Getting Started with pandas:

Introduction to pandas, Library Architecture, Features, Applications, Data Structures, Series, DataFrame, Index Objects, Essential Functionality Reindexing, Dropping entries from an axis, Indexing, selection, and filtering),Sorting and ranking, Summarizing and Computing Descriptive Statistics, Unique Values, Value Counts, Handling Missing Data, filtering out missing data.

Introduction for Pandas:

- "Pandas" is developed by **WES MCKINNEY** in 2008 and is used for data analysis. As for data analysis requires lots of processing like restructing, cleaning, etc. So we use pandas.
- In python, pandas is defined as an open source library which is used for high performance data manipulation and high level data structure.
- It contains high level data structures and manipulation tools ,designed for fast and easy data analysis in python.
- Pandas was built on top of numpy and makes it easy and more effective use for numpy centri application.

Introduction to Pandas Data Structure:

To get started with pandas, you need to be comfortable with two data structure.

1. Series
2. Data Frames

Series:

It is a one dimensional array, like object containing an "array of data" and it's associated with "array of data labels". The data labels are also called index. The given example is best way to create series.

Ex:
```
import pandas as pd

s=pd.Series([-2,5,7,9,23])

print(s)
```

output:

```
0    -2
1     5
2     7
3     9
4    23
dtype: int64
```

We can also return the index and values of the series. While using the series to create the series we should give initial “s” as upper case.

```
Print(s.index)
```

```
Output: RangeIndex(start=0, stop=5, step=1)
```

```
print (s.values)
```

```
Output: array([-2,  5,  7,  9, 23], dtype=int64)
```

Often it will be desirable to create a Series with an index identifying each data point :

```
import pandas as pd
s=pd.Series ([-2, 5, 7, 9, 23], index= ['a','b','c','d','e'])
print(s)
```

```
output:
a   -2
b    5
c    7
d    9
e   23
dtype: int64
```

Now we can the value of any index value:

```
print(s['d'])
print(s[‘a’,’b’,’c’])
output: 9
a   -2
b    5
c    7
dtype: int64
```

Operator on pandas:

We can also perform all the arithmetic and comparison operation in this array. We can perform operation like scalar multiplication, or applying math functions.

Ex:

```
import pandas as pd
k=pd.Series([-7,6,5,-21,5,65])
print(k)
output:
0   -7
1    6
2    5
3  -21
```

```
4     5
5    65
dtype: int64
print(k*k)
output:
0      49
1      36
2      25
3     441
4      25
5    4225
dtype: int64
```

```
print(k*2)
output:
0    -14
1     12
2     10
3    -42
4     10
5    130
dtype: int64
```

```
print(4 in k)
output:
True
Print(7 in k)
Output:
False
```

Working with directories:
Pandas support the directories directly without converting (or) rewriting in series.
Ex:

```
data= {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
Obj=pd.Series(data)
Print(Obj)
Output:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

We can give your own indexing to the series. In pandas can check if the given values are null or not null
Ex:

```
data={'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
place=['a','Ohio','Texas','Oregon','Utah','b']
obj=pd.Series(data,index=place)
```

```
print(obj)


output:
a           NaN
Ohio      35000.0
Texas     71000.0
Oregon    16000.0
Utah       5000.0
b           NaN
dtype: float64

a=pd.isnull(obj)
print(a)
output:
a         True
Ohio      False
Texas     False
Oregon    False
Utah      False
b         True
dtype: bool

b=pd.notnull(obj)
print(b)
output:
a        False
Ohio       True
Texas      True
Oregon     True
Utah       True
b        False
dtype: bool
```

Data Frames:

- A Data Frame represents a tabular, spreadsheet-like data structure containing an collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.). The Data Frame has both a row and column index; it can be thought of as a dict of Series.
- The data frame has both rows and column index. The data is stored as two or more dimensional blocks rather than list, directories, ndarrays or some other collections of one dimensional arrays.
- Data Frames stores data internally in two dimensional format and we can easily represent much high dimensional data in tabular format with hierarchical indexing.

```
import pandas as pd
data={‘Year’:[2021,2020,2019,2018],’Month’:[‘Jan’,’Feb’,’Mar’,’Apr’]}
df=pd.DataFrame(data)
print(df)
output:
   Year  Month
0  2021    Jan
1  2020    Feb
2  2019    Mar
3  2018    Apr
```

- There are number of ways to construct data frames, one of the most common form is dictionary.

```
data={'Year':[2021,2020,2019,2018],'Month':['Jan','Feb','Mar','Apr'],'day':['Mon','Tue','Wed','Thu']}
df=pd.DataFrame (data,columns=['Month','Day','Year'])
print(df)
```

output:

```
   Month  Day  Year
0    Jan  NaN  2021
1    Feb  NaN  2020
2    Mar  NaN  2019
3    Apr  NaN  2018
```

- Whatever the output order it may be we can get our specified order of output for column.
- Same as series we can change the default index by adding index in the DataFrame().

```
data={'Year':[2021,2020,2019,2018],'Month':['Jan','Feb','Mar','Apr'],'day':['Mon','Tue','Wed','Thu']}
df=pd.DataFrame(data,columns=['Month','Day','Year'],index=['a','b','c','d'])
print(df)
```

output:

```
   Month  Day  Year
a    Jan  NaN  2021
```

	Month	Day	Year
b	Feb	NaN	2020
c	Mar	NaN	2019
d	Apr	NaN	2018

- If you want to alter all the values of particular column with a unique value to the following set.
  ```
  import pandas as pd
  k=pd.DataFrame({'R.no':[4401,4402,4403,4404],'Sname':['ABC','DEF','GHI','JKL'],'Rank':[7,6,3,9])
  k[‘Rank’]=2
  print(k)
  output:
  ```

	R.no	Sname	Rank
0	4401	ABC	7
1	4402	DEF	6
2	4403	GHI	3
3	4404	JKL	9

```
import pandas as pd
k=pd.DataFrame({'R.no':[4401,4402,4403,4404],'Sname':['ABC','DEF','GHI','JKL'],'Rank':[7,6,3,9]})
k['Rank']=2
print(k)
output:
```

	R.no	Sname	Rank
0	4401	ABC	2
1	4402	DEF	2
2	4403	GHI	2

	R.no	Sname	Rank
3	4404	JKL	2

- If you want give arange of values to all the record of a particular column. To the above set.
  import numpy as np
  import pandas as pd
  k=pd.DataFrame({'R.no':[4401,4402,4403,4404],'Sname':['ABC','DEF','GHI','JKL'],'Rank':[7,6,3,9]}
  k['Rank']=np.arange(4)
  print(k)
  output:

	R.no	Sname	Rank
0	4401	ABC	0
1	4402	DEF	1
2	4403	GHI	2
3	4404	JKL	3

- Pandas support working of data frames with series data structure.

import numpy as np
import pandas as pd
k=pd.DataFrame({'R.no':[4401,4402,4403,4404],'Sname':['ABC','DEF','GHI','JKL'],'Rank':[7,6,3,9]})
val=pd.Series([11,22],index=[0,3])
k['Rank']=val
print(k)
output:

	R.no	Sname	Rank
0	4401	ABC	11.0
1	4402	DEF	NaN
2	4403	GHI	NaN
3	4404	JKL	22.0

- Possible data input to data frame constructor.
    1. 2D ndarray
    2. List,Tuple, Dictionary of arrays
    3. Numpy structured array
    4. dict(Series)
    5. dict(dicts)
    6. List(dict)
    7. List(Series)
    8. List(Lists)
    9. List(Tuple)
    10. Another data frame
    11. Numpy array

Working with altering index:
We can alter the index as whatever we want.

```
import pandas as pd
a=pd.Series([1,2,3],index=['a','b','c'])
print(a)
output:
a    1
b    2
c    3
dtype: int64
```

```
import pandas as pd
a=pd.Series([1,2,3],index=['a','b','c'])
a.index=['x','y','z']
print(a)
output:
x    1
y    2
z    3
dtype: int64
```

In this concept while working with altering index we cannot change the vales in the series.

```
a.index['x']=5
output:
Index does not support mutable operations.
```

Index object:

- Pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index
- Index objects are immutable i.e. that can be modified by the user.
- Immutability is important so that index objects can be safely shared among different dataframes and also in application.

Built-in Index Object in pandas library:

1. Index
2. int 64 index
3. Multi index
4. Data-Time index
5. Period index

Essential Functionality:

i. Reindexing
ii. Dropping entries from an axis
iii. Indexing, Selection and Filtering

Reindexing:

- Reindexing is a critical method in pandas object.
- The reindexing can be done with the help of reindex object. (it is a built in object)
- This object creates a new object with the data conformed to a new index.

```
import pandas as pd
obj=pd.DataFrame([1,2,3,4],index=['b','c','a','d'])
print(obj)
output:
```

```
   0
b  1
c  2
a  3
d  4
```

```
obj1=obj.reindex(['a','b','c','d','e','f'])
print (obj1)
output:
```

```
     0
a  3.0
b  1.0
c  2.0
d  4.0
```

	0
e	NaN
f	NaN

- fill_value is an attribute for reindex method that which fills a default value to zero to newly created index.
  obj2=obj.reindex(['a','b','c','d','e','f'],fill_value=0)
  print(obj2)
  output:

	0
a	3
b	1
c	2
d	4
e	0
f	0

- For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The method option allows us to do this, using a method such as ffill which forward fills the values.
  import pandas as pd
  obj= pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
  obj1=obj.reindex (range (6), method='ffill')
  print(obj1)
  output:

```
0     blue
1     blue
2   purple
3   purple
4   yellow
5   yellow
dtype: object
```

- The method option allows us to do this, using a method such as bfill which backward fills the values.

```
            import pandas as pd
            obj= pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
            obj2=obj.reindex(range(6),method='bfill')
            print(obj2)
            output:
0     blue
1   purple
2   purple
3   yellow
4   yellow
5      NaN
dtype: object
```

Dropping entries from an axis:
Dropping one or more entries from an axis is easy if you have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis:

```
import numpy as np
import pandas as pd
a=pd.Series(np.arange(5),index=['a','b','c','d','e'])
print(a)
output:
a    0
b    1
c    2
d    3
e    4
dtype: int32

b=a. drop ('c')
print (b)
Output:
a    0
b    1
d    3
e    4
dtype: int32

c=a. drop (['b','d'])
print(c)
Output:
a    0
c    2
e    4
dtype: int32
```

In data frame also we can drop the entries from an axis.

```
data =pd.DataFrame(np.arange(16).reshape((4, 4)),index=['Ohio', 'Colorado', 'Utah', 'New
York'],columns=['one', 'two', 'three', 'four'])
print(data)
output:
```

```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
b=data.drop(['Colorado', 'Ohio'])
print(b)
output:
```

```
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

```
df1=data.drop('two',axis=1)
print(df1)
output:
```

```
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

```
df2=data.drop(['three','one'],axis=1)
print(df2)
output:
```

	two	four
Ohio	1	3
Colorado	5	7
Utah	9	11
New York	13	15

Indexing, selection, and filtering:

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers.

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(obj)
output:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
print(obj['d'])
output:
3.0
```

```
Print(obj[['b', 'a', 'd']])
Output:
b    1.0
a    0.0
d    3.0
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```
Print(obj['b':'c'])
Output:
b    1.0
c    2.0
dtype: float64
```

Setting using these methods works just as you would expect:

```
obj['b':'c']= 5
```

```
print(obj)
output:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns= ['one', 'two', 'three', 'four'])
print(data)
output:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
Print(data['two'])
Output:
Ohio        1
Colorado    5
Utah        9
New York    13
Name: two, dtype: int32
```

```
data[['three', 'one']]
output:
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8

	three	**one**
New York	14	12

This might seem inconsistent to some readers, but this syntax arose out of practicality and nothing more. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

Print(data < 5)
Output:

	one	**two**	**three**	**four**
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
data[data < 5] = 0
print(data)
output:
```

	one	**two**	**three**	**four**
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Sorting and Ranking:

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

By default the data is sorted in ascending order.

```
import pandas as pd
import numpy as np
a=pd.Series(np.arange(4),index=[‘c’,’a’,’b’,’c’])
print(a)
output:
c    0
a    1
b    2
c    3
dtype: int32
```

```
b=a.sort_index()
print(b)
output:
a    1
b    2
c    0
c    3
dtype: int32
```

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],columns=['d', 'a', 'b', 'c'])
print(frame)
output:
```

```
       d  a  b  c
three  0  1  2  3
  one  4  5  6  7
```

```
f1=  frame.sort_index(axis=1)
print(f1)
output:
```

```
       a  b  c  d
three  1  2  3  0
  one  5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order.

```
f2=frame.sort_index(axis=1, ascending=False)
print(f2)
output:
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

If you want sort the elements in ascending order by values instead of index, it is allowed in pandas by using the method called sort_value() i.e. we can sort the array by using sort_value() method.

```
obj = pd.Series([4, 7, -3, 2])
print(obj)
output:
0    4
1    7
2   -3
3    2
dtype: int64

obj1=obj.sort_values()
print(obj1)
output:
2   -3
3    2
0    4
1    7
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
print(obj)
output:
0    4.0
1    NaN
2    7.0
3    NaN
4   -3.0
5    2.0
dtype: float64
```

Ranking:

Ranking is closely related to sorting, assigning ranks from one through the number of valid data points in an array. The rank methods for Series and DataFrame structure to rank the data.

```
import pandas as pd
import numpy as np
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
print(obj)
```

```
output:
0    7
1   -5
2    7
3    4
4    2
5    0
6    4
dtype: int64
```

```
obj1=obj.rank()
print(obj1)
output:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order they’re observed in the data:

```
obj2= obj.rank(method='first')
print(obj)
output:
0    7
1   -5
2    7
3    4
4    2
5    0
6    4
dtype: int64
```

we can also give the ranks in descending order:

```
obj3=obj.rank(ascending='False')
print(obj3)
output:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

DataFrame can compute ranks over the rows or the columns:

```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],'c': [-2, 5, 8, -2.5]})
print(frame)
```

output:

```
      b  a     c
0   4.3  0  -2.0
1   7.0  1   5.0
2  -3.0  0   8.0
3   2.0  1  -2.5
```

```
frame1=frame.rank()
print(frame1)
```

output:

```
     b    a    c
0  3.0  1.5  2.0
1  4.0  3.5  3.0
2  1.0  1.5  4.0
3  2.0  3.5  1.0
```

```
frame2=frame.rank (axis=1)
print(frame2)
```

output:

```
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
```

	b	a	c
2	1.0	2.0	3.0
3	3.0	2.0	1.0

Summarizing and computing descriptive statistics:
Pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame .

```
df=pd.DataFrame([[1.4,np.nan],[7.1,-4.5],[np.nan,np.nan],[0.75,-
1.3]],index=['a','b','c','d'],columns=['one','two'])
print(df)
output:
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Sum() method returns a Series containing column sums:

```
df1=df.sum()
print(df1)
output:
one    9.25
two   -5.80
dtype: float64
```

Passing axis=1 sums over the rows instead:

```
df2=df.sum(axis=1)
print(df2)
output:
a    1.40
b    2.60
c    0.00
d   -0.55
```

```
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the skipna option:

```
df3= df.mean(axis=1, skipna=False)
print(df3)
output:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:

```
df4= df.idxmax()
print(df4)
output:
one    b
two    d
dtype: object
```

Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

```
DF=df.describe()
print(DF)
output:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

In this describe() method ,it retrieves the complete information like count,sum,maximum,minimum,25% 50%,75 and finally standard deviation.

Unique values, value count and membership:

Here we are trying to find the unique values as well as value count in the given data set.

```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
obj1=obj.unique()
print(obj1)
output:
['c' 'a' 'd' 'b']
```

value_count: Return a Series containing counts of unique values.

```
obj2=obj.value_counts()
print(obj2)
output:
c    3
a    3
b    2
d    1
dtype: int64
```

Membership operator:
isin is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
obj1=obj.isin(['b', 'c'])
print(obj1)
output:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

Handling missing data:
missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data.
pandas uses the floating point value NaN (Not a Number) to represent missing data in both floating as well as in non-floating point arrays .

```
import numpy as  np
import pandas as pd
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
print(string_data)
output:'
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

```
string1=string_data.isnull()
print(string1)
output:
0    False
```

```
1    False
2     True
3    False
dtype: bool
```

The built-in Python None value is also treated as NA in object arrays:

```
string_data[0] = None
print(string_data.isnull())
```

output:

```
0     True
1    False
2     True
3    False
dtype: bool
```

Filtering out missing data:

We have a number of options for filtering out missing data. While doing it by hand is always an option, dropna can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
data = pd.Series([1, np.nan, 3.5, np.nan, 7])
print(data)
```

output:

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. dropna by default drops any row containing a missing value:

```
import pandas as pd
import numpy as np
df=pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],[np.nan,np.nan,np.nan], [np.nan, 6.5, 3.]])
print(df)
```

output:

	0	**1**	**2**
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN

```
     0    1    2

3  NaN  6.5  3.0
```

```
cleaned = df.dropna()
print(cleaned)
```

output:

```
     0    1    2

0  1.0  6.5  3.0
```

Passing how='all' will only drop rows that are all NA:

```
df1=df.dropna(how='all')
print(df1)
```

output:

```
     0    1    2

0  1.0  6.5  3.0

1  1.0  NaN  NaN

3  NaN  6.5  3.0
```

Dropping columns in the same way is only a matter of passing axis=1:

```
df2=df.dropna(axis=1,how='all')
print(df2)
```

output:

```
     0    1    2

0  1.0  6.5  3.0

1  1.0  NaN  NaN

2  NaN  NaN  NaN

3  NaN  6.5  3.0
```

Filling missing data:
Rather than filtering out missing data, we can full the missing data with the help of some methods.
The fillna method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value.

```
df3=df.fillna(0)
print(df3)
```

output:

	0	**1**	**2**
0	1.0	6.5	3.0
1	1.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	6.5	3.0

Calling fillna with a dict you can use a different fill value for each column:

```
df4=df.fillna({1: 0.5, 2: -1})
print(df4)
```

output;

	0	**1**	**2**
0	1.0	6.5	3.0
1	1.0	0.5	-1.0
2	NaN	0.5	-1.0
3	NaN	6.5	3.0

We can also use the ffill,bfill attributes with the fillna method.

```
df5=df.fillna(method='ffill')
print(df5)
```

output:

	0	1	2
0	1.0	6.5	3.0
1	1.0	6.5	3.0
2	1.0	6.5	3.0
3	1.0	6.5	3.0

```
df6=df.fillna(method='bfill')
print(df6)
```
output:

	0	1	2
0	1.0	6.5	3.0
1	1.0	6.5	3.0
2	NaN	6.5	3.0
3	NaN	6.5	3.0

While filling the missing, we can also fill the missing values with the arithmetic values of the series.

```
a=pd.Series([2,5,np.nan,3,np.nan,2])
b=a.fillna(a.mean())
print(b)
```
output:
```
0    2.0
1    5.0
2    3.0
3    3.0
4    3.0
5    2.0
dtype: float64
```