PANDAS :

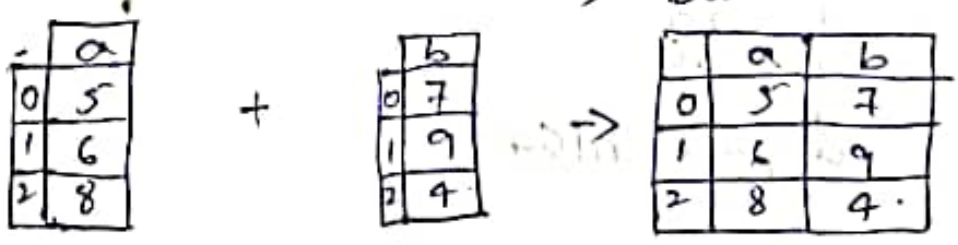
BOYA RATESH...

1. pandas is a python library used for working with data sets.
2. It has functions for analyzing, cleaning, exploring and manipulating data.
3. The name "Pandas" has reference to both "PANEL DATA" & "PYTHON DATA ANALYSIS"
4. It was created by "WES McKinney", in 2008.

FEATURE'S :

1. cleaning up data
2. Handling missing data
3. Alignment and indexing
4. Great handling of data.
5. Handling missing data.
6. python support.
7. Lot of time series
8. Optimized performance.
9. Grouping
10. Multiple file format's are supported.

→ Series + Series → Dataframe.



PANDAS OPERATIONS:

• Loading packages and Initialization

[ ]
```
import numpy as np
import pandas as pd.

labels = ['a', 'b', 'c']
my-data = [10, 20, 30]
arr = np.array(my_data)
d = {'a':10, 'b':20, 'c':30}.

Print("Dictonary:", d)
```

[ ] output:

Dictionary: {'a':10, 'b':20, 'c':30}

→ • CREATING A SERIES (Pandas class)

- -> From numerical data only
- -> From numerical data and correspondy index (row labels)
- -> From numpy array as the sowa of numerical data.
- -> just using a pre-defined dictionary.

[ ]
```
Pd.Series(data = my_data)
```

[ ] output:

```
0   10
1   20
2   30
dtype: int64
```

```
[] Pd.series(data=my-data, index=labels)
```

output:
```
a  10
b  20
c  30
dtype: int64
```

```
[] Pd.series(arr, labels)
```

output:
```
a  10
b  20
c  30
dtype: int 32
```

→ #using a pre-defined Direct Dictioncay object

```
[] Pd.Series(d)
```

output:
```
a  10
b  20
c  30
dtype: int64.
```

→ What type of values can a Pandas series hold?

```
[] print("\nHolding numerical data\n", '-'*25,
    sep='')
print(Pd.Series(arr))
```

output:

Holding numerical data

```
0  10
1  20
2  30
dtype: int32.
```

```
[ ] Print("\n Holding text labels \n", '_'*20, sep='')
    Print(pd.Series(labels))
```

```
[1] output: Holding text labels
            0  a
            1  b
            2  c
            dtype: Object.
```

```
[ ] Print("\n Holding functions \n", '_'*20, sep='')
    Print(pd.Series(data = [sum, print, len]))
```

```
[2] output:
        Holding functions:
        0    <built-in function sum>
        1    <built-in function print>
        2    <built-in function len>
        dtype: object.
```

```
[ ] Print("\n Holding objects from a dictionary \n",
        '_'*40, sep='')
    print(pd.Series(data= [d.keys, d.items, d.values]))
```

```
[3] output:
    Holding objects from a dictionary
    0  <built-in method keys of dict object
       at 0X09A---
    1  <built-in method items of dict
       Object at 0X09A----.
    2  <built-in method values of dict
       Object at 0X09----.
    dtype: Object.
```

INDEXING AND SLICING:

```
[] ser1 = Pd.Series([1,2,3,4],['CA','OR','CO','AZ'])
   ser2 = Pd.Series([1,2,5,4],['CA','OR','NV','AZ'])

   print("\nIndexing by name of the item/object
         (String identifier)\n",'-'*50,sep="")
   print("Value of CA in ser1:",ser1['CA'])
   print("Value of AZ in ser1:",ser1['AZ'])
   Print("Value of NV in ser2:",ser2['NV'])

   Print("\nIndexing by number (Positional value in the
         series)\n",'-'*52,sep='')
   print("Value for CA in ser1:",ser1[0])
   Print("Value for AZ in ser1:",ser1[3])
   print("Value of NV in ser2:",ser2[2])
   Print("\nIndexing by a range\n",'-'*25,sep='')
   print("Value for OR,CO,& AZ in ser1:\n",ser1[1:4,
                                              sep='')
```

[] output:

Indexing by name of the item/object (String identifier):
Value of CA in ser1: 1
Value of AZ in ser1: 4
Value of NV in ser2: 5

Indexing by number (Positional value in series):
Value for CA in ser1: 1
Value for AZ in ser1: 4
Value for NV in ser2: 5

Indexing by a range:
Value for OR, CO, and AZ in ser1

```
OR  2
CO  3
AZ  4
dtype: int64.
```

ADDING/MERGING TWO SERIES WITH COMMON INDICES:

```
Ser1 = Pd.Series([1,2,3,4], ["CA", 'OR', 'CO', 'AZ'])
Ser2 = Pd.Series([1,2,5,4], ['CA', 'OR', 'NV', 'AZ'])
Ser3 = ser1 + ser2

Print("\nAfter adding the two series, the result
      looks like this...\n", '-'*59, sep='')
Print(Ser3)
```

Output:

After adding the two series, the result looks like this....

```
AZ    8.0
CA    2.0
CO    NaN
NV    NaN
OR    4.0
dtype: float64
```

```
print("\n python tries to add values where it
finds common index name, and puts NaN
where indices are missing\n")
print("\n The idea works even for multiplication")
print(ser1 * ser2)
```

Output:

```
AZ    16.0
CA     1.0
CO     NaN
NV     NaN
OR     4.0
dtype: float64
```

[ ] print("\nOr even for combination of mathematical operations!\n", '_'*45, sep='')
print(np.exp(ser1) + np.log10(ser2)).

[ ] Or even for combination of Mathematical Operations!

```
AZ    55.200210
CA     2.710282
CO          NaN
NV          NaN
OR     7.600006
dtype: float64.
```

⇔ DATAFRAME (THE REAL MEAT!)

[ ] from numpy.random import randn as rn.

CREATING AND ACCESSING DATAFRAME

- Indexing
- Adding and deleting rows & columns.
- Subsetting Dataframe.

[ ] np.random.seed(101)
matrix_data = rn(5,4)
row_labels = ['A','B','C','D','E']
column_headings = ['W','X','Y','Z']
df = pd.DataFrame(data=matrix_data, index=row_labels, columns=column_headings)
print("\nTHE data frame looks like\n", '-'*45, sep='')
print(df)

[ ] Output: The data frame looks like:

	W	X	Y	Z
A	2.7	5.8	9.9	8.0
B	2.4	10.4	2.2	9.7
C	5.6	3.5	3.3	8.8
D	7.6	4.3	9.4	9.7
E	8.9	9.3	5.6	10.3

INDEXING AND SLICING (Columns)

- By bracket method :
- By Dot method (Not recommended).

```
[ ] print("\n The 'x' column \n", '-'*25, sep='')
    print(df['x'])
```

[ ] output:

```
 The 'x' column :
-------------------------
A  0.628133
B -0.319318
C  0.740122
D -0.758872
E  1.978757  Name: x, dtype: float64
```

```
[ ] print("\n Type of the column : ",
        type(df['x']), sep=' ')
```

[ ] output:

```
Type of the column :
        <class 'pandas.core.series.Series'>
```

```
[ ] print("\n The 'x' and 'z' columns
    indexed by passing a list \n", '-'*55)
    print(df[['x','z']])
```

[ ]

```
The 'x' and 'z' columns indexed by
passing a list :
-------------------------------------------
          x         z
A -0.628133  0.503826
B -0.319318  0.605765
C  0.740122 -0.955057
D -0.758872  0.590001
E  1.978757  0.683502
```

```
[] print("\nType of the pair of columns: ",
        type(df[['X','Z']]), sep=' ')
```

```
[] Type of the pair of columns: <class 'pandas.
                core.frame.DataFrame'>
```

=> For more than one column, the object turns into a dataframe.

```
[] print("\nThe 'X' column accessed by Dot
   method (NOT recommend)\n", '-' * 55, sep='')
   print(df.X)
```

```
[] The 'X' column accessed by Dot method
   ----------------------------------------

   0   A   0.620133
       B  -0.319318
       C   0.740122
       D  -0.758872
       E   1.978757

   Name: X,
   dtype: float64
```

CREATING and Deleting a column or row:

```
[ ] Print("\n-A column is created by
    alligning it in relation to an existing
    column \n", '-' *75, sep="-")
```

```
[ ] print
    df['New'] = df['x']+df['z']
    Print(
    df['New (sum of x and z)'] = df['x']+df['z']
    Print(df)
```

[ ] A column is created by alligning it in relation to an existing column:

	W	X	Y	Z	New	New (sum of x and z)
A	2.7	0.6	0.1	0.5	1.13	1.13
B	0.6	-0.3	-0.8	0.6	0.28	0.28
C	-2.0	0.7	0.5	0.1	0.15	0.15
D	0.1	-0.7	-0.3	0.9	0.19	0.19
E	0.1	1.9	2.6	0.6	2.66	2.65

```
[ ] print("\n A column is dropped by using
    df.drop() method \n", '-' *35, sep='')
    df = df.drop('New', axis=1)
    #Notice the axis=1 option, axis=0 is
    default, so one has to change it to 1
    print(df)
```

[ ] A column is dropped by alligning drop():

	W	X	Y	Z	New (sum of x & z)
A	2.70	0.62	0.90	0.50	1.13

B	0.65	-0.31	-0.04	0.60	0.28
C	-2.01	0.740	0.90	-0.50	0.15
D	0.18	-0.75	-0.84	0.95	0.10
E	0.190	1.978	2.60	0.68	2.66.

```
[] df1 = df.drop('A')
   Print("\n An in place change can be done.
   by making inplace
   Print("\n A row(index) is dropped by using
    df.drop() method and axis=0 \n", '-'*0.5)
   print(df1).
```

[1] output:

A row(index) is dropped by using dropped df.drop() method and axis=0:

	W	X	Y	Z	New(sum of x and z)
B	0.65	-0.31	-0.04	0.60	0.28
C	-2.01	0.740	0.90	-0.50	0.15
D	0.18	0.75	0.84	0.95	0.10
G	0.190	1.978	2.60	0.65	2.66.

```
[] Print("\n An in place change can be done by making
   inplace - True in the drop method \n")
   df.drop('New(sum of x and z)', axis=1, inplace=
   print(df).                                  True)
```

[] output: An in place change can be done by making New(sum of x and z) in place - True in drop method:

	W	X	Y	Z
A	2.70	0.62	0.90	0.50
B	0.65	-0.31	0.90	0.66
C	-2.01	0.74	-0.84	-0.50
D	0.18	-0.75	0.90	0.95
E	0.190	1.97	2.60	0.68

Selecting/ Indexing Rows...

- Label-based 'loc' method.
- Index(numeric) 'iloc' method.

```
[ ] Print("In Label-based 'loc' method can be used for selecting row(s) \n", '-'*60)
    Print("In single row \n")
    Print( df.loc['c'])
```

[o] Output:

Label-based 'loc' method can be used for selecting row(s):

Single row:

```
W   -2.010
X    0.790
Y    0.52
Z   -0.58
Name: C, dtype: float64
```

```
[ ] Print("\n Multiple rows \n")
    Print (df.loc[['B', 'C']])
```

[o] Multiple rows:

	W	X	Y	Z
B	0.651	-0.31	-0.8	0.60
C	-2.018	0.74	0.52	-0.50

```
[ ] Print("\n Index position based 'iloc' method can be used for selecting row(s)", '-'*20)
    print("\n Single row \n")
    print (df.iloc[2])
```

[3] output:

Index position based 'iloc' method can be used for selecting row(s):

Single row:

```
W   -2.01
X    0.74
Y    0.52
Z   -0.58
Name: C, dtype: float64
```

[ ] print("\n Multiple rows\n").
print(df.iloc[[1,2]])

[3] output:

Multiple rows:

	W	X	Y	Z
B	0.6511	-0.319	-0.84	0.60
C	-2.0101	0.74	0.52	-0.50

SUBSETTING Dataframe...

```
[ ] print("\n The Dataframe\n",'-'*45, sep='')
    print(df)
```

[ ] output:

The Dataframe

	W	X	Y	Z
A	2.706	0.6281	0.907	0.503
B	0.651	-0.319	-0.848	0.605
C	-2.010	0.740	0.528	-0.589
D	0.181	-0.758	-0.933	0.955
E	0.190	1.978	2.605	0.682

```
[ ] print("\n Element at row 'B' and colomn 'Y' is \n")
    print(df.loc['B','Y'])
```

[ ] output:

Element at row 'B' and colomn 'Y' is -0.848076

```
[ ] print("\n Subset comparing of rows B & D, and columns W and Y, is \n")
    print(df.loc[['B','D'],['W','Y']])
```

[ ] output:

Subset comparing of rows B and D, and columns W and Y, is:

	W	Y
B	0.651	-0.840
C	0.188	-0.933

CONDITIONAL SELETION, INDEX RESETTING, MULTI-INDEX:

- Basic idea of conditional check and Boolean DataFrame

```
[] print("\n Boolean DataFrame where we are checking if the values are greater than 0\n",'-'*75, sep='\n')
   print(df>0).
```

[] Output:

Boolean Dataframe where we are checking if the values are greater than 0:

	W	X	Y	Z
A	True	True	True	True
B	True	False	False	True
C	False	True	True	False
D	True	False	False	True
E	True	True	True	True.

```
[] print(df.loc[['A','B','C']]>0).
```

[] Output:

	W	X	Y	Z
A	True	True	True	True
B	True	False	False	True
C	False	True	True	False.

```
[] boolDf = df>0
   print("\n\n DataFrame indexed by boolean dataframe\n",'-'*45, sep='\n')
   print(df[boolDf]).
```

[] Output: Dataframe indexed by boolean dataframe:

	W	X	Y	Z
A	2.7068	0.629	0.9079	0.501
B	0.65111	NaN	NaN	0.60
C	NaN	0.740	0.5298	NaN
D	0.18869	NaN	NaN	0.955
E	0.19078	1.978	2.605	0.693.

PASSING BOOLEAN SERIES TO CONDITIONALLY SUBSET THE DATAFRAME:

```
[ ] matrix_data = np.matrix('22,60,140;
                             42,70,148;
                             30,62,125;
                             15,68,160;
                             25,62,152')
    row_labels = ['A', 'B', 'C', 'D', 'E']
    column_headings = ['Age', 'Height', 'Weight']

    df = pd.DataFrame(data=matrix_data, index=row_
                      labels, columns=column_headings)
    print("\n The dataframe \n", "*"*25, sep='\n')
    print(df)
```

[ ] output:

A new Dataframe.

	Age	Height	weight
A	22	60	40
B	42	70	48
C	30	62	125
D	15	68	160
E	25	62	152

```
[ ] print("\n Rows with height > 65 inch \n",
          '-'*35, sep='\n')
    print(df[df['Height']>65])
```

[ ] output: Rows with height > 65 inch

	Age	Height	weight
A	22	66	140
B	42	70	148
D	35	68	160

```
[ ] boolDf1 = df['Height'] >65
    boolDf2 = df['Weight']>145
    Print("\n Rows with height > 65 inch &
    weight >145 lb", '-'*50, sep='\n')
    Print(df[(boolDf1) & (boolDf2)]).
```

[ ] Output:

Rows with Height >65 inch and weight >145 :

	Age	Height	weight
B	42	70	148
D	35	68	160

```
[ ] Print('\n Dataframe with only Age and
    weight column whose Height >65 inch \n',
    '-'*60, sep='\n')
    print(df[boolDf1][['Age','weight']])
```

[ ] Output:

Dataframe with only Age and weight Column whose Height >65 inch :

	Age	weight
A	22	160
B	42	140
D	35	160

RE-SETTING AND SETTING INDEX :

```
[ ]  matrix_data = np.matrix ('22,66,140;
                               42,70,148;
                               30,62,125;
                               35,68,160;
                               25,62,152')
     row_labels = ['A','B','C','D','E']
     Column_headings = ['Age','Height','weight']
     df = pd.DataFrame(data=matrix_data,
        index=row_labels, columns=Column_heading)
     print("\n After reseting index \n", '-'*25)
     print(df.reset_index())
```

[ ] output:

After resetting index

	index	Age	Height	weight
0	A	22	66	140
1	B	42	70	148
2	C	30	62	125
3	D	35	68	160
4	E	25	62	152

```
[ ]  print('\n After reseting index with
     'drop' option TRUE \n', '-'*45, sep='')
     print(df.reset_index(drop=True))
```

[ ] output:

After reseting index with 'drop' option true:

	Age	Height	weight
0	22	66	140
1	42	70	148

2	30	62	125
3	35	60	160
4	25	62	152

```
print("\n Adding a new column 'profession'
\n", '-'*45, sep='')
df['Profession'] = 'Student Teacher Engineer
                    Doctor Nurse'.split()
print(df)
```

output:

Adding a new column 'profession'

	Age	Height	weight	profession
A	22	66	140	Student
B	42	70	148	Teacher
C	30	62	125	engineer
D	35	68	160	Doctor
E	25	62	152	Nurse

```
print("\n Setting 'profession' column as
    index \n", '-'*45, sep='')
print(df.set_index('Profession'))
```

output:

Setting 'profession' column as index:

Profession	Age	Height	weight
Student	22	66	140
Teacher	42	70	148
engineer	30	62	125
Doctor	35	68	160
Nurse	25	62	152

MULTI-INDEXING :

```
[ ] #Index level.
outside = ['G1', 'G1', 'G1', 'G2', 'G2', 'G2']
inside = [1, 2, 3, 1, 2, 3]
hier_index = list(zip(outside, inside))
print("\n Tuple pairs after the zip and list
      Command \n", '-' * 45, sep='')
print(hier_index).
```

[o] output:

Tuple pairs after the zip and list command:

```
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2),
 ('G2', 3)]
```

```
[ ] hier_index = pd.MultiIndex.from_tuples(hier_index)
print("\n Index hierarchy \n", '-' * 25, sep='')
print(hier_index)
```

[I] Index HIERARCHY :

```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],)
```

```
[ ] print("\n Index hireracy type \n", '-' * 37)
print(type(hier_index))
```

[I] output:

Index hireracy type :

```
<class 'pandas.core.indexes.multi.
                        MultiIndex'>
```

```
print("\n Creating Dataframe with multi-
index \n", '-'*37, sep='')
np.random.seed(101)
df1 = pd.DataFrame(data= np.round(rn(6,3),2),
                   index = hier_index, columns=['A','B','C'])
print(df1)
```

output: creating Dataframe with multi-index

		A	B	C
G1	1	2.71	0.53	0.91
	2	0.50	0.65	-0.32
	3	-0.05	0.61	-2.02
G2	1	0.79	0.53	-0.59
	2	0.19	-0.76	-0.95
	3	0.96	0.19	1.98

```
print("\n Subsetting multi-Index Dataframe using
      two 'loc' methods \n", '-'*60, sep='')
print(df1.loc['G2'].loc[[1,3]][['B','C']])
```

output: Subsetting multi-Index Dataframe using two 'loc' methods:

	B	C
1	0.53	-0.59
3	0.19	1.98

```
print("\n Naming the indices by 'index.names'
  method \n", '-'*45, sep='')
df1.index.names=['Outer', 'Inner']
print(df1)
```

output: Naming the indices by 'index.names' method:

Outer	Inner	A	B	C
G1	1	2.71	0.53	0.91
	2	0.50	0.65	-0.32
	3	-0.05	0.61	-2.02
G2	1	0.79	0.53	-0.59
	2	0.19	-0.76	-0.95
	3	0.96	0.19	1.98

CROSS-SECTION ('XS') COMMAND:

[ ] Print("In creating a cross-section from outer level:\n", df1.xs('G1'))

[ ] output:

Creating a cross-section from outer level:

	A	B	C
Inner			
1	2.71	0.63	0.91
2	0.56	0.65	-0.32
3	-0.85	0.61	-2.02

[ ] Print("\nGrabbing a cross-section from inner level (for all outer level):\n")
print(df1.xs(2, level = "Inner"))

[ ] output:

Grabbing a cross-section from inner level (for all outer level):

	A	B	C
outer			
G1	0.56	0.65	-0.32
G2	0.19	-0.76	-0.93

MISSING VALUES

```
[ ] df = pd.DataFrame({'A': [1, 2, np.nan],
                       'B': [5, np.nan, np.nan],
                       'C': [1, 2, 3]})
df['Status'] = "CA NV AZ".split()
df.set_index('Status', inplace=True)
print(df)
```

[ ] OUTPUT :

Status	A	B	C
CA	1.0	5.0	1
NV	2.0	NaN	2
AZ	NaN	NaN	3

Pandas 'dropna' method:

```
print("\n Dropping any rows with a NaN value:\n",
      '-' *35, sep='')
print(df.dropna(axis=0))
```

output:

Dropping any rows with a NaN value:

	A	B	C
States			
CA	1.0	5.0	1

```
print("\n Dropping any column with a NaN value:\n",
      '-' *35, sep='')
print(df.dropna(axis=1))
```

output:

Dropping any column with a NaN value:

	C
States	
CA	1
NV	2
AZ	3

```
print("\n Dropping a row with a minimum 2
      NaN value using 'thresh' parameter:\n",
      '-' *68, sep='')
print(df.dropna(axis=0, thresh=2))
```

Dropping a row with a minimum 2 NaN value using 'thresh' parameter:

	A	B	C
States			
CA	1.0	5.0	1
NV	2.0	NAN	2

PANDA'S "FILLNA" METHOD:

```
print("\n Filling Values with a default
    Value \n", '-'*55, sep='')
print(df.fillna(value='FILL VALUE'))
```

output:

```
Filling Values with a default value:
----------
```

	A	B	C
State			
CA	1	5	1
NV	2	FILL VALUE	2
AZ	FILL VALUE	FILL VALUE	3

```
print("\nFilling Values with a computed value
    (mean of column A here)\n",'-'*60,sep='')
print(df.fillna(value=df['A'].mean()))
```

output:

```
Filling Values with a computed value (mean of
    column A here):
----------
```

	A	B	C
State			
CA	1.0	5.0	1
NV	2.0	1.5	2
AZ	1.5	1.5	3

GroupBy Method :

[ ]
```
#creating a dataframe
data = { 'company' : ['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],
         'person' : ['sam', 'charlie', 'Amy', 'vanessa', 'carl', 'sarah'],
         'sales' : [200, 120, 340, 124, 243, 350] }.
df = pd.DataFrame(data)
print(df).
```

[1]

	company	person	sales
0	GOOG	sam	200
1	GOOG	charle	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	carl	243
5	FB	sarah	350

[ ]
```
bycomp = df.groupby('company')
print("\n Grouping by 'company' column and
      listing mean sales in", '-'*55, sep='')
print(bycomp.mean())
```

[ ] output:

Grouping by 'company' column & listing mean sales:

company	sales
FB	296.5
GOOG	160.0
MSFT	232.0

```
print("\n Grouping by 'company' column and
      listing sum of Sales \n", '-'*55, sep='')
print(bycomp.sum())
```

output:

Grouping by 'company' column and listing sum of.

company	Sales
FB	593
GooG	320
MSFT	464

```
print("\n All in one line of command (stats for 'FB')
      \n", '-'*67, sep='') #transpose
print(pd.DataFrame(df.groupby('Company').describe().
      loc['FB']).transpose())
```

output: All in one line of command (stats for 'FB'):

	Sales							
	count	mean	std	min	25%	50%	75%	max
FB	2.0	296.5	75.66	243.0	269.75	296.5	323.25	350.0

```
print("\n Same type of extraction with little different
      command \n", '-'*68, sep='')
print(df.groupby('Company').describe().loc[['GooG',
      'MSFT']])
```

output:

Same type of extraction with little different command:

	Sales							
	count	mean	std	min	25%	50%	75%	max
Company								
GooG	2.0	160	84.5	120.0	100.6	160.0	160.0	200.0
MSFT	2.0	232.0	137.73	129.0	178.0	232.0	286.0	340.0

MERGING, JOINING, CONCATENATING

concatenation:

```
[] #creating data frames
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])

[] df_cat1 = pd.concat([df1, df2, df3], axis=0)
print("\n After concatenation along row is",
      '-' * 30, sep='\n')
print(df_cat1)
```

[1] After concatenation along row:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

[] df_cat2 = pd.concat([df1, df2, df3], axis=1)
print("After concatenation along column\n", '-'*20)
print(df_cat2).

[1] output: After concatenation along column:

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

[ ]
```
print ("\nAfter filling missing values with
      zero\n", '-'*60, sep='')
print(df_cat2)
```

[ ]
```
df_cat.fillna(value=0, inplace=0).
```

Output:

After filling missing values with zero:

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	0	0	0	0	0	0	0	0
1	A1	B1	C1	D1	0	0	0	0	0	0	0	0
2	A2	B2	C2	D2	0	0	0	0	0	0	0	0
3	A3	B3	C3	D3	0	0	0	0	0	0	0	0
4	0	0	0	0	A4	B4	C4	D4	0	0	0	0
5	0	0	0	0	A5	B5	C5	D5	0	0	0	0
6	0	0	0	0	A6	B6	C6	D6	0	0	0	0
7	0	0	0	0	A7	B7	C7	D7	0	0	0	0
8	0	0	0	0	0	0	0	0	A8	B8	C8	D8
9	0	0	0	0	0	0	0	0	A9	B9	C9	D9
10	0	0	0	0	0	0	0	0	A10	B10	C10	D10
11	0	0	0	0	0	0	0	0	A11	B11	C11	D11

Merging by a common 'key':

⇒ The merge function allows you to merge Data frames together using a similar logic as merging SQL tables together.

```
left = pd.DataFrame({'key': ['C0', 'C1', 'C2', 'C3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
```

```
print("\n The Dataframe 'left' \n", '-'*30, sep='')
print(left)
```

Output:

The Dataframe 'left'

---

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

```
print("\n The Dataframe 'right' \n", '-'*30, sep='')
print(right)
```

The Dataframe 'right'

---

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

[ ]
```
merge1 = pd.merge(left, right, how='inner', on='Key')
print('\n After simple merging with 'inner'
    method \n", '-'*50, sep='')
print(merge1)
```

[ ] Output:

After simple merging with 'inner' method

	Key	A	B	C	D
0	K0	A0	B0	C0	D1
1	K1	A1	B1	C1	D2
2	K2	A2	B2	C2	D3
3	K3	A3	B3	C3	D4

⇒ JOINING:

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single DataFrame based on 'index keys'.

[ ]
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                     index = ['K0', 'K1', 'K2'])
right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                      index = ['K0', 'K2', 'K3'])
```

```
print(left)
```

[ ]

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

[ ] print(right)

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

[] left.join(right)

[]

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2.

[] left.join(right, how='outer')

[]

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

5.

⇒ USEFUL OPERATIONS:

head() and Unique Values

- head()
- unique()
- nunique()
- value_count().

[] import pandas as pd

```
df = pd.DataFrame({'Col1': [1,2,3,4,5],
                   'Col2': [444,555,666,444,333],
                   'Col3': 'aaa bb c dd eeee'.
df                                          split()})
```

[]

	Col1	Col2	Col3
0	1	444	aaa
1	2	555	bb
2	3	ccc	c
3	4	444	dd
4	5	333	eeee.

[ ]
```
print("\n Method head() is for showing
    first few entries\n", '-' * 50)
df.head()
```

[ ]
Method head() is for showing first few entries:

	col1	col2	col3
0	1	444	aaa
1	2	555	bb
2	3	ccc	c
3	4	444	dd
4	5	333	cccc

[ ]
```
print("\n Finding Unique values in 'col2'\n",
    '-' * 40)
print(df['col2'].unique())
```

[ ]
Finding unique values in col2:
[444 555 666 333 222 777]

[ ]
```
print("\nfinding number of unique values
in 'col2'\n", '-'*40, sep='')
tl = df['col2'].nunique()
```

[ ]
Finding no. of unique values in 'col2':
6

[ ]
```
print("\nTable of unique values in
    'col2'\n", '-' * 40, sep='')
tt = df['col2'].value_counts()
print(tt)
```

[ ] Table of unique values in 'col2':

```
666    3
444    2
555    2
022    1
333    1
777    1
Name: col2, dtype : int64
```

APPLYING FUNCTIONS:

→ Pandas work with 'apply' method in accept any user-Defined function.

[ ]
```
# Define a function.
def testfunc(x):
    if x>500:
        return (10*np.log10(x))
    else:
        return(x/10)
```

[ ]
```
df['funcApplied']= df['col2'].apply(testfunc)
print(df).
```

	col1	col2	col3	funcApplied
0				44.40
1	1	444	aaa	27.99
2	2	555	bb	28.32
3	3	666	c	44.40
4	4	444	dd	33.30
5	5	333	eeee	22.20
6	6	222	fff	27.70
7	7	666	gg	28.23
8	8	777	h	27.67
9	9	666	iii	
	10	555	j	

Rajesh Boya