
2 Sheila   12   Berlin    91
```

- **Using Dataframe.assign() method:** This method will create a new dataframe with a new column added to the old dataframe.

Code Example :

```
import pandas as pd
```

```
student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
df2 = student_df.assign(address=['Chicago', 'London', 'Berlin'])

# Observe the result
print(df2)
```

Output:-

```
    Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
    Name  Age  Marks  address
0   Kate   10    85  Chicago
1  Harry   11    77   London
2 Sheila   12    91   Berlin
```

- **By using a dictionary:** We can use a Python dictionary to add a new column in pandas DataFrame. Use an existing column as the key values and their respective values will be the values for a new column.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}
```

```
# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = {'Chicago': 'Kate', 'London': 'Harry',
'berlin': 'Sheila'}
student_df['Address'] = address

# Observe the output
print(student_df)
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
    Name  Age  Marks  Address
0   Kate   10    85  Chicago
1  Harry   11    77   London
2 Sheila   12    91   berlin
```

12. How to add a row to a Pandas DataFrame?

We can add a single row using DataFrame.loc: We can add the row at the last in our dataframe. We can get the number of rows using len(DataFrame.index) for determining the position at which we need to add the new row.

Code Example :

```
# Add row
import pandas as pd
```

```
student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
student_df.loc[len(student_df.index)] = ['Alex', 19, 93]
print(student_df)
```

Output:

```
    Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
    Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
2 Sheila   12    91
3   Alex   19    93
```

We can also add a new row using the DataFrame.append() function:

Code Example :

```
df2 = {'Name': 'Tom', 'Age': 18, 'Marks': 73}
student_df = student_df.append(df2, ignore_index = True)
print(student_df)
```

Output:

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   11    77
```

```
2  Sheila   12     91
3    Alex   19     93
4     Tom   18     73
```

We can also add multiple rows using the pandas.concat(): by creating a new dataframe of all the rows that we need to add and then appending this dataframe to the original dataframe.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

dict = {'Name':['Amy', 'Maddy'],
        'Age':[19, 12],
        'Marks':[93, 81]
       }
df2 = pd.DataFrame(dict)
print(df2)
df3 = pd.concat([df1, df2], ignore_index = True)

print(df3)
```

Output:

```
Name  Age  Marks
0   Kate   10     85
```

```
1   Harry   14    77
2  Sheila   12    91
   Name  Age  Marks
0    Amy   19    93
1  Maddy   12    81
    Name  Age  Marks
0   Harry   14    77
1  Sheila   12    91
2     Amy   19    93
3   Maddy   12    81
```

13. How to set Index to a Pandas DataFrame?

- **Changing Index column:** In this example, the First Name column has been made the index column of DataFrame.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)
```

Output:

```
    Name  Age  Marks
```

```
0   Kate   10    85
1   Harry   11    77
2  Sheila   12    91
       Age  Marks
Name
Kate     10    85
Harry    11    77
Sheila   12    91
```

- **Set Index using Multiple Column:** In this example, two columns will be made as an index column. The drop parameter is used to Drop the column and the append parameter is used to append passed columns to the already existing index column.

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index(['Name', 'Marks'])
print(student_df)
```

Output:

```
   Name  Age  Marks
0   Kate   10    85
```

```
1   Harry   11    77
2  Sheila   12    91
          Age
Name   Marks
Kate   85     10
Harry  77     11
Sheila 91     12
```

- **Set index using a List:**

Code Example :

```
index = pd.Index(['x1', 'x2', 'x3'])
student_df = student_df.set_index(index)
print(student_df)
```

Output:

```
  Age  Marks
x1   10    85
x2   11    77
x3   12    91
```

- **Set multi-index using a list and column**

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
```

```
print(student_df)
index = pd.Index(['x1', 'x2', 'x3'])
student_df = student_df.set_index([index, 'Name'])
print(student_df)
```

Output:-

```
    Name  Age  Marks
0   Kate   10    85
1   Harry  11    77
2  Sheila  12    91
         Age  Marks
  Name
x1 Kate    10    85
x2 Harry   11    77
x3 Sheila  12    91
```

Pandas Interview Questions for Data Scientists

1. When to use merge() over concat() and vice-versa in Pandas?

The use of concat() function comes into play when combining **homogeneous** DataFrame, while the merge() function is considered first when combining complementary DataFrame.

If we need to merge vertically, we should always use pandas.concat(). whereas if need to merge horizontally via columns, we should go with pandas.merge(), which by default merges on the columns that are in common between the dataframes.

Code Example :

```
df1 = pd.DataFrame({'Key': ['b', 'b', 'a', 'c'], 'data1': range(4)})

df2 = pd.DataFrame({'Key': ['a', 'b', 'd'], 'data2': range(3)})
```

```
#Merge
# The 2 dataframes are merged on the basis of values in column "Key" as it is
# a common column in 2 dataframes

print(pd.merge(df1, df2))

#Concat
# df2 dataframe is appended at the bottom of df1

print(pd.concat([df1, df2]))
```

Output:-

```
  Key  data1  data2
0   b      0      1
1   b      1      1
2   a      2      0
  Key  data1  data2
0   b    0.0    NaN
1   b    1.0    NaN
2   a    2.0    NaN
3   c    3.0    NaN
0   a    NaN    0.0
1   b    NaN    1.0
2   d    NaN    2.0
```

2. What's the difference between pivot_table() and groupby()?

Both pivot_table() and groupby() are used to aggregate your dataframe. The major difference is in the shape of the result.

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
table = pd.pivot_table(df, index =['Name', 'Age'])
print(table)
```

Output:-

```
    Name  Age  Marks
0    Kate   10    85
1   Harry   14    77
2  Sheila   12    91
         Marks
       Name   Age
Harry  14     77
Kate   10     85
Sheila 12     91
```

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12],
'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
```

```
gk = df.groupby('Age')
gk.first()
```

Output:

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   14    77
2 Sheila   12    91
Age  Name   Marks
10     Kate   85
12     Sheila 91
14     Harry 77
```

3. Compare the Pandas methods: map(), applymap(), apply()

- The **map()** method is an elementwise method for only Pandas Series, it maps values of the Series according to input correspondence.

```
import pandas as pd

# Series generation
str_string = 'scalar'
str_series = pd.Series(list(str_string))
print("Original series\n" +
str_series.to_string(index=False,
                     header=False), end='\n\n')

# Using apply method for converting characters
# present in the original series
new_str_series = str_series.map(str.upper)
```

```
print("Transformed series:\n" +
new_str_series.to_string(index=False,
                         header=False), end='\n\n')
```

Output:

```
Original series
s
c
a
l
a
r

Transformed series:
S
C
A
L
A
R
```

- The **applymap()** method is an elementwise function for only DataFrames, it applies a function that accepts and returns a scalar to every element of a DataFrame.

It accepts callables only i.e. a Python function.

Code Example :

```
import pandas as pd

# initialize a dataframe
```

```
df = pd.DataFrame(
        [['a', 'b', 'c'],
        ['d','e','f'],
        ],
        columns=['X', 'Y', 'Z'])

print(df)

new_df = df.applymap(str.upper)
print("Transformed dataframe:\n" +
new_df.to_string(index=False,
                 header=False), end='\n\n')
```

Output:-

```
  X  Y  Z
0  a  b  c
1  d  e  f

Transformed dataframe:
A B
D E F
```

- The **apply()** method also works elementwise, as it applies a function along the input axis of DataFrame. It is suited to more complex operations and aggregation.

Code Example :

```
import pandas as pd

# initialize a dataframe
df = pd.DataFrame(
```

```
        [[10, 12, 33],
        [41, 53, 66],
        [17, 81, 19],
        [10, 11, 12]],
        columns=['X', 'Y', 'Z'])

print(df)

new_df = df.apply(lambda x:x.sort_values(), axis = 1)

print("Transformed dataframe:\n" + \
new_df.to_string(index = False,
         header = False), end = '\n\n')
```

Output:-

```
   X   Y   Z
0  10  12  33
1  41  53  66
2  17  81  19
3  10  11  12

Transformed dataframe:
10 12 33
41 53 66
17 81 19
10 11 12
```

4. Describe a few data operations in Pandas.

There are several useful data operations for DataFrame in Pandas, which are as follows:

1. **String Operation:** Pandas provide a set of string functions for working with string data. The following are the few operations on string data:
    - **lower():** Any strings in the index or series are converted to lowercase letters.
    - **upper():** Any strings in the index or series are converted to uppercase letters.
    - **strip():** This method eliminates spacing from every string in the Series/index, along with a new line.
    - **islower():** If all of the characters in the Series/Index string are lowercase, it returns True. Otherwise, False is returned.
    - **isupper():** If all of the characters in the Series/Index string are uppercase, it returns True. Otherwise, False is returned.
    - **split(' '):** It's a method that separates a string according to a pattern.
    - **cat(sep=' '):** With a defined separator, it concatenates series/index items.
    - **contains(pattern):** If a substring is available in the current element, it returns True; otherwise, it returns False.
    - **replace(a,b):** It substitutes the value b for the value a.
    - **startswith(pattern):** If all of the components in the series begin with a pattern, it returns True.
    - **endswith(pattern):** If all of the components in the series terminate in a pattern, it returns True.
    - **find(pattern):** It can be used to return the pattern's first occurrence.
    - **findall(pattern):** It gives you a list of all the times the pattern appears.
    - **swapcase:** It is used to switch the lower/upper case.
2. **Null values:** When no data is being sent to the items, a Null value/missing value can appear. There may be no values in the

respective columns, which are commonly represented as NaN. Pandas provide several useful functions for identifying, deleting, and changing null values in Data Frames. The following are the functions.

- **isnull():** isnull 's job is to return true if either of the rows has null values.
- **notnull():** It is the inverse of the isnull() function, returning true values for non-null values.
- **dropna():** This function evaluates and removes null values from rows and columns.
- **fillna():** It enables users to substitute other values for the NaN values.
- **replace():** It's a powerful function that can take the role of a regex, dictionary, string, series, and more.
- **interpolate():** It's a useful function for filling null values in a series or data frame.

3. **Row and column selection:** We can retrieve any row and column of the DataFrame by specifying the names of the rows and columns. It is one-dimensional and is regarded as a Series when you select it from the DataFrame.
4. **Filter Data:** By using some of the boolean logic in DataFrame, we can filter the data.
5. **Count Values:** Using the 'value counts()' option, this process is used to count the overall possible combinations.

5. How to get items of series A not present in series B?

In order to find items from series A that are not present in series B by using the isin() method combining it with the **Bitwise NOT** operator in pandas. We can understand this using a code example:

Code Example :

```
# Importing pandas library
import pandas as pd

# Creating 2 pandas Series
series1 = pd.Series([12, 24, 38, 210, 110, 147, 929])
series2 = pd.Series([17, 83, 76, 54, 110, 929, 510])

print("Series1:")
print(series1)
print("\nSeries2:")
print(series2)

# Using Bitwise NOT operator along
# with pandas.isin()
print("\nItems of series1 not present in series2:")
res = series1[~series1.isin(series2)]
print(res)
```

Output:-

```
Series1:
0     12
1     24
2     38
3    210
4    110
5    147
6    929
dtype: int64

Series2:
0     17
```

```
1    83
2    76
3    54
4   110
5   929
6   510
dtype: int64

Items of series1 not present in series2:
0    12
1    24
2    38
3   210
5   147
dtype: int64
```

6. What is the use of pandas.Dataframe.aggregate() function? Explain its syntax and parameters.

Data Aggregation is defined as the process of applying some aggregation function to one or more columns. It uses the following:

- **sum:** It is used to return the sum of the values for the requested axis.
- **min:** It is used to return a minimum of the values for the requested axis.
- **max:** It is used to return maximum values for the requested axis.

Its **Syntax** is:

```
DataFrame.aggregate(func=None, axis=0, *args, **kwargs)
```

Aggregate using one or more operations over the specified axis.

Parameters:

- **func:** It takes string, list, dictionary, or function values as input. It represents the function to use for data aggregation.
- **axis:** It takes in only two values '0' or '1'. 0 is for the index and 1 is for columns.
  If 0 or 'index': The function is applied to each column.
  If 1 or 'columns': The function is applied to each row.
  The default value is set to 0.

It returns the aggregated dataframe as the output.

Code Example :

```
import pandas as pd

data = {
"x": [560, 240, 630],
"y": [300, 1112, 452]
}

df = pd.DataFrame(data)

x = df.aggregate(["sum"])
y = df.aggregate(["min"])
z = df.aggregate(["max"])

print(x)
print(y)
print(z)
```

Output:

```
     x    y
sum  1430  1864
     x   y
min  240  300
     x    y
max  630  1112
```

7. How to format data in your Pandas DataFrame?

When we start working on a dataset or a set of data we need to perform some operations on the values in the DataFrame. At times these values might not be in the right format for you to work on it thus formatting of data is required. There are multiple ways in which we can format data in a Pandas DataFrame.

- One way is by **Replacing All Occurrences of a String in a DataFrame**. In order to replace Strings in our DataFrame, we can use replace() method i.e. all we need is to pass the values that we would like to change, followed by the values we want to replace them with.
- One other way is by **Removing Parts From Strings in the Cells of the DataFrame**. Removing unwanted parts of strings is cumbersome work. Luckily, there is a solution in place! We can do it easily by using map() function on the column result to apply the lambda function over each element or element-wise of the column.
- **Splitting Text in a Column into Multiple Rows in a DataFrame**. The process of splitting text into multiple rows is quite a complex task. We can do so by applying a function to the Pandas DataFrame's Columns or Rows.

8. Explain the GroupBy function in Pandas

Python pandas Dataframe.groupby() function is used for grouping the data according to the categories and applying a function to those categories. It helps in data aggregation in an efficient manner. It splits the data into groups based on some given criteria. The pandas objects can be split on any of their axes. In brief groupby() provides the mapping of labels to their respective group names.

Syntax:

```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=_NoDefault.no_default, squeeze=_NoDefault.no_default, observed=False, dropna=True)
```

Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85, 77, 91]}

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

gk = df.groupby('Age')
gk.first()
```

Output:-

```
   Name  Age  Marks
0   Kate   10    85
1  Harry   14    77
```

```
2  Sheila   12     91
Name Marks
Age
10       Kate   85
12       Sheila 91
14       Harry  77
```

9. How can you find the row for which the value of a specific column is max or min?

We can find the row for which the value of a specific column is by using **idxmax** and **idxmin** functions.

Code Example :

```
import pandas as pd

data = {
"sales": [23, 34, 56],
"age": [50, 40, 30]
}

df = pd.DataFrame(data)

print(df.idxmax())
print(df.idxmin())
```

Output:-

```
sales    2
age      0
dtype: int64
sales    0
age      2
```

```
dtype: int64
```

Pandas Functions Documentation:

https://pandas.pydata.org/docs/reference/general_functions.html

https://www.programiz.com/python-programming/pandas/methods

https://www.kaggle.com/learn/pandas