Pandas

(Code: Subhajit Das)

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Use Cases

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

Installing Pandas

In [29]:
```
#pip install pandas
```

Importing Pandas

In [30]:
```
import pandas as pd
```

Series: A Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). It is like a column in a table

Syntax: pd.Seris(varname)

Parameters in pd.Series()

1. **data:** This parameter is mandatory and refers to the data from which the series will be created. It can be an ndarray, a dictionary, a scalar value, or a list.
2. **index:** This parameter is optional and is used to specify the index labels for the series. It can be an array-like or a list of labels. If not provided, a default integer index is used.
3. **dtype:** This parameter is optional and is used to specify the data type for the series. If not provided, the data type is inferred from the input data.
4. **copy:** This parameter is optional and is used to specify whether to make a copy of the data. If set to True, the series will be created with a copy of the data. If set to False, the series will be created with a reference to the data.
5. **name:** This parameter is optional and is used to specify a name for the series. It can be a string label or None. If not provided, the series will have no name.

In [31]:
```
# Printing series
x = (3,4,7)
var = pd.Series(x)

print(var)
print(type(var))
```

```
0    3
1    4
2    7
dtype: int64
<class 'pandas.core.series.Series'>
```

In [32]:
```
# Changing the index number

x = (3,4,7)
var = pd.Series(x, index=['a','b','c'])

print(var)
```

```
a    3
b    4
c    7
dtype: int64
```

In [33]:
```
# Indexing and Slicing

print(var[2])
print(var[0:1])
```

```
7
a    3
dtype: int64
```

In [34]:
```
# Giving a name of series and changing the datatype
x = [3,4,7]
var = pd.Series(x, dtype = 'string', name = 'Practice')

print(var)
print(type(var))
```

```
0    3
1    4
2    7
Name: Practice, dtype: string
<class 'pandas.core.series.Series'>
```

In [35]:
```
# Print dictionary in series
dict = {"lang":['python','java','C'], 'name':['Subha'], 'num':[2,7,4,1]}
var_dict = pd.Series(dict)

print(var_dict)
```

```
lang    [python, java, C]
name              [Subha]
num          [2, 7, 4, 1]
dtype: object
```

In [36]:
```
# Print a single data series
var_sing = pd.Series(18)

print(var_sing)
```

```
0    18
dtype: int64
```

In [37]:
```
# Print a single value multiple times in a data series
var_sing = pd.Series(18, index=[5,6,7,8])

print(var_sing)
```

```
5    18
6    18
7    18
8    18
dtype: int64
```

In [38]:
```
# Multiply two series
# Here we have imbalance data in two series, if we did this in numpy. It will throw
S1 = pd.Series(18, index=[5,6,7,8])
S2 = pd.Series(18, index=[5,6,7,8,9,10])

print(S1*S2)
```

```
5     324.0
6     324.0
7     324.0
8     324.0
9       NaN
10      NaN
dtype: float64
```

Data Frame: A DataFrame in Pandas is a two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is a data structure that contains labeled axes (rows and columns). It is also a size-mutable data structure, which means that the size of the DataFrame can be changed after it is created. The DataFrame can be thought of as a dict-like container for Series objects.

Syntax: pd.DataFrame(varname)

Parameters in pd.DataFrame()

1. **data:** The data to be stored in the DataFrame. It can be a list, dictionary, numpy array, or another DataFrame.
2. **index:** The row labels for the DataFrame. It can be a list, array, or a pandas Index object. If not specified, a default integer index will be used (0, 1, 2, etc.).
3. **columns:** The column labels for the DataFrame. It can be a list, array, or a pandas Index object. If not specified, default column labels will be used.
4. **dtype:** The data type to be used for the columns. If not specified, data types will be inferred from the data.
5. **copy:** Whether to create a deep copy of the data. By default, it is set to False.
6. **copy_deeptag:** Deprecated parameter that was used in earlier versions of pandas.

In [39]:
```
# Printing DataFrame
x = (3,4,7)
var = pd.DataFrame(x)

print(var)
print(type(var))
```

```
   0
0  3
1  4
2  7
<class 'pandas.core.frame.DataFrame'>
```

Dict to Lists

In [40]:
```
# Print dictionary in Data Frame
# Here all arrays must be of the same length
dict = {"lang":['python','java','C'], 'marks':[95, 80, 94]}
var_dict = pd.DataFrame(dict, index=['Sub 1', 'Sub 2', 'Sub 3'])

print(var_dict)
```

```
         lang  marks
Sub 1  python     95
Sub 2    java     80
Sub 3       C     94
```

In [41]:
```
# If, I want to print a particular column

dict = {"lang":['python','java','C'], 'marks':[95, 80, 94]}
var_dict = pd.DataFrame(dict, index=['Sub 1', 'Sub 2', 'Sub 3'], columns=['lang'])

print(var_dict)
```

```
         lang
Sub 1  python
Sub 2    java
Sub 3       C
```

In [42]:
```
# Printing a particular index value

dict = {"lang":['python','java','C'], 'marks':[95, 80, 94]}
var_dict = pd.DataFrame(dict, index=['Sub 1', 'Sub 2', 'Sub 3'])

print(var_dict['lang']['Sub 1'])
```

```
python
```

List of Lists

In [43]:
```
# Print lists in Data Frame
list = ['python','java','C'], [95, 80, 94]
var_list = pd.DataFrame(list, index=['Subjects', 'Marks'])

print(var_list)
```

```
              0     1   2
Subjects  python  java   C
Marks         95    80  94
```

Through 2 series

In [44]:
```
Ser = {'S1': pd.Series([1,2,3,4]), 'S2': pd.Series([5,6,7,8])}

var_merge = pd.DataFrame(Ser)
print(var_merge)
```

```
   S1  S2
0   1   5
1   2   6
2   3   7
3   4   8
```

Operators

In [45]:
```
# Arithmetic
var_op = pd.DataFrame({'Num1':[1,2,3,4], 'Num2':[10,20,30,40]})

var_op['Add'] = var_op['Num1'] + var_op['Num2']
var_op['Sub'] = var_op['Num1'] - var_op['Num2']
var_op['Multi'] = var_op['Num1'] * var_op['Num2']
var_op['Div'] = var_op['Num1'] / var_op['Num2']

print(var_op)
```

```
   Num1  Num2  Add  Sub  Multi  Div
0     1    10   11   -9     10  0.1
1     2    20   22  -18     40  0.1
2     3    30   33  -27     90  0.1
3     4    40   44  -36    160  0.1
```

In [46]:
```
# Comparison
Num = [8,4,2,9,6]

var_great = pd.DataFrame(Num)
var_great['Large than or equals to 6'] = var_great > 6
print(var_great)
```

```
   0  Large than or equals to 6
0  8                       True
1  4                      False
2  2                      False
3  9                       True
4  6                      False
```

Insert, Update and Delete Data

Insert

Syntax: var.insert() [var= DataFrame]

In [47]:
```
var = pd.DataFrame({'A': [1,2,3,4], 'B': [5,6,7,8]})

print(var)
```

```
   A  B
0  1  5
1  2  6
2  3  7
3  4  8
```

In [48]:
```
# Inserting a data with copying a column
var.insert(2,"C",var["A"]) # Here, 2 is the column position

print(var)
```

```
   A  B  C
0  1  5  1
1  2  6  2
2  3  7  3
3  4  8  4
```

In [49]:
```
# Inserting a data
var.insert(3,"D",[9,10,11,12]) # Here list data should be equals to prev data, othe

print(var)
```

```
   A  B  C   D
0  1  5  1   9
1  2  6  2  10
2  3  7  3  11
3  4  8  4  12
```

In [50]:
```
# Insert limited data with slicing
var['E'] = var['D'][:2]

print(var)
```

```
   A  B  C   D     E
0  1  5  1   9   9.0
1  2  6  2  10  10.0
2  3  7  3  11   NaN
3  4  8  4  12   NaN
```

Update

Syntax: old_var.update(new_var)

In [51]:
```
# Create a DataFrame
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]})

# Create another DataFrame with new data
new_df = pd.DataFrame({'B': [4, 5, 6]})

# Update the original DataFrame with the new data
df1.update(new_df)
# Display the updated DataFrame
print(df1)
```

```
   A  B
0  1  4
1  2  5
2  3  6
```

Delete

In [52]:
```
var_del = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600], 'C':[7,8,9]})

print(var_del)
```

```
   A    B  C
0  1  400  7
1  2  500  8
2  3  600  9
```

In [53]:
```
# delete a particual column using pop() function
var_pop = var_del.pop('B')

print("The deleted data is: ")
print(var_pop)

print("The original data right now: ")
print(var_del)
```

```
The deleted data is:
0    400
1    500
2    600
Name: B, dtype: int64
The original data right now:
   A  C
0  1  7
1  2  8
2  3  9
```

In [54]:
```
# delete a particual column using del() function
del var_del['C']

var_del
```

Out[54]:

	A
0	1
1	2
2	3

CSV File: A CSV (Comma Separated Values) file is a plain text file that stores tabular data (numbers and text) in a simple format, where each line of the file typically represents one data record.

Whereas an Excel file is a binary file that can store formatting, perform operations on data, and contain multiple worksheets

Write CSV file : Will get a file with data

Syntax: var.to_csv('.csv') [var=DataFrame]

In [55]:
```
sample = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600], 'C':[7,8,9], 'D':[1000

print(sample)

# To create csv file
# This index argument will not print the index in new csv file

#sample.to_csv('Sample.csv', index = False, header=['Num1','Num2','Num3','Num4'])
```

```
   A    B  C     D
0  1  400  7  1000
1  2  500  8  1100
2  3  600  9  1200
```

Read CSV File

Syntax: var=pd.read_csv(FilePath)

Parameters of pd.read_csv()

1. **filepath or file object:** The path or name of the CSV file to be read, or a file-like object containing the CSV data.
2. **sep or delimiter:** The delimiter character or sequence of characters used to separate the fields in the CSV file. The default is a comma (','), but it can be changed to any other character or sequence.
3. **header:** The row number or sequence of row numbers to be used as the column names of the DataFrame. By default, the function assumes that the first row contains the column names. If None is passed, no header is assumed.
4. **index_col:** The column or sequence of columns to be used as the index of the DataFrame. By default, the function doesn't set any column as the index.
5. **usecols:** The list of column names or column indices to be read from the CSV file. This parameter allows you to select a subset of columns to be included in the DataFrame. By default, all columns are read.
6. **nrows:** The number of rows to be read from the CSV file. This parameter can be used to read only a specific number of rows instead of the entire file.
7. **skiprows:** The number of rows or sequence of row numbers to be skipped from the start of the file while reading. It can be used to skip header rows or any other unnecessary rows.
8. **na_values:** A scalar, str, list, or dictionary specifying how missing values are represented in the CSV file. It allows you to specify custom values that should be treated as missing values.
9. **dtype:** A dictionary specifying the data type of specific columns. It allows you to specify the data types of columns that pandas may not be able to infer automatically.

In [56]:
```
csv_read = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_read
```

Out[56]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [57]:
```
# Reading particular rows and columns
csv_part = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_part
```

Out[57]:

	Not_AvailableME_of_Responder	AVG_HOUSEHOLD_INCOME
0	Raja Aarav Malhotra	105000
1	Subhajit Das	75000
2	Kavya Kumari	40000

In [58]:
```
# Skiping particular rows
csv_read = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_read
```

Out[58]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4	10500(
1	Subhajit Das	3	7500(
2	Arpita Chaudhury	4	10000(

In [59]:
```
# Setting index value as column
csv_col = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Prac

csv_col
```

Out[59]:

Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOME
Raja Aarav Malhotra	4.0	105000.0
Subhajit Das	3.0	75000.0
Kavya Kumari	NaN	40000.0
Malabika Ghosh	10.0	NaN
Arpita Chaudhury	4.0	100000.0

In [60]:
```
# Change the header and set a row name
csv_col = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Prac

csv_col
```

Out[60]:

	Malabika Ghosh	10	Unnamed: 2	BagNot_Availablen, Howrah, West Bengal.	38.5	Non-veg	60000	TV, FRIDGE, WM, 2W, Geyser, kitchen chimney	Tap Water	Unnamed: 9	Not
0	Arpita Chaudhury	4	100000	Sakherbazar Behala, West Bengal	34.5	Veg	50000	TV AC WM FRIDGE 4W	Tap Water	Laptop Smartphone	Nc

In [61]:
```
# Change the column name manually
csv_colname = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda

csv_colname
```

Out[61]:

	1	2	3
0	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOME
1	Raja Aarav Malhotra	4	105000
2	Subhajit Das	3	75000
3	Kavya Kumari	NaN	40000
4	Malabika Ghosh	10	NaN
5	Arpita Chaudhury	4	100000

In [62]:
```
# Set automatic column name
csv_auto = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_auto
```

```
<ipython-input-62-78085898167b>:2: FutureWarning: The prefix argument has been de
precated and will be removed in a future version. Use a list comprehension on the
column names in the future.


  csv_auto = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda
Practice.csv", header = None, prefix = 'Column Name')
```

Out[62]:

	Column Name0	Column Name1	Column Name2
0	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOME
1	Raja Aarav Malhotra	4	105000
2	Subhajit Das	3	75000
3	Kavya Kumari	NaN	40000
4	Malabika Ghosh	10	NaN
5	Arpita Chaudhury	4	100000

In [63]:
```
# Change datatype of a column data
csv_read = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_read
```

Out[63]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

Pandas Function

Functions in panda:

1. **pd.read_csv():** Reads a CSV file into a Pandas DataFrame.
2. **pd.DataFrame():** Creates a new DataFrame from data like arrays, lists, or dictionaries.
3. **df.head():** Returns the first n rows of a DataFrame.
4. **df.tail():** Returns the last n rows of a DataFrame.
5. **df.info():** Provides a summary of the DataFrame's structure, including column names, data types, and non-null values.
6. **df.describe():** Generates descriptive statistics of the DataFrame, such as count, mean, min, and max for each column.
7. **df.shape:** Returns a tuple of the number of rows and columns in the DataFrame.
8. **df.columns:** Returns the column labels of the DataFrame.
9. **df.values:** Returns the underlying data as a numpy array.
10. **df.isnull():** Returns a DataFrame of the same shape as the input, with True values wherever the input is null, and False elsewhere.
11. **df.dropna():** Removes rows or columns with null values from the DataFrame.
12. **df.fillna():** Fills null values in the DataFrame with a specified value or method.
13. **df.groupby():** Groups the DataFrame by one or more columns and performs an aggregate function on each group.
14. **df.merge():** Combines two DataFrames based on a common column or index.
15. **df.sort_values():** Sorts the DataFrame by one or more columns in ascending or descending order.
16. **df.pivot_table():** Creates a pivot table from a DataFrame by specifying the index, columns, and values.
17. **df.apply():** Applies a function to each element or column of the DataFrame.
18. **df.plot():** Creates various types of plots, such as line plots, bar plots, and scatter plots.
19. **df.to_csv():** Writes the DataFrame to a CSV file.
20. **df.to_excel():** Writes the DataFrame to an Excel file.

Note: "df" refers to a DataFrame object.

In [64]:
```
csv = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Practice

# Get row values
print(csv.index)

# Get column values
print(csv.columns)

# Get description of data
csv.describe()
```

```
RangeIndex(start=0, stop=5, step=1)
Index(['Not_AvailableME_of_Responder', 'NUMBER_OF_MEMBERS_IN_HOUSEHOLD',
       'AVG_HOUSEHOLD_INCOME', 'HOUSE_ADDRESS', 'AGE_OF_MEMBERS',
       'FOOD_CONSUMPTION_PATTERNS', 'EXPENDITURES', 'HOUSEHOLD_ASSETS',
       'SOURCES_OF_WATER', 'AVAILABILITY_OF_SMART_DEVICES',
       'Other_SmartDevices', 'ACCESS_TO_INTERNET', 'ANY_PREDOMINANT_AILMENT'],
      dtype='object')
```

Out[64]:

	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOME	AGE_OF_MEMBERS	EXP
count	4.000000	4.000000	5.000000	
mean	5.250000	80000.000000	38.600000	5
std	3.201562	29720.924167	3.974921	1
min	3.000000	40000.000000	34.500000	3
25%	3.750000	66250.000000	35.500000	4
50%	4.000000	87500.000000	38.500000	5
75%	5.500000	101250.000000	40.000000	6
max	10.000000	105000.000000	44.500000	6

In [65]:
```
# To see top some rows of table
csv.head(2) # By default it's 5, if we not pass any parameter
```

Out[65]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(

In [66]:
```
# To see buttom some rows of table
csv.tail(2) # By default it's 5, if we not pass any parameter
```

Out[66]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [67]:
```
# Row slicing to see particular rows
csv[2:4]
```

Out[67]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaI

In [68]:
```
# Column slicing to see particular columns
csv.iloc[:, :2] # Also we can write loc[] instead of iloc[]
```

Out[68]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD
0	Raja Aarav Malhotra	4.0
1	Subhajit Das	3.0
2	Kavya Kumari	NaN
3	Malabika Ghosh	10.0
4	Arpita Chaudhury	4.0

In [69]:
```
# Particular data through slicing
csv.loc[[2,4],["Not_AvailableME_of_Responder",'AVG_HOUSEHOLD_INCOME']]
```

Out[69]:

	Not_AvailableME_of_Responder	AVG_HOUSEHOLD_INCOME
2	Kavya Kumari	40000.0
4	Arpita Chaudhury	100000.0

In [70]:
```
# Getting a particular value through iloc[]
csv.iloc[0,2]
```

Out[70]:
```
105000.0
```

In [71]:
```
# Change index datatype to array
print(type(csv))

csv.index.array
```

```
<class 'pandas.core.frame.DataFrame'>
```

Out[71]:
```
<PandasArray>
[0, 1, 2, 3, 4]
Length: 5, dtype: int64
```

In [72]:
```
# Change full dataframe to 2D numpy array
csv.to_numpy() # We can also write:    var = np.asarray(csv)
```

Out[72]:
```
array([['Raja Aarav Malhotra', 4.0, 105000.0,
        'Barrackpore, West Bengal, India', 44.5, 'NonVeg', 65000,
        'TV, FRIDGE, AC, WM, 2W', 'Tubewell',
        'Computer, Laptop, Smartphone', 'Not_Available', 'Yes',
        'Type-1 Diabetes'],
       ['Subhajit Das', 3.0, 75000.0, 'Howrah, West Bengal, India', 35.5,
        'Veg', 45000, 'TV, AC, FRIDGE, WM, 4W', 'Tubewell',
        'Computer, Smartphone', nan, 'Yes', 'No'],
       [' Kavya Kumari', nan, 40000.0, 'Dum Dum', 40.0, nan, 35000,
        'TV, FRIDGE, WM', nan, 'Laptop, Smartphone', 'Not_Available',
        'Yes', 'Type-2 Diabetes'],
       ['Malabika Ghosh', 10.0, nan,
        'BagNot_Availablen, Howrah, West Bengal.', 38.5, 'Non-veg',
        60000, 'TV, FRIDGE, WM, 2W, Geyser, kitchen chimney',
        'Tap Water', nan, 'Not_Available', 'Yes', 'Diabetes'],
       ['Arpita Chaudhury', 4.0, 100000.0,
        'Sakherbazar Behala , West Bengal', 34.5, 'Veg', 50000,
        'TV AC WM FRIDGE 4W', 'Tap Water', 'Laptop Smartphone',
        'Not_Available', 'No', 'No']], dtype=object)
```

In [73]:
```
# Sorting rows to Desending order
csv.sort_index(axis = 0, ascending = False) # Here axis = 0 means row wise and 1 me
```

Out[73]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
4	Arpita Chaudhury	4.0	100000.(
3	Malabika Ghosh	10.0	NaN
2	Kavya Kumari	NaN	40000.(
1	Subhajit Das	3.0	75000.(
0	Raja Aarav Malhotra	4.0	105000.(

Update a particular data of csv file

In [74]:
```
csv.loc[0,'Not_AvailableME_of_Responder'] = 'Subhajit'

csv
```

Out[74]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Subhajit	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [75]:
```
# Drop a column
csv.drop('NUMBER_OF_MEMBERS_IN_HOUSEHOLD', axis = 1) # For column axis is 1
```

Out[75]:

	Not_AvailableME_of_Responder	AVG_HOUSEHOLD_INCOME	HOUSE_ADDRESS	AGE_OF_MEMBERS
0	Subhajit	105000.0	Barrackpore, West Bengal, India	44.5
1	Subhajit Das	75000.0	Howrah, West Bengal, India	35.5
2	Kavya Kumari	40000.0	Dum Dum	40.0
3	Malabika Ghosh	NaN	BagNot_Availablen, Howrah, West Bengal.	38.5
4	Arpita Chaudhury	100000.0	Sakherbazar Behala , West Bengal	34.5

In [76]:
```
# Drop a row
csv.drop(4, axis = 0)
```

Out[76]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Subhajit	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN

In [77]:
```
# Rename a particular column name

csv.rename(columns={'Not_AvailableME_of_Responder': 'Name_of_Responder', 'AVG_HOUSE
```

Out[77]:

	Name_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_INCOME	HOUSE_ADDRESS	A
0	Subhajit	4.0	105000.0	Barrackpore, West Bengal, India	
1	Subhajit Das	3.0	75000.0	Howrah, West Bengal, India	
2	Kavya Kumari	NaN	40000.0	Dum Dum	
3	Malabika Ghosh	10.0	NaN	BagNot_Availablen, Howrah, West Bengal.	
4	Arpita Chaudhury	4.0	100000.0	Sakherbazar Behala , West Bengal	

Handling Missing Data (Dropna and Fillna)

Parameters of dropna function

1. **axis:** int or str (default 0)
   - Specifies the axis along which the missing values should be dropped.
   - If axis=0, it drops the rows containing missing values.
   - If axis=1, it drops the columns containing missing values.
2. **how:** str (default 'any')
   - Specifies the condition on which the rows or columns should be dropped.
   - If how='any', it drops the rows or columns that have any missing values.
   - If how='all', it drops the rows or columns that have all missing values.
3. **thresh:** int (default None)
   - Specifies the minimum number of non-null values required to keep the rows or columns.
   - If a row or column has less than the specified threshold value of non-null values, it will be dropped.
4. **subset:** array-like (default None)
   - Specifies a subset of columns or rows where missing values should be considered.
   - If subset is specified, only the specified columns or rows will be checked for missing values.
   - It can be a single column/row name or a list of column/row names.
5. **inplace:** bool (default False)
   - Modifies the DataFrame in place if True, without creating a new object.
   - If inplace is False, it returns a new DataFrame with missing values dropped.

Dropna: Drop the NaN values

Syntax: var.dropna()

In [78]:
```
csv_miss = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Pra

csv_miss
```

Out[78]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [79]:
```
# Delete rows that contains NaN
csv_miss.dropna()
```

Out[79]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [80]:
```
# Delete columns that contains NaN
csv_miss.dropna(axis = 1)
```

Out[80]:

	Not_AvailableME_of_Responder	HOUSE_ADDRESS	AGE_OF_MEMBERS	EXPENDITURES	HOUSEHO
0	Raja Aarav Malhotra	Barrackpore, West Bengal, India	44.5	65000	TV, FRID
1	Subhajit Das	Howrah, West Bengal, India	35.5	45000	TV, AC, F
2	Kavya Kumari	Dum Dum	40.0	35000	TV, F
3	Malabika Ghosh	BagNot_Availablen, Howrah, West Bengal.	38.5	60000	TV, FRID(Geyser, kitc
4	Arpita Chaudhury	Sakherbazar Behala , West Bengal	34.5	50000	TV AC WM

In [81]:
```
# Remove fully blank row with dropna
csv_miss.dropna(how='all')
```

Out[81]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [82]:
```
# Remove missing data of particular column / remove that row
csv_miss.dropna(subset=["Other_SmartDevices"])
```

Out[82]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [83]:
```
# Remove row that contains 2 null values
csv_miss.dropna(thresh = 2) # If we give 1, it will remove those row which contains
```

Out[83]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [84]:
```
# Get a new file of missing data
csv_miss.dropna(inplace = True)
csv_miss
```

Out[84]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

Fillna: Fill NaN values

Syntax: var.fillna()

Parameters of Fillna Function

1. **value:** This parameter can take a scalar value, dictionary, Series, or DataFrame. It represents the value to be used for replacement. If it is a dictionary, then it is used to specify a value for each column or index label.
2. **method:** This parameter can take values like 'ffill', 'bfill', 'pad', or 'backfill'. If used, it replaces the missing values using the specified method. 'ffill' is used to fill the previous valid value forward, 'bfill' is used to fill the next valid value backward, 'pad' is equivalent to 'ffill', and 'backfill' is equivalent to 'bfill'.
3. **axis:** This parameter determines the axis along which the NaN values are filled. By default, it is set to 0 and fills missing values vertically, column-wise. If set to 1, missing values are filled horizontally, row-wise.
4. **inplace:** This parameter is a boolean value that determines if the operation should be performed in-place or if a new object should be returned. If set to True, the DataFrame or Series is modified in-place, while if set to False, a new object with the filled values is returned.
5. **limit:** This parameter is used when the method parameter is set to 'ffill' or 'bfill'. It specifies the maximum number of consecutive NaN values to forward or backward fill.
6. **downcast:** This parameter allows for downcasting of integer or floating-point data types. It can take values like 'infer', 'integer', 'signed', 'unsigned', 'float', etc.

In [85]:
```
# Fill null value with a data
csv_miss.fillna('No Data')
```

Out[85]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [86]:
```
# Fill null values in column with a different data
csv_miss.fillna({"Other_SmartDevices": "Not Available", "ANY_PREDOMINANT_AILMENT":
```

Out[86]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [87]:
```
# Fill null values with forward data
csv_miss.fillna(method = "ffill")
```

Out[87]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [88]:
```
# Fill null values with backward data
csv_miss.fillna(method = "bfill", axis = 1) # Here, axis will fill data backward co
```

Out[88]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [89]:
```
# Fill data with a limit
csv_miss.fillna("No Data", limit=2) # It will fill 2 NaN values
```

Out[89]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [90]:
```
# Get a new file of missing data with filled values
csv_miss.fillna("No Data", inplace = True)
csv_miss
```

Out[90]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
4	Arpita Chaudhury	4.0	100000.(

In [91]:
```
# Replace all null values with the mean, only valid for int and float
csv.fillna(csv.mean()) # For mode: mode()
```

```
<ipython-input-91-5c9c8e98591d>:2: FutureWarning: The default value of numeric_on
ly in DataFrame.mean is deprecated. In a future version, it will default to Fals
e. In addition, specifying 'numeric_only=None' is deprecated. Select only valid c
olumns or specify the value of numeric_only to silence this warning.
  csv.fillna(csv.mean()) # For mode: mode()
```

Out[91]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Subhajit	4.00	105000.(
1	Subhajit Das	3.00	75000.(
2	Kavya Kumari	5.25	40000.(
3	Malabika Ghosh	10.00	80000.(
4	Arpita Chaudhury	4.00	100000.(

Handling Missing Values (Replace and Interpolate)

Replace: Replace() function is used to replace a string, regex, list, dictionary, series, number, etc. from a Pandas Dataframe in Python

Syntax: var.replace()

Parameters of Replace Function

1. **to_replace:** This parameter takes the value or pattern to be replaced. It can be a single value, a list of values, or a dictionary of values.
2. **value:** This parameter is used to specify the new value to replace the `to_replace` value. It can be a single value, a list of values, or a dictionary of values. When `to_replace` is given as a dictionary, the `value` parameter should be None.
3. **inplace:** This parameter is optional and by default set to False. If set to True, it modifies the DataFrame in-place and returns None. If set to False, it returns a new DataFrame with the replaced values, leaving the original DataFrame unchanged.
4. **limit:** This parameter is optional and specifies the maximum number of replacements to be made. By default, it is set to None, meaning all occurrences will be replaced.
5. **regex**: This parameter is optional and by default set to False. If set to True.
6. **method:** This parameter is optional and is used to specify the interpolation method when replacing missing values. It can take values like 'pad', 'ffill', 'bfill', etc.

In [92]:
```
csv_val = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/NSCA_DC Panda Prac

csv_val
```

Out[92]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [93]:
```
# Replace a particular value to another value
csv_val.replace(to_replace = 40.0, value = 60.0)
```

Out[93]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [94]:
```
# Replace multiple data at a time
csv_val.replace([44.5, 35.5, 60.0], 40.0)
```

Out[94]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [95]:
```
# Replace A to Z, a to z data with No Data
csv_val.replace("[A-Za-z]","No Data",regex=True)
```

Out[95]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	No DataNo DataNo DataNo Data No DataNo DataNo ...	4.0	105000.(
1	No DataNo DataNo DataNo DataNo DataNo DataNo D...	3.0	75000.(
2	No DataNo DataNo DataNo DataNo Data No DataNo...	NaN	40000.(
3	No DataNo DataNo DataNo DataNo DataNo DataNo D...	10.0	NaN
4	No DataNo DataNo DataNo DataNo DataNo Data No ...	4.0	100000.(

In [96]:
```
# Replace A to Z, a to z data with No Data of particular column
csv_val.replace({"HOUSE_ADDRESS": "[A-Za-z]"},"No Data",regex=True)
```

Out[96]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [97]:
```
# Replace a particular data with forward filling
csv_val.replace(40.0,method="ffill")
```

Out[97]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [98]:
```
# Replace a particular data with backward filling
csv_val.replace(40.0,method="bfill")
```

Out[98]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [99]:
```
# Replace a particular data with backward filling and limit
csv_val.replace('Yes',method="bfill",limit=2)
```

Out[99]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

In [100]:
```
# If we want to copy of this data
csv_val.replace('Yes', method="bfill", limit=2, inplace = True)

csv_val
```

Out[100]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	NaN	40000.(
3	Malabika Ghosh	10.0	NaN
4	Arpita Chaudhury	4.0	100000.(

Interpolate: This method takes a number of arguments, including the method argument, which specifies the type of interpolation to use. The default method is linear, which will interpolate between the known values using a straight line. Other methods available include time, quadratic, cubic, and spline.

Syntax: var.interpolate()

Parameters of Interpolate Function:

1. **method:** This parameter specifies the interpolation method to be used. The default method is "linear", but you can also use "nearest", "zero", "slinear", "quadratic", "cubic", "spline", "barycentric", "polynomial", or "krogh". Each method uses a different approach to interpolate values.
2. **axis:** This parameter specifies the axis along which to interpolate values. The default value is 0, which means that interpolation is performed along the columns (vertically). If you want to interpolate values along the rows (horizontally), you can pass `axis=1`.
3. **limit:** This parameter is used to limit the number of consecutive NaN values that can be filled. For example, if `limit=2`, it will fill up to 2 consecutive NaN values, but if there are more than 2 consecutive NaN values, it will not fill them.
4. **inplace:** This parameter specifies whether to perform the interpolation operation in place or return a new DataFrame with interpolated values. The default value is `False`, which means that the function returns a new DataFrame.
5. **limit_direction:** This parameter determines the direction in which the interpolation is performed. It can take either "forward", "backward", or "both". By default, it is set to "forward", which means that interpolation starts from the first valid value and proceeds forward.
6. **limit_area:** This parameter specifies the area of values to be used when limiting the number of consecutive NaN values that can be filled. It can take either "inside", "outside", or "both". The default value is "inside", which means that only NaN values surrounded by valid values on both sides will be filled.

In [101]:
```
# Fill data automatically, only valid for int and float data type
# By default, it will fill data linearly
csv_val.interpolate()
```

Out[101]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	6.5	40000.(
3	Malabika Ghosh	10.0	70000.(
4	Arpita Chaudhury	4.0	100000.(

In [102]:
```
''' If we want to change from linear, we have to use a method parameter and pass ar
    barycentric, krogh, akima, pchip, polynomial, spline) '''
csv_val.interpolate(method = "index")
```

Out[102]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	6.5	40000.(
3	Malabika Ghosh	10.0	70000.(
4	Arpita Chaudhury	4.0	100000.(

In [103]:
```
# Interpolate with limit
csv_val.interpolate(limit_direction = "forward", limit = 2) # We have forward, back
```

Out[103]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	6.5	40000.(
3	Malabika Ghosh	10.0	70000.(
4	Arpita Chaudhury	4.0	100000.(

In [104]:
```
# Interpolate in a area
csv_val.interpolate(limit_area = "inside") # Outside will fill the data of outter a
```

Out[104]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	6.5	40000.(
3	Malabika Ghosh	10.0	70000.(
4	Arpita Chaudhury	4.0	100000.(

In [105]:
```
# Copy of the orginal data
csv_val.interpolate(inplace = True)
csv_val
```

Out[105]:

	Not_AvailableME_of_Responder	NUMBER_OF_MEMBERS_IN_HOUSEHOLD	AVG_HOUSEHOLD_INCOMI
0	Raja Aarav Malhotra	4.0	105000.(
1	Subhajit Das	3.0	75000.(
2	Kavya Kumari	6.5	40000.(
3	Malabika Ghosh	10.0	70000.(
4	Arpita Chaudhury	4.0	100000.(

Merging and Concat

Merge

Syntax: pd.merge(var1, var2)

Parameters of Merge Function:

1. **how:** Type of merge to be performed. Options are ‘left’, ‘right’, ‘outer’, ‘inner’, ‘cross’. The default is ‘inner’.
2. **on:** Column or index level names to join on. These must be found in both DataFrames.
3. **left_on:** Column or index level names to join on in the left DataFrame.
4. **right_on:** Column or index level names to join on in the right DataFrame.
5. **left_index:** Use the index from the left DataFrame as the join key(s). Default is False.
6. **right_index:** Use the index from the right DataFrame as the join key. Default is False.
7. **sort:** Sort the join keys lexicographically in the result DataFrame. Default is False.
8. **suffixes:** A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in left and right respectively.

9. **indicator:** To indicate if the join keys were present in the input dataframes. If `indicator` is set to `True` , an additional column named `_merge` is added to the output dataframe. This `_merge` column can have three types of values:

- `both` : The join key is found in both dataframes.
- `left_only` : The join key is only found in the left dataframe.

In [106]:
```
# B is also common in both dataframe, so it will take as B_x and B_y
var1 = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]})
var2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.merge(var1,var2,on='A') # Here, on defines the common row of both tale
```

Out[106]:

	A	B_x	B_y
0	1	400	700
1	2	500	800
2	3	600	900

In [107]:
```
var1 = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]})
var2 = pd.DataFrame({'A': [1, 2, 3], 'C': [700, 800, 900]})

pd.merge(var2,var1,on='A') # To get C column first, var2 are passning first as argu
```

Out[107]:

	A	C	B
0	1	700	400
1	2	800	500
2	3	900	600

In [108]:
```
# Adding Suffix
var1 = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]})
var2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.merge(var1,var2,on='A', suffixes=("Lot","Lot"))
```

Out[108]:

	A	BLot	BLot
0	1	400	700
1	2	500	800
2	3	600	900

In [109]:
```
# Another method of merging
var1 = pd.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]})
var2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.merge(var1, var2, left_index = True, right_index = True, suffixes=("Id","Lot"))
```

Out[109]:

	AId	BId	ALot	BLot
0	1	400	1	700
1	2	500	2	800
2	3	600	3	900

In [110]:
```
var1 = pd.DataFrame({'A': [1, 2, 3, 4, 5], 'B': [400, 500, 600, 700, 800]})
var2 = pd.DataFrame({'A': [1, 2, 3, 5, 6], 'C': [900, 1000, 1100, 1200, 1300]})

pd.merge(var1, var2, how='outer', indicator = True) # How will fill missing values,
                                                    # Indicator will show, which da
```

Out[110]:

	A	B	C	_merge
0	1	400.0	900.0	both
1	2	500.0	1000.0	both
2	3	600.0	1100.0	both
3	4	700.0	NaN	left_only
4	5	800.0	1200.0	both
5	6	NaN	1300.0	right_only

In [111]:
```
# Getting the mean value
var1 = pd.DataFrame({'A': [1, 2, 3, 4, 5], 'B': [400, 500, 600, 700, 800]})
var2 = pd.DataFrame({'A': [1, 2, 3, 5, 6], 'C': [900, 1000, 1100, 1200, 1300]})

var = pd.merge(var1, var2)
var.mean() # Also we can use min() and max()
```

Out[111]:
```
A       2.75
B     575.00
C    1050.00
dtype: float64
```

Concat

Syntax: pd.concat([var1, var2])

Parameters of Concat Function

1. **objs:** A sequence or mapping of Series or DataFrame objects.
2. **axis:** The axis to concatenate along. Default is 0.
3. **join:** How to handle indexes on other axes. Options are ‘inner’, ‘outer’. Default is ‘outer’.
4. **ignore_index:** If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, …, n - 1. Default is False.
5. **keys:** Sequence to add an identifier to the result indexes. Default is None.
6. **levels:** Specific levels (unique values) to use for constructing a MultiIndex. Default is None.
7. **names:** Names for the levels in the resulting hierarchical index. Default is None.
8. **verify_integrity:** Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation. Default is False.
9. **sort:** Sort non-concatenation axis if it is not already aligned when join is ‘outer’. Default is False.
10. copy: If False, do not copy data unnecessarily. Default is True.

In [112]:
```
# Concat Series
sr1 = pd.Series([1,2,3,4])
sr2 = pd.Series([100,200,300,400])

pd.concat([sr1,sr2])
```

Out[112]:
```
0      1
1      2
2      3
3      4
0    100
1    200
2    300
3    400
dtype: int64
```

In [113]:
```
# Concat DataFrame
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.concat([df1,df2], axis=1) # 0 will concat row wise and 1 column wise. By default
```

Out[113]:

	A	B	A	B
0	1	400	1.0	700.0
1	2	500	2.0	800.0
2	3	600	3.0	900.0
3	4	1000	NaN	NaN

In [114]:
```
# Using join in concat
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.concat([df1,df2], axis=1, join='inner') # We can pass inner and outer join. Inne
```

Out[114]:

	A	B	A	B
0	1	400	1	700
1	2	500	2	800
2	3	600	3	900

In [115]:
```
# Concat based on key
# Using join in concat
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'A': [1, 2, 3], 'B': [700, 800, 900]})

pd.concat([df1,df2], axis=0, keys=["df1","df2"])
```

Out[115]:

		A	B
df1	0	1	400
	1	2	500
	2	3	600
	3	4	1000
df2	0	1	700
	1	2	800
	2	3	900

Join and Append

Join

Syntax: var1.join(var2)

Parameters of Join Function:

1. **other:** This parameter specifies the DataFrame, Series, or dict of DataFrames/Series to join or concatenate with.
2. **on:** This parameter specifies the column name(s) or index level(s) to join on. If not specified and the two DataFrames have a common column or index, it will be used.
3. **how:** This parameter specifies the type of join to perform. It can take values like 'inner', 'outer', 'left', or 'right'. By default, it is set to 'inner'.
4. **lsuffix and rsuffix:** These parameters specify suffixes to add to overlapping column names in the left and right DataFrames, respectively. This is useful when joining DataFrames with common column names.
5. **sort:** This parameter specifies whether to sort the result DataFrame by the join keys. By default, it is set to False.
6. **indicator:** This parameter specifies whether to include a column named '_merge' that indicates the source of each row in the result DataFrame. It can take values like True or False. By default, it is set to False.
7. **suffixes:** This parameter specifies suffixes to add to overlapping column names after the join. It is a tuple of strings where the first element is added to the left DataFrame and the second element is added to the right DataFrame.

In [116]:
```
# We can't able to give same column name in join
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'C': [1, 2], 'D': [700, 800]})

df1.join(df2)
```

Out[116]:

	A	B	C	D
0	1	400	1.0	700.0
1	2	500	2.0	800.0
2	3	600	NaN	NaN
3	4	1000	NaN	NaN

In [117]:
```
# If we want to add index
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]}, index=["a","b",
df2 = pd.DataFrame({'C': [1, 2], 'D': [700, 800]}, index=["a","b"])

df1.join(df2)
```

Out[117]:

	A	B	C	D
a	1	400	1.0	700.0
b	2	500	2.0	800.0
c	3	600	NaN	NaN
d	4	1000	NaN	NaN

In [118]:
```
# Using other joins
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'C': [1, 2], 'D': [700, 800]})

df2.join(df1, how="outer") # We can use left, right, inner and outter join accordin
```

Out[118]:

	C	D	A	B
0	1.0	700.0	1	400
1	2.0	800.0	2	500
2	NaN	NaN	3	600
3	NaN	NaN	4	1000

In [119]:
```
# Join will throw error if both column names are same to avoid this, we have to use
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
df2 = pd.DataFrame({'B': [1, 2], 'C': [700, 800]})

df1.join(df2, lsuffix = "_Table", rsuffix="_Table")
```

Out[119]:

	A	B_Table	B_Table	C
0	1	400	1.0	700.0
1	2	500	2.0	800.0
2	3	600	NaN	NaN
3	4	1000	NaN	NaN

Append: This function will be remove in later pandas version, instead of this we can use concat function

Syntax: var1.append(var2)

Parameters of Append Function:

1. **other:** This is the object to be appended. It can be a Series, DataFrame, or a dictionary/list of Series/DataFrames.
2. **ignore_index:** It is an optional parameter that can be set as True or False. If set as True, it will reset the index in the resulting DataFrame. The default value is False.

In [120]:
```
# Concat and append is same in this case
ap1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
ap2 = pd.DataFrame({'B': [1, 2], 'C': [700, 800]})

ap1.append(ap2) #pd.concat([ap1,ap2]) also give same result
```

```
<ipython-input-120-1fbf47fbe928>:5: FutureWarning: The frame.append method is dep
recated and will be removed from pandas in a future version. Use pandas.concat in
stead.
  ap1.append(ap2) #pd.concat([ap1,ap2]) also give same result
```

Out[120]:

	A	B	C
0	1.0	400	NaN
1	2.0	500	NaN
2	3.0	600	NaN
3	4.0	1000	NaN
0	NaN	1	700.0
1	NaN	2	800.0

In [121]:
```
# Set the index number syncronously
ap1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [400, 500, 600, 1000]})
ap2 = pd.DataFrame({'C': [1, 2], 'D': [700, 800]})

ap1.append(ap2, ignore_index=True) #pd.concat([ap1,ap2], ignore_index=True) also gi
```

```
<ipython-input-121-25d42fa1ee0e>:5: FutureWarning: The frame.append method is dep
recated and will be removed from pandas in a future version. Use pandas.concat in
stead.
  ap1.append(ap2, ignore_index=True) #pd.concat([ap1,ap2], ignore_index=True) als
o give same result
```

Out[121]:

	A	B	C	D
0	1.0	400.0	NaN	NaN
1	2.0	500.0	NaN	NaN
2	3.0	600.0	NaN	NaN
3	4.0	1000.0	NaN	NaN
4	NaN	NaN	1.0	700.0
5	NaN	NaN	2.0	800.0

Group By: Pandas groupby is used for grouping the data according to the categories and applying a function to the categories

Parameters of GroupBy Function:

1. **by:** This parameter specifies the column or columns to group the data by. It can be a column name or a list of column names.
2. **axis:** This parameter specifies whether to group along rows (`axis=0`) or columns (`axis=1`).
3. **level:** This parameter is used for hierarchical indexing. It specifies the level(s) to group by in a multi-index DataFrame.
4. **as_index:** This parameter determines whether the grouping columns should be included as part of the index (`as_index=True`) or as regular columns (`as_index=False`).
5. **sort:** This parameter specifies whether to sort the groups by the grouping columns. By default, it is set to `True` .
6. **group_keys:** This parameter determines whether to include the grouping keys as part of the index for each group when calling `.apply()` on a grouped DataFrame.

There are additional parameters like `dropna` , `squeeze` , etc., that can be used depending on the specific use case.

In [122]:
```
grp = pd.DataFrame({'Name': ['a','b','c','c','d','b','a','c','d','a'],
                    'Sub_1': ['4','6','2','3','7','4','8','6','3','4'],
                    'Sub_2': ['2','7','6','4','3','7','8','4','6','2']})

grp
```

Out[122]:

	Name	Sub_1	Sub_2
0	a	4	2
1	b	6	7
2	c	2	6
3	c	3	4
4	d	7	3
5	b	4	7
6	a	8	8
7	c	6	4
8	d	3	6
9	a	4	2

In [123]:
```
group = grp.groupby("Name")

for x,y in group:
  print(x)
  print(y)
```

```
a
  Name Sub_1 Sub_2
0    a     4     2
6    a     8     8
9    a     4     2
b
  Name Sub_1 Sub_2
1    b     6     7
5    b     4     7
c
  Name Sub_1 Sub_2
2    c     2     6
3    c     3     4
7    c     6     4
d
  Name Sub_1 Sub_2
4    d     7     3
8    d     3     6
```

In [124]:
```
# Showing a particular data
group.get_group('a')
```

Out[124]:

	Name	Sub_1	Sub_2
0	a	4	2
6	a	8	8
9	a	4	2

In [125]:
```
# Getting min and max data
group.min() # max() for maximum data
```

Out[125]:

Name	Sub_1	Sub_2
a	4	2
b	4	7
c	2	4
d	3	3

In [126]:
```
# Getting mean data
group.mean()
```

Out[126]:

Name	Sub_1	Sub_2
a	161.333333	94.000000
b	32.000000	38.500000
c	78.666667	214.666667
d	36.500000	18.000000

In [127]:
```
# Convert into a list
'''li = grp.to_list()
li'''
```

Out[127]:
```
'li = grp.to_list()\nli'
```

Pivot Table and Melt

Melt: The melt function will take the DataFrame and unpivot it, so that the columns that are not specified as identifier variables are moved to the row axis. This will leave just two non-identifier columns, 'variable' and 'value'

Syntax: pd.melt(varname)

Parameters of melt function:

1. **id_vars:** This parameter specifies the column(s) that should be used as the identifier variables. These columns will remain as they are in the resulting melted DataFrame.
2. **value_vars:** This parameter specifies the column(s) that should be melted into variable and value columns. If not specified, all columns not set as id_vars will be melted.
3. **var_name:** This parameter specifies the name of the variable column that will be created to store the column names of the melted variables.
4. **value_name:** This parameter specifies the name of the value column that will be created to store the values of the melted variables.
5. **col_level:** This parameter is used when the input DataFrame has multi-level columns. It specifies the level(s) to melt.
6. **ignore_index:** This parameter specifies whether to reset the index of the resulting melted DataFrame. If set to True, the resulting DataFrame will have a default integer index starting from 0.
7. **col_order:** This parameter allows you to specify the order of the resulting columns in the melted DataFrame.

There are a few more parameters available that can be useful in specific scenarios.

In [128]:
```
mel = pd.DataFrame({"Day":[1, 2, 3, 4], "Maths":[4, 6, 5, 8], "Comp":[4, 2, 5, 7]})
mel
```

Out[128]:

	Day	Maths	Comp
0	1	4	4
1	2	6	2
2	3	5	5
3	4	8	7

In [129]:
```
# Melt Function, we are vertically reshaping the table
pd.melt(mel)
```

Out[129]:

	variable	value
0	Day	1
1	Day	2
2	Day	3
3	Day	4
4	Maths	4
5	Maths	6
6	Maths	5
7	Maths	8
8	Comp	4
9	Comp	2
10	Comp	5
11	Comp	7

In [130]:
```
# Change the id of table and customize the headings
pd.melt(mel,id_vars=['Day'], var_name="Subjects", value_name='Marks')
```

Out[130]:

	Day	Subjects	Marks
0	1	Maths	4
1	2	Maths	6
2	3	Maths	5
3	4	Maths	8
4	1	Comp	4
5	2	Comp	2
6	3	Comp	5
7	4	Comp	7

Pivot: A pivot table is a data summarization tool that allows you to quickly and easily analyze data across multiple dimensions. It can be used to summarize data by category, group, or other variable. To create a pivot table in Pandas, you will need to use the pivot_table() function. The pivot_table() function takes several arguments, including the data frame, the index, the columns, and the aggregation function.

Syntax: var.pivot()

Parameters of Pivot Function:

1. **index:** This parameter specifies the column(s) to be used as the index in the resulting pivot table.
2. **columns:** This parameter specifies the column(s) to be used as the columns in the resulting pivot table.
3. **values:** This parameter specifies the column(s) to be used as the values in the resulting pivot table.
4. **aggfunc:** This parameter specifies the aggregation function to be applied to the values. The default is "mean", but other options include "sum", "count", "min", "max", etc.

5. **fill_value:** This parameter specifies the value to replace missing or NaN values with in the resulting pivot table.
6. **margins:** This parameter specifies whether to include totals or margins for each group in the resulting pivot table. By default, it is set to False.

There are also other optional parameters available, such as "columns_order" and "dropna", which can be used to further customize the pivot table.

In [131]:
```
pi = pd.DataFrame({"Day":[1, 2, 3, 4, 5],
                   "Student":['a','c','b','c','a'],
                   "Maths":[4, 6, 5, 8, 9],
                   "Comp":[4, 2, 5, 7, 8]})
pi
```

Out[131]:

	Day	Student	Maths	Comp
0	1	a	4	4
1	2	c	6	2
2	3	b	5	5
3	4	c	8	7
4	5	a	9	8

In [132]:
```
# Using pivot function
pi.pivot(index='Day', columns='Student')
```

Out[132]:

	Maths			Comp		
Student	**a**	**b**	**c**	**a**	**b**	**c**
Day						
1	4.0	NaN	NaN	4.0	NaN	NaN
2	NaN	NaN	6.0	NaN	NaN	2.0
3	NaN	5.0	NaN	NaN	5.0	NaN
4	NaN	NaN	8.0	NaN	NaN	7.0
5	9.0	NaN	NaN	8.0	NaN	NaN

In [133]:
```
# To get a single column value
pi.pivot(index='Day', columns='Student', values='Comp')
```

Out[133]:

Student	a	b	c
Day			
1	4.0	NaN	NaN
2	NaN	NaN	2.0
3	NaN	5.0	NaN
4	NaN	NaN	7.0
5	8.0	NaN	NaN

Pivot Table: To perform numerical operations

Syntax: var.pivot_table()

Parameters of PivotTable Functions:

1. **values:** Specifies the column to be aggregated.
2. **index:** Specifies the column(s) to be used as row index(es) in the resulting pivot table.
3. **columns:** Specifies the column(s) to be used as column index(es) in the resulting pivot table.
4. **aggfunc:** Specifies the aggregation function to be applied to the values. Examples include sum, mean, count, max, min, etc.
5. **fill_value:** Specifies the value to replace missing values with.
6. **margins:** Specifies whether to include row/column-wise subtotals and grand total in the resulting pivot table.
7. **margins_name:** Specifies the name to be used for row/column-wise subtotals and the grand total.
8. **dropna:** Specifies whether to exclude rows/columns with missing values from the resulting pivot table.
9. **observed:** Specifies whether to only include observed values of categorical factors.
10. **margins_name:** Specifies the name to be used for row/column-wise subtotals and the grand total.

In [134]:
```
table = pd.DataFrame({"Day":[1, 1, 1, 2, 2],
                      "Student":['a','b','b','a','a'],
                      "Maths":[4, 6, 5, 8, 9],
                      "Comp":[4, 2, 5, 7, 8]})
table
```

Out[134]:

	Day	Student	Maths	Comp
0	1	a	4	4
1	1	b	6	2
2	1	b	5	5
3	2	a	8	7
4	2	a	9	8

In [135]:
```
# Using pivot table function
table.pivot_table(index='Student', columns='Day', aggfunc='mean') # Same way in agg
```

Out[135]:

	Comp		Maths	
Day	**1**	**2**	**1**	**2**
Student				
a	4.0	7.5	4.0	8.5
b	3.5	NaN	5.5	NaN

In [136]:
```
# Get avg margin value
table.pivot_table(index='Student', columns='Day', aggfunc='sum', margins= True)
```

Out[136]:

	Comp			Maths		
Day	**1**	**2**	**All**	**1**	**2**	**All**
Student						
a	4.0	15.0	19	4.0	17.0	21
b	7.0	NaN	7	11.0	NaN	11
All	11.0	15.0	26	15.0	17.0	32