Pandas

Pandas is a powerful Python library specifically designed to work on data frames that have "relational" or "labelled" data.

Applications of Pandas

The most common applications of Pandas are as follows:

Data Analysis – Easily manipulate and analyze structured data (like CSV, Excel).

Data Cleaning – Handle missing values, duplicates, and format inconsistencies.

Data Transformation – Filter, group, merge, and reshape datasets.

Time Series Analysis – Work with time-indexed data for trends and forecasts.

Data Visualization – Integrate with libraries (e.g., Matplotlib, Seaborn) for plotting.

Data Export/Import – Read/write data from various formats (CSV, Excel, SQL, JSON).

Pandas Data Structures

Data structures in Pandas are designed to handle data efficiently. They allow for the organization, storage, and modification of data in a way that optimizes memory usage and computational performance. Python Pandas library provides two primary data structures for handling and analyzing data:

Data Structure	Dimensions	Description
Series	1	A one-dimensional labelled homogeneous array; size immutable.
DataFrame	2	A two-dimensional labelled, size-mutable tabular structure with potentially heterogeneously typed columns.

Note: All Pandas data structures are value mutable, meaning their contents can be changed. However, their size mutability varies.

Series

A Series is a one-dimensional labeled array that can hold any data type. It can store integers, strings, floating-point numbers, etc. Each value in a Series is associated with a label (index), which can be an integer or a string.

```
import pandas as pd

data = ['Steve', '35', 'Male', '3.5']
series = pd.Series(data, index=['Name', 'Age', 'Gender', 'Rating'])
print(series)
```

```
Name      Steve
Age          35
Gender     Male
Rating      3.5
dtype: object
```

DataFrame

A DataFrame is a two-dimensional labeled data structure with columns that can hold different data types. It is similar to a table in a database or a spreadsheet. Consider the following data representing the performance rating of a sales team.

```
import pandas as pd

# Data represented as a dictionary
data = {
    'Name': ['Steve', 'Lia', 'Vin', 'Katie'],
    'Age': [32, 28, 45, 38],
    'Gender': ['Male', 'Female', 'Male', 'Female'],
    'Rating': [3.45, 4.6, 3.9, 2.78]
}

# Creating the DataFrame
df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

```
    Name  Age  Gender  Rating
0  Steve   32    Male    3.45
1    Lia   28  Female    4.60
2    Vin   45    Male    3.90
3  Katie   38  Female    2.78
```

```python
import pandas as pd

# Data represented as a dictionary
data = {
    'Name': ['Steve', 'Lia', 'Vin', 'Katie'],
    'Age': [32, 28, 45, 38],
    'Gender': ['Male', 'Female', 'Male', 'Female'],
    'Rating': [3.45, 4.6, 3.9, 2.78]
}

# Creating the DataFrame
df = pd.DataFrame(data)

# Display a Series within a DataFrame
print(df['Name'])
```

```
0    Steve
1      Lia
2      Vin
3    Katie
Name: Name, dtype: object
```

Pandas Series

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print("Output:\n",s)
```

```
Output:
 100    a
101    b
102    c
103    d
dtype: object
```

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print(s)
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

Basics of Pandas Series Slicing

Series slicing can be done by using the [:] operator, which allows you to select subset of elements from the series object by specified start and end points.

Below are the syntax's of the slicing a Series:

Series[start:stop:step]: It selects elements from start to end with specified step value.

Series[start:stop]: It selects items from start to stop with step 1.

Series[start:]: It selects items from start to the rest of the object with step 1.

Series[:stop]: It selects the items from the beginning to stop with step 1.

Series[:]: It selects all elements from the series object.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print(s[:3])
#retrieve the last three element
print(s[-3:])
# Slice multiple elements
print(s['a':'d'])
```

```
a    1
b    2
c    3
dtype: int64
c    3
d    4
e    5
dtype: int64
a    1
b    2
c    3
d    4
dtype: int64
```

Modifying Values after Slicing

After slicing a Series, you can also modify the values, by assigning the new values to those sliced elements.

```
import pandas as pd
s = pd.Series([1,2,3,4,5])

# Display the original series
print("Original Series:\n",s)

# Modify the values of first two elements
s[:2] = [100, 200]

print("Series after modifying the first two elements:",s)
```

```
Original Series:
 0    1
1    2
2    3
3    4
4    5
dtype: int64
Series after modifying the first two elements: 0    100
1    200
2      3
3      4
4      5
dtype: int64
```

Arithmetic Operations on a Pandas Series

Operation	Syntax	Description
Addition	`s + 2`	Adds 2 to each element
Subtraction	`s - 2`	Subtracts 2 from each element
Multiplication	`s * 2`	Multiplies each element by 2
Division	`s / 2`	Divides each element by 2
Exponentiation	`s ** 2`	Squares each element (power of 2)
Modulus	`s % 2`	Computes remainder when divided by 2
Floor Division	`s // 2`	Performs integer division (quotient floored)

Arithmetic Operations Between Two Series

```
import pandas as pd

# Create Series s1 with indexes: a, b, c, d, e
s1 = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])

# Create Series s2 with indexes: x, a, b, c
s2 = pd.Series([9, 8, 6, 5], index=['x', 'a', 'b', 'c'])

# Arithmetic operations
print('\nAddition:\n', s1 + s2)
print('\nSubtraction:\n', s1 - s2)
print('\nMultiplication:\n', s1 * s2)
print('\nDivision:\n', s1 / s2)
```

```
Addition:
 a    9.0
b    8.0
c    8.0
d    NaN
e    NaN
x    NaN
dtype: float64

Subtraction:
 a   -7.0
b   -4.0
c   -2.0
d    NaN
e    NaN
x    NaN
dtype: float64
```

```
Multiplication:
 a     8.0
b    12.0
c    15.0
d     NaN
e     NaN
x     NaN
dtype: float64

Division:
 a    0.125000
b    0.333333
c    0.600000
d         NaN
e         NaN
x         NaN
dtype: float64
```

Converting Series to Other Objects

Following are the commonly used methods for converting Series into other formats:

Method	Description
to_list()	Converts the Series into a Python list.
to_numpy()	Converts the Series into a NumPy array.
to_dict()	Converts the Series into a dictionary.
to_frame()	Converts the Series into a DataFrame.
to_string()	Converts the Series into a string representation for display.

```python
import pandas as pd

# Create a Pandas Series
s = pd.Series([1, 2, 3])

# Convert Series to a Python list
result = s.to_list()

print("Output:",result)
print("Output Type:", type(result))

# Convert Series to a NumPy Array
result = s.to_numpy()

print("Output:",result)
print("Output Type:", type(result))
```

```
Output: [1, 2, 3]
Output Type: <class 'list'>
Output: [1 2 3]
Output Type: <class 'numpy.ndarray'>
```

```python
import pandas as pd

# Create a Pandas Series
s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# Convert Series to a Python dictionary
result = s.to_dict()

print("Output:",result)
print("Output Type:", type(result))
```

```
Output: {'a': 1, 'b': 2, 'c': 3}
Output Type: <class 'dict'>
```

Pandas – DataFrame

A DataFrame in Python's pandas' library is a two-dimensional labeled data structure that is used for data manipulation and analysis. It can handle different data types such as integers, floats, and strings. It is size mutable. We can think of a DataFrame as similar to an SQL table or a spreadsheet data representation.

Creating a pandas DataFrame

A pandas DataFrame can be created using the following constructor parameters.

SR. No	Parameter & Description
1	**data** data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	**index** For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.
3	**columns** This parameter specifies the column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.
4	**dtype** Data type of each column.
5	**copy** This command (or whatever it is) is used for copying of data, if the default is False.

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

```
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13
```

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
```

```
    Name  Age
0    Tom   28
1   Jack   34
2  Steve   29
3  Ricky   42
```

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
```

Indexing in Pandas

Similar to Python and NumPy indexing ([ ]) and attribute (.) operators, Pandas provides straightforward methods for accessing data within its data structures. However, because the data type being accessed can be unpredictable, relying exclusively on these standard operators may lead to optimization challenges.

Pandas provides several methods for indexing and selecting data, such as:

- Label-Based Indexing with .loc
- Integer Position-Based Indexing with .iloc
- Indexing with Brackets []

Label-Based Indexing with .loc

Following is the syntax of slicing a DataFrame using the .loc[] attribute :

DataFrame.loc[row_label_start:row_label_end, column_label_start:column_label_end]

Where, row_label_start and row_label_end are indicates the start and end labels of the DataFrame rows. Similarly, column_label_start and column_label_end are the column labels.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4), # Create a DataFrame 'df' with random numbers (8 rows and 4 columns)
                  index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'], # Rows are labeled with custom index ['a' to 'h']
                  columns=['A', 'B', 'C', 'D']) # Columns are named 'A', 'B', 'C', and 'D'

# Display the original DataFrame
print("Original DataFrame:\n", df)

# .loc is used for label-based indexing (row and column labels)
print('\nResult:\n', df.loc[:, 'A'])  # ':' means all rows, 'A' specifies the column
```

```
Original DataFrame:
           A         B         C         D
a -0.090608  0.072678  1.506535 -0.993647
b  0.758503  0.070799 -0.094593 -0.872920
c -0.216746  0.394235  2.939156 -1.737788
d  1.233330 -1.036672  0.583306  0.519301
e -1.705275  0.155426  1.431306 -0.014671
f  2.205250  0.463624 -1.417899 -1.536805
g -1.127973 -1.531993 -0.731792 -0.434562
h -1.228793  1.231154 -0.567553 -0.134365

Result:
 a   -0.090608
b    0.758503
c   -0.216746
d    1.233330
e   -1.705275
f    2.205250
g   -1.127973
h   -1.228793
Name: A, dtype: float64
```

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])

# Select few rows for multiple columns, say list[]
print(df.loc[['a','b','f','h'],['A','C']])
```

```
          A         C
a -0.743335  0.358730
b  0.902213 -0.447809
f -0.253850 -1.795058
h -0.597445  0.137873
```

Integer Position-Based Indexing with .iloc

The Pandas DataFrame.iloc[] attribute used to slice a DataFrame based on the integer position (i.e, integer-based indexing) of rows and columns.

Following is the syntax of slicing a DataFrame using the .iloc[] attribute −

DataFrame.iloc[row_start:row_end, column_start:column_end]

Where, row_start and row_end are indicates the start and end integer-based index values of the DataFrame rows. Similarly, column_start and column_end are the column index values.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

print("Original DataFrame:\n", df)

# select all rows for a specific column
print('\nResult:\n',df.iloc[:4])
```

```
Original DataFrame:
          A         B         C         D
0  0.095874 -0.993600 -0.690025  1.403570
1 -1.071942  0.630795 -2.255658 -0.782670
2  1.189937 -0.103614 -0.289287 -0.203541
3 -0.783170 -0.172301  0.521611 -0.535989
4 -1.442989  1.296939 -1.225757  1.994669
5 -0.941695 -0.490993 -0.620028 -0.977928
6 -1.036550  0.700464 -0.820962  0.183185
7  0.366967  0.696571  0.256713 -0.893327

Result:
          A         B         C         D
0  0.095874 -0.993600 -0.690025  1.403570
1 -1.071942  0.630795 -2.255658 -0.782670
2  1.189937 -0.103614 -0.289287 -0.203541
3 -0.783170 -0.172301  0.521611 -0.535989
```

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

df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

# Integer slicing
print(df.iloc[1:5, 2:4])
print(df.iloc[[1, 3, 5], [1, 3]])
```

```
          C         D
1  0.483723 -0.206030
2  0.413034 -0.108127
3  0.390134 -0.438651
4  0.067238  1.312553
          B         D
1 -2.386186 -0.206030
3  0.907397 -0.438651
5  0.600926 -0.087802
```

```python
import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)

# Slice a single column
col_A = df.iloc[:, 0]
print("Slicing a single column A using iloc[]:")
print(col_A)

# Slice multiple columns
cols_AB = df.iloc[:, 0:2]
print("Slicing multiple columns A and B using iloc[]:")
print(cols_AB)
```

```
Slicing a single column A using iloc[]:
0    1
1    2
2    3
Name: A, dtype: int64
Slicing multiple columns A and B using iloc[]:
   A  B
0  1  4
1  2  5
2  3  6
```

Indexing with Brackets []

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

# Accessing Multiple Columns
print(df['A'])
print(df[['A', 'B']])
```

```
0   -2.376217
1   -0.914106
2    0.131745
3    0.399043
4    0.625874
5   -0.013148
6    1.739137
7    1.783914
Name: A, dtype: float64
          A         B
0 -2.376217  2.646317
1 -0.914106  2.124020
2  0.131745 -0.590083
3  0.399043 -0.367624
4  0.625874  0.329894
5 -0.013148  0.040176
6  1.739137  0.508661
7  1.783914 -0.556763
```

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

# Accessing Rows
print(df[:1])
```

```
          A         B         C         D
0  0.393932 -1.029468 -1.980494  0.427398
```

Feature	`.loc[]`	`.iloc[]`	`[]`
Uses	**Labels**	**Integer positions**	**Columns or rows** (context-based)
Select rows/columns	Yes	Yes	Only columns or row slices
Supports slicing	Yes (inclusive)	Yes (exclusive)	Yes (but limited)
Boolean indexing	Yes	Yes	No (usually not)
Example	`df.loc['x', 'A']`	`df.iloc[0, 0]`	`df['A']`

Modifying Values After Slicing

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame([['a', 'b'], ['c', 'd'], ['e', 'f'], ['g', 'h']],
                  columns=['col1', 'col2'])

# Display the Original DataFrame
print("Original DataFrame:", df, sep='\n')

# Modify a subset of the DataFrame using iloc
df.iloc[1:3, 0] = ['x', 'y']

# Display the modified DataFrame
print('Modified DataFrame:',df, sep='\n')
```

```
Original DataFrame:
  col1 col2
0    a    b
1    c    d
2    e    f
3    g    h
Modified DataFrame:
  col1 col2
0    a    b
1    x    d
2    y    f
3    g    h
```

```python
import pandas as pd

# Create a DataFrame
df = pd.DataFrame([['a', 'b'], ['c', 'd'], ['e', 'f'], ['g', 'h']],
                  columns=['col1', 'col2'])

# Display the Original DataFrame
print("Original DataFrame:", df, sep='\n')

# Modify a subset of the DataFrame using iloc
df.iloc[1:3, 0:2] = [['x', 'y'], ['z', 'A']]

# Display the modified DataFrame
print('Modified DataFrame:',df, sep='\n')
```

```
Original DataFrame:
  col1 col2
0    a    b
1    c    d
2    e    f
3    g    h
Modified DataFrame:
  col1 col2
0    a    b
1    x    y
2    z    A
3    g    h
```

Modifying DataFrame

Some of the most common DataFrame modifications include:

- Renaming column or row labels.
- Adding or inserting new columns.
- Updating or replacing existing column values.
- Removing unnecessary columns.

Renaming Column/Rows Labels in a DataFrame

```
import pandas as pd

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

# Rename column 'A' to 'aa'
df = df.rename(columns={'A': 'aa'})

# Display modified DataFrame
print("Modified DataFrame:")
print(df)

# Rename the multiple row labels
df = df.rename(index={'x': 'r1', 'y':'r2', 'z':'r3'})

# Display modified DataFrame
print("Modified DataFrame:")
print(df)
```

```
Modified DataFrame:
   aa  B
0   1  4
1   2  5
2   3  6
Modified DataFrame:
   aa  B
0   1  4
1   2  5
2   3  6
```

Adding or inserting new columns.

```
import pandas as pd

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

# Add a new column 'C' with values
df['C'] = [7, 8, 9]

# Display updated DataFrame
print("DataFrame after adding a new column 'C':")
print(df)

# Insert a new column 'D' at position 1
df.insert(1, 'D', [10, 11, 12])

# Display updated DataFrame
print("DataFrame after inserting column 'D' at position 1:")
print(df)
```

```
DataFrame after adding a new column 'C':
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
DataFrame after inserting column 'D' at position 1:
   A   D  B  C
0  1  10  4  7
1  2  11  5  8
2  3  12  6  9
```

Replacing the Contents of a DataFrame

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3],'B': [4, 5, 6],'c':[7, 8, 9]})

# Replace the contents of column 'A' with new values
df['A'] = [10, 20, 30]

# Display updated DataFrame
print("DataFrame after replacing column 'A':")
print(df)

# Replace the contents
df.replace({'A': {1: 100}, 'B': {6: 200}, 'c':{7:1000}}, inplace=True)
#df.replace({'A': 1, 'B': 6}, 100, inplace=True)

# Display updated DataFrame
print("DataFrame after replacement:")
print(df)
```

```
DataFrame after replacing column 'A':
    A  B  c
0  10  4  7
1  20  5  8
2  30  6  9
DataFrame after replacement:
    A    B     c
0  10    4  1000
1  20    5     8
2  30  200     9
```

Deleting Columns

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3],'B': [4, 5, 6],'C': [7, 8, 9]})

# Display the original DataFrame
print("Original DataFrame:", df, sep='\n')

# Delete columns 'A' and 'B'
df = df.drop(columns=['A', 'B'])

# Display updated DataFrame
print("DataFrame after deleting columns 'A' and 'B':")
print(df)
```

```
Original DataFrame:
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
DataFrame after deleting columns 'A' and 'B':
   C
0  7
1  8
2  9
```

Removing Rows from a DataFrame

Dropping DataFrame Rows by Index Values

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],'B': [4, 5, 6, 7, 8]})

# Display original DataFrame
print("Original DataFrame:")
print(df)

# Drop the row with index 3
result = df.drop(3)

# Display the result
print("\nAfter dropping the row at index 3:")
print(result)
```

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

After dropping the row at index 3:
   A  B
0  1  4
1  2  5
2  3  6
4  5  8
```

Removing Rows using Index Slicing

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],'B': [4, 5, 6, 7, 8]})

# Display original DataFrame
print("Original DataFrame:")
print(df)

# Drop the row using the index slicing
result = df.drop(df.index[2:4])

# Display the result
print("\nAfter dropping the row at 2 and 3:")
print(result)
```

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

After dropping the row at 2 and 3:
   A  B
0  1  4
1  2  5
4  5  8
```

Dropping Multiple Rows by Labels

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],'B': [4, 5, 6, 7, 8],
'C': [9, 10, 11, 12, 13]}, index=['r1', 'r2', 'r3', 'r4', 'r5'])

# Display original DataFrame
print("Original DataFrame:")
print(df)

# Drop the rows by row-labels
result = df.drop(['r1', 'r3'])

# Display the result
print("\nAfter dropping the rows:")
print(result)
```

```
Original DataFrame:
    A  B   C
r1  1  4   9
r2  2  5  10
r3  3  6  11
r4  4  7  12
r5  5  8  13

After dropping the rows:
    A  B   C
r2  2  5  10
r4  4  7  12
r5  5  8  13
```

Removing Rows Based on a Condition

```
import pandas as pd
# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],'B': [4, 5, 6, 7, 8],
'C': [90, 0, 11, 12, 13]}, index=['r1', 'r2', 'r3', 'r4', 'r5'])

# Display original DataFrame
print("Original DataFrame:")
print(df)

# Dropping rows where column 'C' contains 0
result = df[df["C"] != 0]

# Display the result
print("\nAfter dropping the row where 'C' has 0:")
print(result)
```

```
Original DataFrame:
    A  B   C
r1  1  4  90
r2  2  5   0
r3  3  6  11
r4  4  7  12
r5  5  8  13

After dropping the row where 'C' has 0:
    A  B   C
r1  1  4  90
r3  3  6  11
r4  4  7  12
r5  5  8  13
```

Arithmetic Operations on DataFrame with Scalar Value

Operation	Example with Operator	Description
Addition	df + 2	Adds 2 to each element of the DataFrame
Subtraction	df - 2	Subtracts 2 from each element
Multiplication	df * 2	Multiplies each element by 2
Division	df / 2	Divides each element by 2
Exponentiation	df ** 2	Raises each element to the power of 2
Modulus	df % 2	Finds the remainder when divided by 2
Floor Division	df // 2	Divides and floors the quotient

Arithmetic Operations Between Two DataFrames

```
import pandas as pd

# Create two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8]})
df2 = pd.DataFrame({'A': [10, 20, 30], 'B': [50, 60, 70]}, index=[1, 2, 3])

# Display the input DataFrames
print("DataFrame 1:\n", df1)
print("\nDataFrame 2:\n", df2)

# Perform arithmetic operations
print("\nAddition of Two DataFrames:\n", df1 + df2)
print("\nSubtraction of Two DataFrames:\n", df1 - df2)
print("\nMultiplication of Two DataFrames:\n", df1 * df2)
print("\nDivision of Two DataFrames:\n", df1 / df2)
```

Arithmetic Functions in Pandas

In addition to the above operators, Pandas provides various functions to perform arithmetic operations on Pandas Data structure, which can handle missing values efficiently and provides additional options for customization, like selecting the axis and specifying levels.

S.No	Function	Description
1	add(other[, axis, level, fill_value])	Element-wise addition (binary operator +).
2	sub(other[, axis, level, fill_value])	Element-wise subtraction (binary operator -).
3	mul(other[, axis, level, fill_value])	Element-wise multiplication (binary operator *).
4	div(other[, axis, level, fill_value])	Element-wise floating division (binary operator /).
5	truediv(other[, axis, level, ...])	Element-wise floating division (binary operator /).
6	floordiv(other[, axis, level, ...])	Element-wise integer division (binary operator //).
7	mod(other[, axis, level, fill_value])	Element-wise modulo operation (binary operator %).
8	pow(other[, axis, level, fill_value])	Element-wise exponential power (binary operator **).
9	dot(other)	Matrix multiplication with another DataFrame or array.
10	radd(other[, axis, level, fill_value])	Reverse element-wise addition.
11	rsub(other[, axis, level, fill_value])	Reverse element-wise subtraction.

12	rmul(other[, axis, level, fill_value])	Reverse element-wise multiplication.
13	rdiv(other[, axis, level, fill_value])	Reverse element-wise floating division.
14	rfloordiv(other[, axis, level, ...])	Reverse element-wise integer division.
15	rmod(other[, axis, level, fill_value])	Reverse element-wise modulo operation.
16	rpow(other[, axis, level, fill_value])	Reverse element-wise exponential power.

```
df1 = pd.DataFrame({'A': [10, 20], 'B': [30, 40]})
df2 = pd.DataFrame({'A': [1, 2]})  # 'B' is missing here

# Using +
print("Using +:\n", df1 + df2)
print("\n")
#using .add
print("using .add:\n", df1.add(df2, fill_value=0))
```

```
Using +:
     A   B
0  11 NaN
1  22 NaN


using .add:
     A     B
0  11  30.0
1  22  40.0
```

```python
import pandas as pd

df = pd.DataFrame({'A': [10, 20], 'B': [30, 40]})
s = pd.Series([1, 2], index=[0, 1])  # Aligns with rows

# Add Series to DataFrame row-wise (default behavior)
print(df.add(s, axis=0))

s2 = pd.Series([1, 100], index=['A', 'B'])  # Aligns with columns

# Add Series to DataFrame column-wise
print(df.add(s2, axis=1))
```

```
    A   B
0  11  31
1  22  42
    A    B
0  11  130
1  21  140
```

Pandas - IO Tools

The Pandas I/O API supports a wide variety of data formats. Here is a summary of supported formats and their corresponding reader and writer functions:

Format	Reader Function	Writer Function
Tabular Data	read_table()	NA
CSV	read_csv()	to_csv()
Fixed-Width Text File	read_fwf()	NA
Clipboard	read_clipboard()	to_clipboard()
Pickling	read_pickle()	to_pickle()
Excel	read_excel()	to_excel()
JSON	read_json()	to_json()
HTML	read_html()	to_html()
XML	read_xml()	to_xml()
LaTeX	NA	to_latex()
HDF5 Format	read_hdf()	to_hdf()
Feather	read_feather()	to_feather()
Parquet	read_parquet()	to_parquet()
ORC	read_orc()	to_orc()
SQL	read_sql()	to_sql()
Stata	read_stata()	to_stata()

Among these, the most frequently used functions for handling text files are ***read_csv()*** and ***read_table()***. Both convert flat files into DataFrame objects.

```
# Import StringIO to load a file-like object for reading CSV
from io import StringIO

# Create string representing CSV data
data = """S.No,Name,Age,City,Salary
1,Tom,28,Toronto,20000
2,Lee,32,HongKong,3000
3,Steven,43,Bay Area,8300
4,Ram,38,Hyderabad,3900"""

# Use StringIO to convert the string data into a file-like object
obj = StringIO(data)

# read CSV into a Pandas DataFrame
df = pd.read_csv(obj)

print(df)
```

```
   S.No    Name  Age       City  Salary
0     1     Tom   28    Toronto   20000
1     2     Lee   32   HongKong    3000
2     3  Steven   43   Bay Area    8300
3     4     Ram   38  Hyderabad    3900
```

Note: **`StringIO`** lets you treat a string like a file so that functions like **`pd.read_csv()`** can read data from it without needing an actual file.

```
import pandas as pd

# dictionary of lists
d = {'Car': ['BMW', 'Lexus', 'Audi', 'Mercedes', 'Jaguar', 'Bentley'],
'Date_of_purchase': ['2024-10-10', '2024-10-12', '2024-10-17', '2024-10-16', '2024-10-19', '2024-10-22']}

# creating dataframe from the above dictionary of lists
dataFrame = pd.DataFrame(d)
print("Original DataFrame:\n",dataFrame)

# write dataFrame to SalesRecords CSV file
dataFrame.to_csv("Output_written_CSV_File.csv")

# display the contents of the output csv
print("The output csv file written successfully...")
```

```
Original DataFrame:
         Car Date_of_purchase
0       BMW       2024-10-10
1     Lexus       2024-10-12
2      Audi       2024-10-17
3  Mercedes       2024-10-16
4    Jaguar       2024-10-19
5   Bentley       2024-10-22
The output csv file written successfully...
```