UNIT II

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 to pandas:

Pandas is a powerful and open-source Python library. The Pandas library is used for data manipulation and analysis. Pandas consist of data structures and functions to perform efficient operations on data.

Pandas is well-suited for working with **tabular data**, such as **spreadsheets** or **SQL tables**.

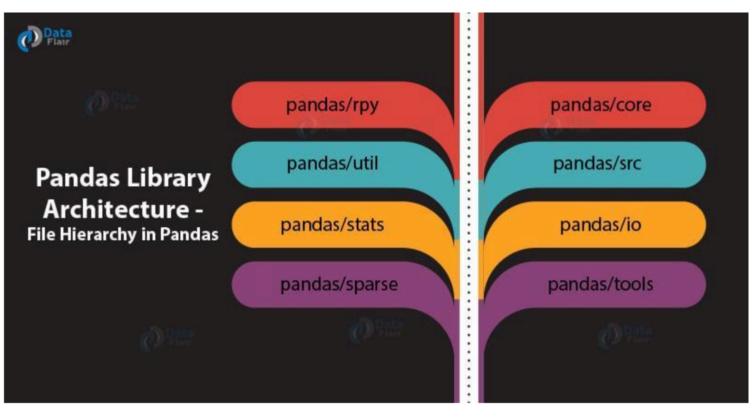
The data produced by Pandas is often used as input for plotting functions in **Matplotlib**, statistical analysis in **SciPy**, and machine learning algorithms in **Scikit-learn**.

We use pandas for:

- Data set cleaning, merging, and joining.
- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data.
- Columns can be inserted and deleted from DataFrame and higher-dimensional objects.
- Powerful group by functionality for performing split-apply-combine operations on data sets.
- Data Visualization.

Library Architecture:

File hierarchy in pandas consists of



- **pandas/core:** Consists of data structures about the Pandas library.

- **pandas/src:** Holds the basic functionality of Pandas depend on certain algorithms. They are usually written in C.
- **pandas/io:** Carries the tools to input and output, files, data, etc
- **pandas/tools:** Codes and algorithms for various functions and operations in Pandas. For example: Merge and join, concatenation, etc.
- **pandas/sparse**: Carries the sparse versions, i.e., the versions made to handle missing values of various Data Structures in Pandas.
- **pandas/stats**: Contains functions related to statistics, like linear regression
- **pandas/util**: Consist of testing tools and various other utilities to debug the library.
- **pandas/rpy**: Consists of an interface which helps to connect to R. It is called R2Py

Features of Pandas:

- It has a DataFrame object that is quick and effective, with both standard and custom indexing.
- Utilized for reshaping and turning of the informational indexes.
- For aggregations and transformations, group by data.
- It is used to align the data and integrate the data that is missing.
- Provide Time Series functionality.
- Process a variety of data sets in various formats, such as matrix data, heterogeneous tabular data, and time series.
- Manage the data sets' multiple operations, including subsetting, slicing, filtering, groupBy, reordering, and reshaping.
- It incorporates with different libraries like SciPy, and scikit-learn.

Applications

1. Data cleaning and preprocessing

Pandas is an excellent tool for cleaning and preprocessing data. It offers various functions for handling missing values, transforming data, and reshaping data structures.

2. Data exploration

Pandas makes it easy to explore and understand your data. You can quickly calculate summary and basic statistics, filter multiple rows or tables, and visualize data using Pandas' integration with Matplotlib.

3. Feature engineering

Pandas provides robust functionality for creating new features from existing data, such as calculating aggregate statistics, creating dummy variables, and applying custom functions.

4. Time series analysis

Pandas has built-in support for handling time series data, streamlining work with time-stamped data, resampling operations, and rolling statistics calculations.

5. Data science

Pandas plays a crucial role in preparing data for machine learning models. By cleaning, preprocessing, and transforming data with Pandas, you can create structured datasets that can be used with machine learning libraries like scikit-learn or TensorFlow.

Data Structures, Series, DataFrame:

Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*.

Syntax: *pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)*

Parameters:

- ***data****: array- Contains data stored in Series.*
- ***index****: array-like or Index (1d)*
- ***dtype****: str, numpy.dtype, or ExtensionDtype, optional*
- ***name****: str, optional*
- ***copy****: bool, default False*
- *fast_path : is an* ***internal parameter*** *that is not meant for public use. If true additional checks are not done*

Example:

list = [1,2,3,4,5] # create series form a integer list

res = pd.Series(list)

print(res)

output:

0 1

1 2

2 3

3 4

4 5

dtype: int64

- we can get the array representation and index object of the Series via its values and index attributes

Example :

list.values

Output : array([1,2,3,4,5])

list.index

Output: Int64Index([0, 1, 2, 3])

- A Series can be created with an index identifying each data point:

Example:

obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

obj2

Output:

d 4

b 7

a -5

c 3

- Compared with a regular NumPy array, we can use values in the index when selecting single values or a set of values:

Example:

->obj2['a']

Output: -5

->obj2['d'] = 6

-> obj2[['c', 'a', 'd']]

Output:

c 3

a -5

d 6

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3
```

```
In [15]: obj2[obj2 > 0]
Out[15]:
d    6
b    7
c    3
```

```
In [16]: obj2 * 2
Out[16]:
d    12
b    14
a   -10
c     6
```

```
In [17]: np.exp(obj2)
Out[17]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
```

- A Series can be created by passing a dictionary

Example:

sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = Series(sdata)
obj3

- Output:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000

Note: When only passing a dictionary, the index in the resulting Series will have the dictionary keys in sorted order.

Example :

states = ['California', 'Ohio', 'Oregon', 'Texas']

obj4 = Series(sdata, index=states)

obj4

Output:

California NaN

Ohio 35000

Oregon 16000

Texas 71000

In this case, 3 values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number) which is considered in pandas to mark missing or *NA* values.

- The **isnull** and **notnull** functions in pandas should be used to detect missing data:

Example 1:pd.isnull(obj4) Example 2: pd.notnull(obj4)

Output:

Example 1	Example 2
California True	California False
Ohio False	Ohio True
Oregon False	Oregon True
Texas False	Texas True

- Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

Example :

```
obj4.name = 'population'
obj4.index.name = 'state'
obj4
```

Output:

```
state
California    NaN
Ohio          35000
Oregon        16000
Texas         71000
Name: population
```

DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric,string, boolean, etc.). The DataFrame has both a row and column index.

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
```

'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

frame = DataFrame(data)

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

frame

Output:

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

- If we specify a sequence of columns, the DataFrame's columns will be exactly what we pass:

Example:

DataFrame(data, columns=['year', 'state', 'pop'])

Output:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9

- If we pass a column that isn't contained in data, it will appear with NA values in the result:

Example

frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'], index=['one', 'two', 'three', 'four', 'five'])

frame2

Output:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

Example:

```
frame2['state']
```

Output:

```
one Ohio
two Ohio
three Ohio
four Nevada
five Nevada
Name: state
```

Example:

```
frame2.year
```

Ouput:

```
one    2000
two    2001
three 2002
four   2001
five   2002
Name: year
```

- Rows can also be retrieved by position or name by a couple of methods, such as the **ix** indexing field

Example:

```
frame2.ix['three']
```

Output:

```
year   2002
state  Ohio
pop    3.6
debt   NaN
Name: three
```

- Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

Example:

```
frame2['debt'] = 16.5
frame2
```

Output:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5

two 2001 Ohio 1.7 16.5
three 2002 Ohio 3.6 16.5
four 2001 Nevada 2.4 16.5
five 2002 Nevada 2.9 16.5

Example:

```
frame2['debt'] = np.arange(5.)
frame2
```

Output:

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4

- When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes:

Example:

```
val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame2['debt'] = val
frame2
```

Output:

	Year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

- Assigning a column that doesn't exist will create a new column. The del keyword will delete columns as with a dict:

Example:

```
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

Output:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

Example:
del frame2['eastern']
frame2.columns
Output:
Index([year, state, pop, debt], dtype=object)

- Another common form of data is a nested dict of dicts format:

Example:
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:
Example:
frame3 = DataFrame(pop)
frame3

Output:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

- Transpose of a DataFrame can be:

Example:
frame3.T

Output:

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

- If a DataFrame's index and columns have their name attributes set, these will also be displayed:

Example"
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
Output:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

- Like Series, the values attribute returns the data contained in the DataFrame as a 2D ndarray:

Example:
frame3.values

Output:
array([[nan, 1.5],
[2.4, 1.7],
[2.9, 3.6]])

Index Objects:

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 Data Frame is internally converted to an Index:

Example :

obj = Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index

Output
Index([a, b, c], dtype=object)

Example :
index[1:]
Output: Index([b, c], dtype=object)

Note : Index objects are immutable and thus can't be modified by the user

Main Index objects in pandas:

Class	Description
`Index`	The most general Index object, representing axis labels in a NumPy array of Python objects.
`Int64Index`	Specialized Index for integer values.
`MultiIndex`	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
`DatetimeIndex`	Stores nanosecond timestamps (represented using NumPy's `datetime64` dtype).
`PeriodIndex`	Specialized Index for Period data (timespans).

1.Index:

```
import pandas as pd
index = pd.Index(["apple", "banana", "cherry"])
print(index)
```

output:

```
Index(['apple', 'banana', 'cherry'], dtype='object')
```

2. Int64Index:

```
index = pd.Int64Index([10, 20, 30, 40])
print(index)
```

output:

```
Int64Index([10, 20, 30, 40], dtype='int64')
```

3. A **MultiIndex** (also called a hierarchical index) allows multiple levels of indexing in a DataFrame or Series. This is useful when working with complex datasets that have multiple categorical variables.

```
import pandas as pd
arrays = [
    ['A', 'A', 'B', 'B'],  # First level
    [1, 2, 1, 2]           # Second level
]

multi_index = pd.MultiIndex.from_arrays(arrays, names=('Letter', 'Number'))
print(multi_index)
```

output:

```
MultiIndex([('A', 1),
            ('A', 2),
            ('B', 1),
            ('B', 2)],
           names=['Letter', 'Number'])
```

4. DatetimeIndex (Stores timestamps):
Is useful for working with time-series data, allowing efficient indexing, filtering, and resampling based on time.

Example:

```
dates = pd.date_range("2024-01-01", periods=5, freq="D")
print(dates)
```

Output:
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04', '2024-01-05'], dtype='datetime64[ns]', freq='D')

5.PeriodIndex:

PeriodIndex is a specialized index type in Pandas that represents **time spans** (e.g., months, quarters, years) instead of individual timestamps.

```
import pandas as pd

periods = pd.period_range(start="2024-01", periods=5, freq="M")
print(periods)
```

output:
PeriodIndex(['2024-01', '2024-02', '2024-03', '2024-04', '2024-05'], dtype='period[M]')

RealTime Use Case

Financial reports (monthly, quarterly, yearly data)	✅ PeriodIndex
Business cycles, sales periods	✅ PeriodIndex
Stock market data (daily/hourly prices)	✅ DatetimeIndex
IoT data, event logs, sensor readings	✅ DatetimeIndex
Scheduling events with exact times	✅ DatetimeIndex

Index methods and properties:

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index i deleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index i
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

Feature	Series	Index Object
Definition	A one-dimensional labeled array that holds data.	An immutable sequence used for row or column labels.
Mutability	Mutable (can modify values)	Immutable (cannot modify values)
Usage	Stores actual data values with an index.	Only stores index labels, not values.
Data Structure	A full-fledged data structure with data and an index.	A simpler structure used only to label Series or DataFrames.
Example Creation	`pd.Series([10, 20, 30], index=['a', 'b', 'c'])`	`pd.Index(['a', 'b', 'c'])`
Contains Data?	Yes, it contains actual data values.	No, it only holds index labels.
Supports Arithmetic?	Yes, supports arithmetic operations.	No, arithmetic is not directly supported.
Supports Slicing?	Yes, can slice and modify values.	Yes, but slicing creates a new Index (since it's immutable).
Supports Reindexing?	Yes, you can reindex a Series.	Yes, but it creates a new Index.
Example Use Case	Storing temperature data for cities.	Storing unique identifiers like product codes.

Essential Functionality

Reindexing

A critical method on pandas objects is reindex, which means to create a new object with the data *conformed* to a new index.

Example:

obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj
Output:
d 4.5
b 7.2
a -5.3
c 3.6
Calling reindex on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

Example:
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2

Output:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN

- To Fill NaN values use fill_value

obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)

Output
a -5.3
b   7.2
c   3.6
d   4.5
e   0.0

For filling of values when reindexing. The method option is used as ffill which forward fills the values:

Example:
obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3.reindex(range(6), method='ffill')

Output:
0 blue
1 blue
2 purple
3 purple
4 yellow
5 yellow

Reindex method options:

Argument	Description
ffill or pad	Fill (or carry) values forward
bfill or backfill	Fill (or carry) values backward

- With **DataFrame**, reindex can alter either the (row) index, columns, or both. When passed just a sequence, the rows are reindexed in the result:

Example:
frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],columns=['Ohio', 'Texas', 'California'])
frame

Output:

```
  Ohio Texas California
a   0     1       2
c   3     4       5
d   6     7       8
```

frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2

Output:

```
     Ohio Texas California
a     0     1       2
b    NaN   NaN     NaN
c     3     4       5
d     6     7       8
```

- The columns can be reindexed using the columns keyword:

states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states)

Output:

```
     Texas Utah California
a      1    NaN     2
c      4    NaN     5
d      7    NaN     8
```

- Both can be reindexed in one shot, though interpolation will only apply row-wise (axis0):

Example: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill', columns=states)

Output:

	Texas	Utah	California
a	1	NaN	2
b	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Dropping entries from an axis

Drop method will return a new object with the indicated value or values deleted from an axis:
Example:

```
obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_obj = obj.drop('c')
new_obj
```

Output:

```
a  0
b  1
d  3
e  4
```

Example:

```
obj.drop(['d', 'c'])
```

Output:

```
a 0
b 1
e 4
```

- With DataFrame, index values can be deleted from either axis:

Example:

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

Output:

	one	two	three	four
Utah	8	9	10	11

New York 12 13 14 15

Example:
data.drop('two', axis=1)
Output:

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

data.drop(['two', 'four'], axis=1)
Output:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

<u>Indexing, selection, and filtering:</u>

Series index values can be used instead of numbers as in Numpy indexing

Example:
obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj['b']

output: 1.0

>> obj[1]

output: 1.0

>> obj[2:4]
Output
c 2
d 3

>> obj[['b', 'a', 'd']]
Output:
b 1
a 0
d 3

>> obj[[1, 3]]
Output:
b 1
d 3

>> obj[obj < 2]
Output:
a 0
b 1

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

>> obj['b':'c']
Output:
b 1
c 2

- *Setting* using these methods works just as you would expect:

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

>>obj

Output:

a 0

b 5

c 5

d 3

- **With DataFrame**

Example :

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

>> data['two'] Column display

Output:

Ohio	1
Colorado	5
Utah	9
New York	13

Name: two, dtype: int32

>> data[['three', 'one']] Two Column display

Output:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

>> data[:2] -> Slicing

Output:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
>> data[data['three'] > 5]          -> Filtering the data with given condition
Output:
            one   two    three   four
Colorado     4     5       6       7
Utah         8     9      10      11
New York    12    13      14      15

>> data < 5        -->  to display Boolean values
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.loc['Colorado', ['two', 'three']]  data to retrieve with a row and columns

Output:
 two     5
 three   6
Name: Colorado, dtype: int32

>> data.loc[:'Utah','two']       Slicing to implement on rows and columns

Output:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32

>> data.loc[data.three > 5, :'two']  from 3rd column the rows which are greater than 5

Output:
            One  two
Colorado     0    5
Utah         8    9
New York    12   13
```

<u>Sorting and ranking:</u>

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

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()
```

output :
a 1
b 2
c 3
d 0
dtype: int64

- With a DataFrame, you can sort by index on either axis:

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

Output:

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

>> frame.sort_index(axis=1)
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

>> frame.sort_index(axis=1, ascending=False)

Output:

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

- To sort a Series by its values,

>> obj = pd.Series([4, 7, -3, 2])
>>obj.sort_values()

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])
>> obj.sort_values()
```

Output:

```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

- On DataFrame, to sort by the values in one or more columns, pass one or more column names to the by option

```
>>frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
>>frame
```

Output:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

```
>> frame.sort_values(by='b')
```

Output:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

```
>> frame.sort_values(by=['a','b'])
```

Output:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

- *Ranking* is closely related to sorting, assigning ranks from one through the number of valid data points in an array.

Syntax:
rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)

The rank() method takes the following arguments:

- axis: specifies whether to rank rows or columns
- method: specifies how to handle equal values
- numeric_only: rank only numeric data if True
- na_option: specifies how to handle NaN
- ascending: specifies whether to rank in ascending order
- pct: specifies whether to display the rank as a percentage.

Example:

```
data = {'Score': [78, 85, 96, 85, 90]}
df = pd.DataFrame(data)
df['Rank'] = df['Score'].rank()
print(df)
```

Output:

```
  Score  Rank
0    78   1.0
1    85   2.5
2    96   5.0
3    85   2.5
4    90   4.0
```

Ranking with Method

The max method assigns maximum possible rank to the equal values.

```
data = {'Score': [78, 85, 96, 85, 90]}
df = pd.DataFrame(data)

# rank using the 'max' method for ties
df['Rank'] = df['Score'].rank(method='max')

print(df)
Score  Rank
0    78   1.0
1    85   3.0
2    96   5.0
3    85   3.0
4    90   4.0
```

 descending order with the highest score receiving the lowest rank.

```
>> data = {'Score': [78, 85, 96, 85, 90]}
>>df = pd.DataFrame(data)
# rank in descending order
>>df['Rank'] = df['Score'].rank(ascending=False)
>> print(df)
```

Output:

```
0    78   5.0
1    85   3.5
2    96   1.0
3    85   3.5
4    90   2.0
```

- The numeric_only argument is used to rank only numeric columns when applied to a DataFrame.

Example:

```
data = {
   'Score': [78, 85, 96, 85, 90],
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva']
}
df = pd.DataFrame(data)

# rank all columns
print('All columns:')
print(df.rank())
print()

# rank numeric columns only
print('Numeric columns:')
print(df.rank(numeric_only=True))
```

output:

```
All columns:
  Score  Name
0   1.0   1.0
1   2.5   2.0
2   5.0   3.0
3   2.5   4.0
4   4.0   5.0

Numeric columns:
  Score
0   1.0
1   2.5
2   5.0
```

3   2.5
4   4.0

- na_option argument to determine how NaN values in the data are handled.

Example:

```
data = {'Score': [78, 85, None, 85, 90]}
df = pd.DataFrame(data)
# rank with NaN placed at the bottom
df['Rank'] = df['Score'].rank(na_option='bottom')
print(df)
```

Output:

```
   Score  Rank
0   78.0   1.0
1   85.0   2.5
2    NaN   5.0
3   85.0   2.5
4   90.0   4.0
```

Summarizing and Computing Descriptive Statistics:

Descriptive statistics are essential tools in data analysis, offering a way to summarize and understand your data. In Python's Pandas library, there are numerous methods available for computing descriptive statistics on Series and DataFrame objects.

These methods provide various aggregations like sum(), mean(), and quantile(), as well as operations like cumsum() and cumprod() that return an object of the same size.

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

- describe(percentiles=**None,** include=**None,** exclude=**None)**
  will return count,mean,std,min,q1(25%),q2(50%-median),q3(75%) and max on numeric data
  percentiles by default (25%,50% and 75%) and can be changed to [0,1](0%,50%,100%)
  include, exclude will take which column type should be included

```
>>  import pandas as pd
>> import numpy as np
>> s=pd.Series([2,1,5,6,7,9])
>> s.describe()
```

```
Output:
count    6.00000
mean     5.00000
std      3.03315
min      1.00000
25%      2.75000
```

```
50%      5.50000
75%      6.75000
max      9.00000
dtype: float64
```

- If data was in non-numeric type(object) then will return count,unique,top and freq as shown below

```
>> s=pd.Series(['b',1,'c',6,'c','a','b','c'])
>> s.describe()
```

Output:

```
count     8
unique    5   (no. of unique values in the series)
top       c      ( Most repeated item in series)
freq      3       (Count of a repeated item)
dtype: object
```

```
>> df = pd.DataFrame({'category': ['d','e','f','g'],'numeric': [1, 2, 3,4], 'object': ['b', 'c', 'd','e']})
 df.describe(include=np.object)
 #df.describe(exclude=np.number)
 #df.describe(include='all')
```

Output:

	category	object
count	4	4
unique	4	4
top	e	e
freq	1	1

groupby()

Group DataFrame or Series using a mapper or by a Series of columns.

```
>>df = pd.DataFrame({'Dept': ['CSM', 'CSM', 'CSE', 'CSD', 'CSE'],
                'Sal': [10000, 20000, 15000, 25000, 30000]})
>> df.groupby('Dept')['Sal'].max()   # returns the maximum salary from each dept
```

Output :

```
Dept
CSD    25000
CSE    30000
CSM    20000
```

Unique Values, Value Counts:

Unique function in pandas gives unique values present in Series and DataFrame.

nunique(): Compute array of unique values in a Series, returned in the order observed(includes NaN).
unique(): Returns count of unique values excluding NaN

```
>> obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c',None])
>> obj.unique()
```

Output:
array(['c', 'a', 'd', 'b', None], dtype=object)

```
>> obj.nunique()
```

Output:4

value_counts() : Return a Series containing counts of unique values in descending order of input and count.

- **Note** : value_counts() can't be applied on DataFrame.

syntax :value_counts**(**normalize**=False,**sort**=True,**ascending**=False,**bins**=None,**dropna**=True)**
normalize: If True then the object returned will contain the relative frequencies of the unique values.
Bins: grouping into half-open bins

```
>>s = pd.Series([3, 1, 2, 3, 4,5, np.nan])
>>s.value_counts()
```

Output:

```
3.0    2
5.0    1
4.0    1
2.0    1
1.0    1
dtype: int64
```

```
>> s.value_counts(bins=3)
```

Output:

```
(3.667, 5.0]      2
(2.333, 3.667]    2
```

(0.995, 2.333]   2
dtype: int64

Note: value_counts is also available as a top-level pandas method that can be used with any array or sequence:

```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
pd.value_counts(obj.values, sort=False)
```

Output:

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

Handling Missing Data:

Missing data is common in most data analysis applications. 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.

Indentifying missing values:

isnull   Return like-type object containing boolean values indicating which values are missing / NA.

notnull Negation of isnull.

```
>> string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
>> string_data.isnull()
```

Output:

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

```
>>string_data.notnull()
```

Output:

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

Filtering of missing data:

dropna: Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.

Syntax: dropna**(**axis=**0,** inplace=**False, ****kwargs**)** **For Series**

>> data = pd.Series([1, None, 3.5, None, 7])
>> data.dropna() or data[data.notnull()]

Output:
0 1.0
2 3.5
4 7.0
dtype: float64

Syntax : dropna**(**axis=**0,** how=**'any',** thresh=**None,** subset=**None,** inplace=**False)**
 For DatFrame

axis=0 – drop rows where missing values 1- drop columns where missing values are present
how= 'any' :If any NA values are present, drop that row or column.
'all' : If all values are NA, drop that row or column.
tresh = tells how many non-NA values should be present in a row/column
subset= list of column names should be included for dropping
inplace=False : to return the modified data frame
True: to modify the original data frame

>> ddata = pd.DataFrame([[1., 6.5, 3.], [1., None, None],[None, None, None], [None, 6.5, 3.]])

	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

**Refer exercises from jupyter notebook file

Filling of missing data:
Rather than filtering out missing data (and potentially discarding other data along with it), we can fill in the "holes" in any number of ways. For most purposes, the fillna method is used.

Syntax:
fillna**(**value=**None,**method=**None,**axis=**None,**inplace=**False,**limit=**None,**downcast=**None)**

value : scalar, dict, Series, or DataFrame
method: {'backfill', 'bfill', 'pad', 'ffill', None}
axis : {0 or 'index', 1 or 'columns'}
inplace : bool, default False
limit: How many NaN values to be filled

**Refer exercises from jupyter notebook file

Answer the below questions
(a) Find the number of unique cities in the dataset.
(b) List all the unique categories in the 'Category' column.
(c) Count how many times each city appears in the dataset.
(d) Explain the difference between df['City'].nunique() and df['City'].unique().
(e) Sort the city column