Python Pandas

What is Pandas?

👉Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

👉Pandas is a Python library that simplifies data analysis and manipulation. It provides easy-to-use data structures and functions for working with structured data. With Pandas, you can organize, clean, and analyze data effectively, making it a valuable tool for data-related tasks

Installation of Pandas

Step 1: Check Python Version

First, make sure you have Python installed on your system. Open a terminal or command prompt and run the following command to check the Python version:

python --version

Step 2: Open Command Prompt or Terminal

Open the Command Prompt (Windows) or Terminal (Mac/Linux) on your computer.

Step 3: Install Pandas Using pip

To install Pandas, use the pip package manager. Run the following command:

👉**pip install pandas**

Step 4: Check Pandas Version

To check the Pandas version, run the following command:

👉import pandas as pd

👉print(pd.__version__)

Import Pandas(Pandas as pd)

Once Pandas is installed, import it in your applications by adding the `import` keyword:

--Pandas is usually imported under the `pd` alias.

👉import pandas as pd

Python Pandas Data Structure

Python Pandas provides two main data structures: Series and DataFrame.

1. **Series:**

   👉A Series is a one-dimensional labeled array that can hold any data type. It is similar to a column in a spreadsheet or a single column of data in a NumPy array. The Series consists of two main components: the index and the data values.

import pandas as pd

```
# Creating a Series
data = [10, 20, 30, 40, 50]
s = pd.Series(data)
print(s)
```

Output

```
0   10
1   20
2   30
3   40
4   50
dtype: int64
```

2. **DataFrame:**

   👉A DataFrame is a two-dimensional labeled data structure with columns of potentially different data types. It is similar to a table in a relational database or a spreadsheet with rows and columns. It can be thought of as a collection of Series objects where each Series represents a column.

Series:

Example

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

```
# Create a Series from an array
data_array = np.array([10, 20, 30, 40, 50])
series_array = pd.Series(data_array)
print("Series from array:")
print(series_array)
print()

# Create a Series from a list
data_list = [10, 20, 30, 40, 50]
series_list = pd.Series(data_list)
print("Series from list:")
print(series_list)
print()

# Create a Series from a tuple
data_tuple = (10, 20, 30, 40, 50)
series_tuple = pd.Series(data_tuple)
print("Series from tuple:")
```

```
print(series_tuple)
print()

# Create a Series from a dictionary
data_dict = {'A': 10, 'B': 20, 'C': 30, 'D': 40, 'E': 50}
series_dict = pd.Series(data_dict)
print("Series from dictionary:")
print(series_dict)
print()

#If we take the scalar values, then the index must be provided.
# Create a Series with a scalar value
scalar_value = 5
series_scalar = pd.Series(scalar_value, index=['a', 'b', 'c', 'd', 'e'])
print("Series with scalar value:")
print(series_scalar)

Series from array:
0   10
```

```
1   20
2   30
3   40
4   50
dtype: int64
```

Series from list:

```
0   10
1    20
2    30
3    40
4    50
dtype: int64
```

Series from tuple:

```
0   10
1   20
2   30
3   40
```

```
4   50
dtype: int64

Series from dictionary:
A   10
B   20
C   30
D   40
E   50
dtype: int64

Series with scalar value:
a   5
b   5
c   5
d   5
e   5
dtype: int64
```

 # Create a Series from a set

To create a Series from a set, you need to provide an ordered collection, such as a list or an array, that maintains the desired order of the elements.
Sets do not have an inherent order, so they cannot be directly converted into a Series.
If you have a set and want to create a Series, you need to convert the set into an ordered collection, such as a list or an array, before creating the Series.

```
import pandas as pd
data_set = {10, 20, 30, 40, 50}
data_list = list(data_set)

series_set = pd.Series(data_list)
print("Series from set:")
print(series_set)
```

```
Series from set:
0   40
1   10
2   50
3   20
4   30
dtype: int64
```

Series object attributes

👉The Pandas Series object has several attributes that provide information about the series. Here's a description of some commonly used Series attributes:

- ✅ **Series.index:**This attribute defines the index of the Series, which represents the labels for each element in the Series.
- ✅ **Series.shape:** It returns a tuple representing the shape of the Series. For a one-dimensional Series, the shape tuple will have only one element representing the length of the Series.
- ✅ **Series.dtype:** It returns the data type of the elements in the Series.
- ✅ **Series.size:** It returns the number of elements in the Series.
- ✅ **Series.empty:** It returns a boolean value indicating whether the Series is empty or not. Returns True if the Series is empty, and False otherwise.
- ✅ **Series.hasnans:** It returns a boolean value indicating whether there are any NaN (missing) values in the Series. Returns True if there are NaN values, and False otherwise.
- ✅ **Series.nbytes:** It returns the number of bytes consumed by the Series data.
- ✅ **Series.ndim:** It returns the number of dimensions in the Series data. For a Series, the ndim attribute will always be 1.

- ✅ **Series.itemsize:** It returns the size in bytes of each individual element in the Series.

Example

```
import pandas as pd
import numpy as np

data = [10, 20, 30, 40, 50]
series = pd.Series(data)

print("Series:", series)
print("Index:", series.index)
print("Shape:", series.shape)
print("Data Type:", series.dtype)
print("Size:", series.size)
print("Is Empty:", series.empty)
print("Has NaNs:", series.hasnans)
print("Number of Bytes:", series.nbytes)
print("Number of Dimensions:", series.ndim)
print("Item Size:", series.itemsize)
```

Output: 👇

```
Series:
0   10
```

```
1   20
2   30
3   40
4   50
dtype: int64
Index: RangeIndex(start=0, stop=5, step=1)
Shape: (5,)
Data Type: int64
Size: 5
Is Empty: False
Has NaNs: False
Number of Bytes: 40
Number of Dimensions: 1
Item Size: 8
```

```
import pandas as pd

# Create a Series
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
```

```
# Series.index
print("Index:", series.index)

# Series.shape
print("Shape:", series.shape)

# Series.dtype
print("Data Type:", series.dtype)

# Series.size
print("Size:", series.size)

# Series.empty
print("Is Empty:", series.empty)

# Series.hasnans
print("Has NaNs:", series.hasnans)

# Series.nbytes
print("Number of Bytes:", series.nbytes)

# Series.ndim
print("Number of Dimensions:", series.ndim)
```

```
# Series.itemsize
print("Item Size:", series.itemsize)
```

Output: 👇

```
Index: RangeIndex(start=0, stop=5, step=1)
Shape: (5,)
Data Type: int64
Size: 5
Is Empty: False
Has NaNs: False
Number of Bytes: 40
Number of Dimensions: 1
Item Size: 8
```

In this example, we create a Series series from a list [10, 20, 30, 40, 50]. We then print the various attributes of the Series.

- **Series.index** ➡ returns a RangeIndex object representing the index labels of the Series, in this case, a default range index from 0 to 4.
- **Series.shape** ➡ returns a tuple (5,) indicating that the Series has 5 elements.
- **Series.dtype** ➡ returns int64 as the data type of the Series elements.
- **Series.size** ➡ returns 5 indicating the number of elements in the Series.

- **Series.empty** ➡ returns False since the Series is not empty.
- **Series.hasnans** ➡ returns False since there are no NaN (missing) values in the Series.
- **Series.nbytes** ➡ returns 40 indicating the number of bytes consumed by the Series data.
- **Series.ndim** ➡ returns 1 indicating that the Series is one-dimensional.
- **Series.itemsize** ➡ returns 8 indicating the size in bytes of each individual element in the Series, which corresponds to the int64 data type.

Series object attributes

To retrieve specific information from a Series object in Pandas, you can use the following attributes and methods:

Retrieving Index array and Data array:

- ✅ **Index array:**
  ➡ You can access the index labels of a Series object using the Series.index attribute. It returns an Index object containing the index labels.

- ✅ **Data array:**
  ➡ You can access the data values of a Series object using the Series.values attribute. It returns a NumPy array containing the data values.

Example:

```
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data)

index_array = series.index
```

```
data_array = series.values

print("Index Array:", index_array)
print("Data Array:", data_array)
```

Output: 👇

```
Index Array: RangeIndex(start=0, stop=5, step=1)
Data Array: [10 20 30 40 50]
```

Retrieving Types (dtype) and Size of Type (itemsize):

- ✅ **Types (dtype):**
  ➡️ You can retrieve the data type of the elements in a Series object using the Series.dtype attribute. It returns the dtype object representing the data type.

- ✅ **Size of Type (itemsize):**
  ➡️ You can retrieve the size in bytes of each individual element in the Series using the Series.dtype.itemsize attribute.

Example:

```
import pandas as pd
data = [10, 20, 30, 40, 50]
series = pd.Series(data)

dtype = series.dtype
itemsize = series.dtype.itemsize
```

```
print("Data Type:", dtype)
print("Item Size:", itemsize)
```

Output: 👇

```
Data Type: int64
Item Size: 8
```

Retrieving Shape, Dimension, Size, and Number of Bytes:

- ✅ **Shape:**
  ➡️ You can retrieve the shape of a Series object using the Series.shape attribute. It returns a tuple representing the shape of the data.

- ✅ **Dimension:**
  ➡️ You can retrieve the number of dimensions in the Series data using the Series.ndim attribute. For a Series, the ndim attribute will always be 1.
- ✅ **Size:**
  ➡️ You can retrieve the number of elements in the Series using the Series.size attribute.

- ✅ **Number of Bytes:**
  ➡️ You can retrieve the total number of bytes consumed by the Series data using the Series.nbytes attribute.

Example:

```
import pandas as pd
```

```
data = [10, 20, 30, 40, 50]
series = pd.Series(data)

shape = series.shape
dimension = series.ndim
size = series.size
nbytes = series.nbytes

print("Shape:", shape)
print("Dimension:", dimension)
print("Size:", size)
print("Number of Bytes:", nbytes)
```

Output: 👇

```
Shape: (5,)
Dimension: 1
Size: 5
Number of Bytes: 40
```

Checking Emptiness:

- ✅ **Empty:**
  ➡️ You can check whether a Series object is empty or not using the Series.empty attribute. It returns a boolean value, True if the Series is empty and False otherwise.

Example:

```
import pandas as pd

empty_series = pd.Series([])
nonempty_series = pd.Series([10, 20, 30])

print("Empty Series:", empty_series.empty)
print("Non-empty Series:", nonempty_series.empty)
```

Output: 👇

```
Empty Series: True
Non-empty Series: False
```

In this example, `empty_series` is an empty Series, so `empty_series.empty` returns `True`. On the other hand, `nonempty_series` is a non-empty Series, so `nonempty_series.empty` returns `False`.

Checking Presence of NaNs:

- ✅ **Has NaNs:**
  ➡️ You can check whether a Series object contains any NaN (missing) values using the Series.hasnans attribute. It returns a boolean value, True if there are NaN values in the Series and False otherwise.

```
import pandas as pd
```

```
import numpy as np

data = [10, 20, np.nan, 40, np.nan]
series = pd.Series(data)

print("Has NaNs:", series.hasnans)
```

Output: 👇

```
Has NaNs: True
```

In this example, the Series `series` contains NaN values, so `series.hasnans` returns `True`.

Series Functions

Series.map():

➡️ This function is used to map the values of a Series based on a provided mapping or a callable function. It takes a dictionary, Series, or a function as input and returns a new Series with the mapped values.

Example:

```
mport pandas as pd

data = {'A': 'Apple', 'B': 'Banana', 'C': 'Cherry'}
series = pd.Series(['A', 'B', 'C'])
mapped_series = series.map(data)
```

```
print("Mapped Series:")
print(mapped_series)
```

Output: 👇

```
Mapped Series:
0    Apple
1    Banana
2    Cherry
dtype: object
```

✅ `Series.std()`:

➡️ This function calculates the standard deviation of the values in a Series. It returns a scalar value representing the standard deviation.

```
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data)
std = series.std()

print("Standard Deviation:", std)
```

Output: 👇

```
Standard Deviation: 15.811388300841896
```

✅ `Series.to_frame()`:

➡️ This function converts a Series object to a DataFrame. It returns a new DataFrame where the Series becomes a single column, and the index of the Series becomes the DataFrame index.

```
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data)
df = series.to_frame()

print("DataFrame:")
print(df)
```

Output: 👇

```
DataFrame:
    0
0  10
1  20
2  30
3  40
4  50
```

✅ `Series.value_counts()`:

➡️ This function returns a Series containing counts of unique values in the original Series. The resulting Series is sorted in descending order by default.

```
import pandas as pd

data = ['A', 'B', 'A', 'C', 'B', 'B']
series = pd.Series(data)
value_counts = series.value_counts()
print("Value Counts:")
print(value_counts)
```

Output: 👇

```
Value Counts:
B   3
A   2
C   1
dtype: int64
```

Example

```
import pandas as pd
import numpy as np
# Example 1: Mapping values using a dictionary
a = pd.Series(['Java', 'C', 'C++', np.nan])
```

```
mapped_series = a.map({'Java': 'Core'})
print(mapped_series)
```

```
0    Core
1     NaN
2     NaN
3     NaN
dtype: object
```

 # Example 2: Mapping values using a format string

```
a = pd.Series(['Java', 'C', 'C++', np.nan])
mapped_series = a.map('I like {}'.format, na_action='ignore')
print(mapped_series)
```

```
0    I like Java
1       I like C
2     I like C++
3            NaN
dtype: object
```

 # Example 3: Mapping values using a format string and handling missing values

```
a = pd.Series(['Java', 'C', 'C++', np.nan])
mapped_series = a.map('I like {}'.format)
mapped_series_ignore_na = a.map('I like {}'.format, na_action='ignore')
print(mapped_series)
print(mapped_series_ignore_na)
```

The `na_action='ignore'` parameter is specified to handle missing values. It tells the `map()` function to ignore any missing values (`np.nan`) in the Series and not perform any transformation on them.

```
0    I like Java
```

```
1       I like C
2     I like C++
3     I like nan
dtype: object
0    I like Java
1       I like C
2     I like C++
3            NaN
dtype: object
```

Python Pandas DataFrame

👉A DataFrame is a two-dimensional data structure in pandas that can store and manipulate data in a tabular form. It consists of rows and columns, similar to a spreadsheet or a SQL table.

There are several ways to create a DataFrame in pandas:

✅Using a dictionary:

👉You can create a DataFrame by passing a dictionary of lists, arrays, or series. The keys of the dictionary represent the column names, and the values represent the data for each column. The columns must be of the same length.

Example:

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'London', 'Paris']
}
```

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

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

✅ Using lists:

👉You can create a DataFrame from a list of lists or a list of arrays. Each inner list or array represents a row in the DataFrame.

Example:

```
import pandas as pd

data = [
   ['Alice', 25, 'New York'],
   ['Bob', 30, 'London'],
   ['Charlie', 35, 'Paris']
]
df = pd.DataFrame(data)
print(df)
```

```
        0   1         2
0    Alice  25  New York
1      Bob  30    London
2  Charlie  35     Paris
```

```
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

```
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'], index=['A', 'B', 'C'])
print(df)
```

```
      Name  Age      City
A    Alice   25  New York
B      Bob   30    London
C  Charlie   35     Paris
```

✅ Using NumPy ndarrays:

👉You can create a DataFrame from a 2D NumPy ndarray. Each row in the ndarray corresponds to a row in the DataFrame.

Example:

```
import pandas as pd
import numpy as np

data = np.array([
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'London'],
    ['Charlie', 35, 'Paris']
])

df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

```
    Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

✅**Using Series:**

👉You can create a DataFrame from a dictionary of Series. Each Series represents a column in the DataFrame.

Example:

```
import pandas as pd

data = {
   'Name': pd.Series(['Alice', 'Bob', 'Charlie']),
   'Age': pd.Series([25, 30, 35]),
   'City': pd.Series(['New York', 'London', 'Paris'])
}
df = pd.DataFrame(data)
print(df)
```

A DataFrame is a two-dimensional data structure in pandas that can store and manipulate data in a tabular form. It consists of rows and columns, similar to a spreadsheet or a SQL table.

There are several ways to create a DataFrame in pandas:

1. Using a dictionary:

You can create a DataFrame by passing a dictionary of lists, arrays, or series. The keys of the dictionary represent the column names, and the values represent the data for each column. The columns must be of the same length.

2. Using lists:
   You can create a DataFrame from a list of lists or a list of arrays. Each inner list or array represents a row in the DataFrame.

3. Using NumPy ndarrays:
   You can create a DataFrame from a 2D NumPy ndarray. Each row in the ndarray corresponds to a row in the DataFrame.

4. Using Series:
   You can create a DataFrame from a dictionary of Series. Each Series represents a column in the DataFrame.

Once you have created a DataFrame, you can perform various operations on it. Here are some common operations:

- Column Selection: You can select one or more columns from a DataFrame using indexing or by specifying the column names.
- Column Addition: You can add new columns to a DataFrame by assigning values to them. The new columns can be based on existing columns or computed from them.
- Column Deletion: You can delete one or more columns from a DataFrame using the del keyword or the drop() function.
- Row Selection: You can select specific rows from a DataFrame based on conditions or by specifying row indices.
- Row Addition: You can add new rows to a DataFrame using the append() function or by creating a new DataFrame and concatenating it with the original DataFrame.

- Row Deletion: You can delete one or more rows from a DataFrame using the drop() function with the appropriate row indices.

Here are some common operations that you can perform on a DataFrame along with examples:

1. **Column Selection:**
   You can select one or more columns from a DataFrame using indexing or by specifying the column names.
   Example:

```
import pandas as pd
data = {
  'Name': ['Alice', 'Bob', 'Charlie'],
  'Age': [25, 30, 35],
  'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)
```

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

✅ **# Selecting a single column**

```
name_column = df['Name']
print(name_column)
```

```
0      Alice
1        Bob
2    Charlie
```

```
Name: Name, dtype: object
```

✅ # Selecting multiple columns

```
name_age_columns = df[['Name', 'Age']]
print(name_age_columns)
```

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

✅ Column Addition:

You can add new columns to a DataFrame by assigning values to them. The new columns can be based on existing columns or computed from them.

```
import pandas as pd
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)

# Adding a new column based on an existing column
df['Upper_Name'] = df['Name'].str.upper()
print(df)
```

```
      Name  Age      City Upper_Name
0    Alice   25  New York      ALICE
1      Bob   30    London        BOB
2  Charlie   35     Paris    CHARLIE
```

 # Adding a new column with computed values

```
df['Birth_Year'] = pd.datetime.now().year - df['Age']
print(df)
```

```
      Name  Age      City Upper_Name  Birth_Year
0    Alice   25  New York      ALICE        1998
1      Bob   30    London        BOB        1993
2  Charlie   35     Paris    CHARLIE        1988
```

✅ Column Deletion:

You can delete one or more columns from a DataFrame using the `del` keyword or the `drop()` function.

Example:

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)
```

```
      Name  Age      City
0    Alice   25  New York
```

```
1      Bob   30    London
2  Charlie   35     Paris
```

```
# Deleting a column using the del keyword
del df['Age']
print(df)
```

```
      Name      City
0    Alice  New York
1      Bob    London
2  Charlie     Paris
```

```
# Deleting a column using the drop() function
df = df.drop('City', axis=1)
print(df)
```

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

✅ Row Selection:

You can select specific rows from a DataFrame based on conditions or by specifying row indices.

Example:

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
```

```
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)

# Selecting rows based on a condition
adults = df[df['Age'] >= 30]
print(adults)
```

```
      Name  Age
1      Bob   30
2  Charlie   35
```

```
# Selecting rows by specifying row indices
specific_rows = df.loc[[0, 2]]
print(specific_rows)
```

✅ Row Addition:

You can add new rows to a DataFrame using the `append()` function or by creating a new DataFrame and concatenating it with the original DataFrame

Example:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
```

```
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)

# Adding a new row using append() function
new_row = pd.DataFrame({'Name': ['Dave'], 'Age': [28], 'City': ['Berlin']})
df = df.append(new_row, ignore_index=True)
print(df)
```

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
3     Dave   28    Berlin
```

```
# Adding new rows by concatenating DataFrames
new_rows = pd.DataFrame([
    {'Name': 'Emily', 'Age': 32, 'City': 'Sydney'},
    {'Name': 'Frank', 'Age': 27, 'City': 'Tokyo'}
])
df = pd.concat([df, new_rows], ignore_index=True)
print(df)
```

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

```
3     Dave   28    Berlin
4    Emily   32    Sydney
5    Frank   27     Tokyo
```

Row Deletion:

You can delete one or more rows from a DataFrame using the `drop()` function with the appropriate row indices.

```
import pandas as pd
# Create a sample DataFrame
data = {'Name': ['John', 'Jane', 'Alice', 'Bob'],
    'Age': [25, 30, 35, 40]}
df = pd.DataFrame(data)
print(df)
```

```
    Name  Age
0   John   25
1   Jane   30
2  Alice   35
3    Bob   40
```

Deleting rows using the drop() function

```
# Drop rows with labels 0 and 2
df = df.drop([0, 2], axis=0)# [0, 2] is the list of row labels you want to drop. In this case, it specifies that you want to drop the rows with labels 0 and 2.
print(df)
```

```
   Name  Age
1  Jane   30
3   Bob   40
```

the rows with labels 0 (John) and 2 (Alice) are dropped, and the resulting DataFrame contains only the remaining rows 1 (Jane) and 3 (Bob).

DataFrame Functions

✅ Pandas DataFrame.append():

- 👉 Description: Add the rows of another DataFrame to the end of the given DataFrame.
- Example:

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})

df_appended = df1.append(df2)
print(df_appended)
```

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

✅ Pandas DataFrame.apply():

- 👉 Description: Apply a function to every single value of the DataFrame.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
def square(x):
  return x ** 2
df_squared = df.apply(square)
```

```
print(df_squared)
```

```
   A   B
0  1  16
1  4  25
2  9  36
```

✅ Pandas DataFrame.assign():

- 👉 Description: Add new columns into a DataFrame.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df_new = df.assign(C=[7, 8, 9], D=[10, 11, 12])
print(df_new)
```

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

✅ Pandas DataFrame.astype():

- 👉 Description: Cast the DataFrame to a specified dtype.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df_casted = df.astype({'A': float})
print(df_casted)
```

✅ Pandas DataFrame.concat():

- 👉 Description: Perform concatenation operation along an axis in the DataFrame.
- Example:

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})
df_concatenated = pd.concat([df1, df2])
print(df_concatenated)
```

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

✅ Pandas DataFrame.count():

- 👉 Description: Count the number of non-NA cells for each column or row.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
column_counts = df.count()
print(column_counts)


row_counts = df.count(axis=1)
```

```
print(row_counts)
```

```
A    2
B    2
dtype: int64
0    2
1    1
2    1
dtype: int64
```

✅ Pandas DataFrame.astype():

- 👉 Description: Cast the DataFrame to a specified dtype.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df_casted = df.astype({'A': float})
print(df_casted)
```

✅ Pandas DataFrame.describe():

- 👉 Description: Calculate statistical data like percentile, mean, and standard deviation of the numerical values in the DataFrame.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})
description = df.describe()
print(description)
```

```
count  5.000000
```

```
mean    3.000000
std     1.581139
min     1.000000
25%     2.000000
50%     3.000000
75%     4.000000
max     5.000000
```

✅ Pandas DataFrame.drop_duplicates():

- 👉 Description: Remove duplicate values from the DataFrame.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 2, 3, 4, 4, 5]})
df_unique = df.drop_duplicates()
print(df_unique)
```

✅ Pandas DataFrame.groupby():

- Description: Split the data into various groups based on a column or multiple columns.
- Example:

```
import pandas as pd

df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', 'foo'],
                   'B': ['one', 'one', 'two', 'two', 'one'],
                   'C': [1, 2, 3, 4, 5]})

grouped = df.groupby('A')
print(grouped.get_group('foo'))
```

✅ Pandas DataFrame.head():

- 👉 Description: Return the first n rows of the DataFrame based on position.
- Example:

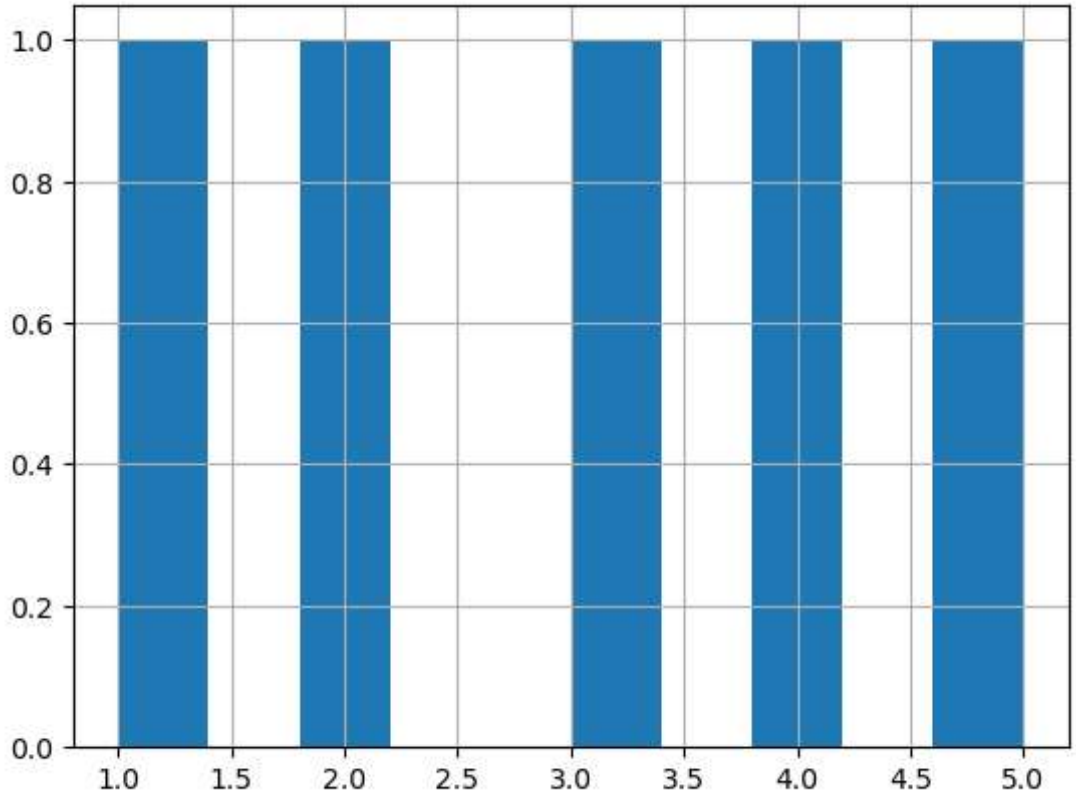
```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})
first_three_rows = df.head(3)
print(first_three_rows)
```

```
   A
0  1
1  2
2  3
```

✅ Pandas DataFrame.hist():

- 👉 Description: Divide the values within a numerical variable into "bins" and create a histogram.
- Example:

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})
df['A'].hist()
plt.show()
```



✅ Pandas DataFrame.iterrows():

- 👉 Description: Iterate over the rows of the DataFrame as (index, series) pairs.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})
for index, row in df.iterrows():
  print(f"Index: {index}, Row: {row}")
```

```
Index: 0, Row: A    1
B    a
```

```
Name: 0, dtype: object
Index: 1, Row: A    2
B    b
Name: 1, dtype: object
Index: 2, Row: A    3
B    c
Name: 2, dtype: object
```

✅ Pandas DataFrame.mean():

- 👉 Description: Return the mean of the values for the requested axis (column or row).
- Example:

```
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
column_mean = df.mean()
print(column_mean)
row_mean = df.mean(axis=1)
print(row_mean)
```

```
A    2.0
B    5.0
dtype: float64
0    2.5
1    3.5
2    4.5
dtype: float64
```

✅ Pandas DataFrame.melt():

- 👉 Description: Unpivot the DataFrame from a wide format to a long format.

- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print(df )
```

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

```
melted_df = df.melt()
print(melted_df)
```

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

✅ Pandas DataFrame.merge():

- 👉 Description: Merge two datasets together into one based on common columns or indices.
- Example:

```
import pandas as pd
left_df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})
right_df = pd.DataFrame({'A': [4, 5, 6], 'C': ['x', 'y', 'z']})
merged_df = pd.merge(left_df, right_df, on='A')
print(merged_df)
```

```
Empty DataFrame
Columns: [A, B, C]
Index: []
```

✅ Pandas DataFrame.pivot_table():

- Description: Aggregate data with calculations such as sum, count, average, max, and min, and create a pivot table.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', 'foo'],
                   'B': ['one', 'one', 'two', 'two', 'one'],
                   'C': [1, 2, 3, 4, 5]})
pivot_table = df.pivot_table(values='C', index='A', columns='B', aggfunc='sum')
print(pivot_table)
```

```
B    one  two
A
bar    2    4
foo    6    3
```

✅ Pandas DataFrame.query():

- Description: Filter the DataFrame based on a boolean expression.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4, 5], 'B': ['foo', 'bar', 'foo', 'baz', 'qux']})
filtered_df = df.query('A > 2')
print(filtered_df)
```

✅ Pandas DataFrame.shift():

- Description: Shift column values or subtract the column value with the previous row value.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})
shifted_df = df['A'].shift(1)
print(shifted_df)
```

✅ Pandas DataFrame.sort():

- 👉 Description: Sort the DataFrame based on one or more columns.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [3, 2, 1], 'B': ['c', 'b', 'a']})
sorted_df = df.sort_values(by='A')
print(sorted_df)
```

```
   A  B
2  1  a
1  2  b
0  3  c
```

✅ Pandas DataFrame.sum():

- 👉 Description: Return the sum of the values for the requested axis (column or row).
- Example:

```
import pandas as pd
```

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

column_sum = df.sum()
print(column_sum)

row_sum = df.sum(axis=1)
print(row_sum)
```

✅ **Pandas DataFrame.to_excel():**

- 👉 Description: Export the DataFrame to an Excel file.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df.to_excel('output.xlsx', index=False)
```

✅ **Pandas DataFrame.transpose():**

- 👉 Description: Transpose the index and columns of the DataFrame.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
transposed_df = df.transpose()
print(transposed_df)
```

✅ Pandas DataFrame.where():

- 👉 Description: Check the DataFrame for one or more conditions and replace values where the condition is False.
- Example:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
condition = df['A'] > 2
filtered_df = df.where(condition, other=0)
print(filtered_df)
```

✅ **Merge:**

👉 The merge() function in pandas is used to combine two or more DataFrames based on a common column or index. It performs database-style join operations (e.g., inner join, outer join, left join, right join) to combine the DataFrames. The result is a new DataFrame that contains the combined data.

```
import pandas as pd
# Create the first DataFrame
data1 = {'Name': ['John', 'Jane', 'Alice'],
        'Age': [25, 30, 35]}
df1 = pd.DataFrame(data1)

# Create the second DataFrame
data2 = {'Name': ['Alice', 'Bob', 'Charlie'],
        'City': ['New York', 'London', 'Paris']}
df2 = pd.DataFrame(data2)
```

```
    # Merge the DataFrames based on the 'Name' column
    merged_df = pd.merge(df1, df2, on='Name')
    print(merged_df)
```

Output 👇

```
  Name  Age     City
0 Alice   35  New York
```

✅ **Join:**

👉 The join() function in pandas is used to join two or more DataFrames horizontally based on their index or columns. It aligns the DataFrames based on the specified index or column and combines them. It is similar to merge but supports only the left join operation by default.

```
import pandas as pd
# Create the first DataFrame
data1 = {'Name': ['John', 'Jane', 'Alice'],
      'Age': [25, 30, 35]}
df1 = pd.DataFrame(data1)

# Create the second DataFrame
data2 = {'City': ['New York', 'London', 'Paris']}
df2 = pd.DataFrame(data2, index=['John', 'Jane', 'Alice'])
```

```
# Join the DataFrames based on the index
joined_df = df1.join(df2)
print(joined_df)
```

Output 👇

```
Name  Age     City
0  John   25  New York
1  Jane   30    London
2  Alice  35     Paris
```

✅ **Concatenate:**

👉 The concat() function in pandas is used to concatenate two or more DataFrames vertically or horizontally along a particular axis. It stacks the DataFrames on top of each other (vertically) or side by side (horizontally). It is useful when you want to combine DataFrames without performing any column or index-based matching.

```
import pandas as pd
# Create the first DataFrame
data1 = {'Name': ['John', 'Jane'],
        'Age': [25, 30]}
df1 = pd.DataFrame(data1)
# Create the second DataFrame
data2 = {'Name': ['Alice'],
        'Age': [35]}
df2 = pd.DataFrame(data2)
```

```
# Concatenate the DataFrames vertically
concatenated_df = pd.concat([df1, df2])
print(concatenated_df)
```

Output 👇

```
  Name  Age
0  John   25
1  Jane   30
0  Alice  35
```

```
import pandas as pd
# Create two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]})

# Merge example
merged_df = pd.merge(df1, df2, left_on='A', right_on='C', how='inner')
print("Merged DataFrame:")
print(merged_df)
print()

# Join example
joined_df = df1.join(df2, lsuffix='_left', rsuffix='_right')
print("Joined DataFrame:")
```

```
print(joined_df)
print()

# Concatenate example
concatenated_df = pd.concat([df1, df2], axis=0)
print("Concatenated DataFrame (Vertical):")
print(concatenated_df)
print()
concatenated_df = pd.concat([df1, df2], axis=1)
print("Concatenated DataFrame (Horizontal):")
print(concatenated_df)
```

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

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

Concatenated DataFrame (Vertical):

```
   A  B   C    D
0  1  4  NaN  NaN
1  2  5  NaN  NaN
2  3  6  NaN  NaN
0  NaN NaN  7.0  10.0
1  NaN NaN  8.0  11.0
2  NaN NaN  9.0  12.0
```

Concatenated DataFrame (Horizontal):

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

Pandas Function

```
import pandas as pd
# Create a DataFrame
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})
```

```
# Pandas DataFrame.append()
appended_df = df1.append(df2)
print("Appended DataFrame:")
print(appended_df)

# Pandas DataFrame.apply()
def square(x):
   return x ** 2
applied_df = df1.apply(square)
print("Applied DataFrame:")
print(applied_df)

# Pandas DataFrame.assign()
df1.assign(C=[7, 8, 9])
print("DataFrame with new column:")
print(df1)

# Pandas DataFrame.astype()
df1.astype({'A': float})
print("DataFrame with specified dtype:")
print(df1)
```

```
# Pandas DataFrame.concat()
concatenated_df = pd.concat([df1, df2])
print("Concatenated DataFrame:")
print(concatenated_df)

# Pandas DataFrame.count()
column_count = df1.count()
print("Column Count:")
print(column_count)

# Pandas DataFrame.describe()
description = df1.describe()
print("DataFrame Description:")
print(description)

# Pandas DataFrame.drop_duplicates()
deduplicated_df = df1.drop_duplicates()
print("Deduplicated DataFrame:")
print(deduplicated_df)

# Pandas DataFrame.groupby()
grouped_df = df1.groupby('A').sum()
print("Grouped DataFrame:")
```

```
print(grouped_df)

# Pandas DataFrame.head()
first_two_rows = df1.head(2)
print("First two rows of DataFrame:")
print(first_two_rows)

# Pandas DataFrame.hist()
df1['A'].hist(bins=3)
plt.show()

# Pandas DataFrame.iterrows()
for index, row in df1.iterrows():
    print(f"Index: {index}, Row: {row}")

# Pandas DataFrame.mean()
mean_value = df1.mean()
print("Mean Value:")
print(mean_value)

# Pandas DataFrame.melt()
melted_df = df1.melt()
print("Melted DataFrame:")
```

```
print(melted_df)

# Pandas DataFrame.merge()
merged_df = pd.merge(df1, df2, on='A')
print("Merged DataFrame:")
print(merged_df)

# Pandas DataFrame.pivot_table()
pivot_table = df1.pivot_table(values='B', index='A', aggfunc='mean')
print("Pivot Table:")
print(pivot_table)

# Pandas DataFrame.query()
filtered_df = df1.query('A > 2')
print("Filtered DataFrame:")
print(filtered_df)

# Pandas DataFrame.sample()
sampled_df = df1.sample(n=2)
print("Sampled DataFrame:")
print(sampled_df)

# Pandas DataFrame.shift()
```

```
shifted_df = df1['A'].shift(1)
print("Shifted DataFrame:")
print(shifted_df)

# Pandas DataFrame.sort()
sorted_df = df1.sort_values(by='A')
print("Sorted DataFrame:")
print(sorted_df)

# Pandas DataFrame.sum()
column_sum = df1.sum()
print("Column Sum:")
print(column_sum)

# Pandas DataFrame.to_excel()
df1.to_excel('output.xlsx', index=False)

# Pandas DataFrame.transpose()
transposed_df = df1.transpose()
print("Transposed DataFrame:")
print(transposed_df)

# Pandas DataFrame.where()
```

```
condition = df1['A'] > 2
filtered_df = df1.where(condition, other=0)
print("Filtered DataFrame:")
print(filtered_df)
```

 # Pandas DataFrame.rename() & DataFrame.fillna()

```
import pandas as pd
import numpy as np

# Create a DataFrame with missing values
df = pd.DataFrame({'A': [1, 2, np.nan, 4],
                   'B': [5, np.nan, 7, 8],
                   'C': [9, 10, 11, np.nan]})

print("Original DataFrame:")
print(df)
print()

# Rename columns
renamed_df = df.rename(columns={'A': 'Column1', 'B': 'Column2', 'C': 'Column3'})
print("Renamed DataFrame:")
print(renamed_df)
print()

# Fill missing values with 0
filled_df = df.fillna(0)
print("DataFrame with filled values:")
print(filled_df)
print()
```

```
# Forward fill missing values
forward_filled_df = df.fillna(method='ffill')
print("DataFrame with forward filled values:")
print(forward_filled_df)
```

```
Original DataFrame:
    A   B    C
0  1.0  5.0  9.0
1  2.0  NaN  10.0
2  NaN  7.0  11.0
3  4.0  8.0  NaN

Renamed DataFrame:
   Column1  Column2  Column3
0      1.0      5.0      9.0
1      2.0      NaN     10.0
2      NaN      7.0     11.0
3      4.0      8.0      NaN

DataFrame with filled values:
    A   B    C
0  1.0  5.0  9.0
1  2.0  0.0  10.0
2  0.0  7.0  11.0
3  4.0  8.0  0.0

DataFrame with forward filled values:
    A   B    C
```

```
0 1.0 5.0  9.0
1 2.0 5.0 10.0
2 2.0 7.0 11.0
3 4.0 8.0 11.0
```

Pandas Time Series:

Pandas provides powerful tools for working with time series data. It includes various functionalities for indexing, slicing, and manipulating time series data. Time series data is represented as a Series or DataFrame with a DateTimeIndex, which allows easy manipulation and analysis of time-based data.

Pandas Datetime:

Pandas datetime module provides classes and functions for working with date and time data. The key class is Timestamp, which represents a specific moment in time. The DatetimeIndex class is used to index Pandas Series or DataFrame with timestamps. Some commonly used functions and methods for working with dates and times are:

- **pd.to_datetime():** 👉 Converts a string or an object to a pandas Timestamp object.
- **pd.date_range():** 👉 Generates a range of dates or timestamps.
- **pd.Timestamp():** 👉 Creates a Timestamp object from a string or numeric value.
- **DatetimeIndex.year/month/day/hour/minute/second:** 👉 Extracts specific components of a datetime index.

```
import pandas as pd
# Convert a string to a pandas Timestamp
date_str = '2023-06-08'
date = pd.to_datetime(date_str)
print(date)
# Output: 2023-06-08 00:00:00
```

```
# Generate a range of dates
date_range = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
print(date_range)
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06', '2023-01-07', '2023-01-08',
               '2023-01-09', '2023-01-10',
               ...
               '2023-12-22', '2023-12-23', '2023-12-24', '2023-12-25',
               '2023-12-26', '2023-12-27', '2023-12-28', '2023-12-29',
               '2023-12-30', '2023-12-31'],
              dtype='datetime64[ns]', length=365, freq='D')
```

```
# Create a Timestamp object
timestamp = pd.Timestamp('2023-06-08 12:00:00')
print(timestamp.year)
# Output: 2023
```

Pandas Time Offset:

Pandas provides time offset aliases that represent common time intervals. These aliases can be used for time frequency conversions, resampling, and other time-related operations. Some commonly used time offsets are:

- D: Day
- H: Hour
- T or min: Minute
- S: Second
- M: Month end

- A: Year end

Example usage:

```
import pandas as pd
# Create a time offset of 1 hour
offset = pd.offsets.Hour()
print(offset)
# Output: <Hour>

# Add a time offset to a timestamp
timestamp = pd.Timestamp('2023-06-08 12:00:00')
new_timestamp = timestamp + pd.offsets.Hour(2)
print(new_timestamp)
# Output: 2023-06-08 14:00:00
```

Pandas Time Periods:

Pandas time periods represent fixed-length intervals, such as a day, month, or year. Time periods can be used to aggregate and group time series data. Some commonly used functions and methods for working with time periods are:

- ✅ **pd.Period():** Creates a Period object from a string or numeric value.
- ✅ **pd.period_range():** Generates a range of time periods.
- ✅ **Period.asfreq():** Changes the frequency of a time period.
- ✅ **PeriodIndex:** Index of Period objects.

Example usage:

```
import pandas as pd
# Create a Period object
period = pd.Period('2023-06', freq='M')
print(period)
# Output: 2023-06

# Generate a range of time periods
period_range = pd.period_range(start='2023-01', end='2023-12', freq='M')
print(period_range)
```

```
PeriodIndex(['2023-01', '2023-02', '2023-03', '2023-04', '2023-05', '2023-06',
             '2023-07', '2023-08', '2023-09', '2023-10', '2023-11', '2023-12'],
            dtype='period[M]')
```

Pandas NumPy:

Pandas and NumPy are two popular libraries in Python for data manipulation and analysis. While NumPy provides a fundamental array object for numerical computing, Pandas builds on top of NumPy to provide more advanced data structures and analysis tools.

Boolean Indexing:

Boolean indexing is a technique used in Pandas and NumPy to select subsets of data based on boolean conditions. It allows filtering data based on specific criteria or conditions. With boolean indexing, you can create boolean masks that act as filters to select rows or columns that satisfy certain conditions.

```
import pandas as pd
```

```
import numpy as np

# Create a Pandas Series
s = pd.Series([1, 2, 3, 4, 5])
# Boolean indexing with Pandas
filtered_pandas = s[s > 3]
print(filtered_pandas)

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Boolean indexing with NumPy
filtered_numpy = arr[arr > 3]
print(filtered_numpy)
```

Concatenating Data:

Concatenation is the process of combining multiple data structures along a particular axis. Both Pandas and NumPy provide functions for concatenating data.

Example:

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

```
# Concatenating DataFrames with Pandas
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})
concatenated_pandas = pd.concat([df1, df2])
print(concatenated_pandas)

# Concatenating arrays with NumPy
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])
concatenated_numpy = np.concatenate([arr1, arr2])
print(concatenated_numpy)
```

Pandas vs NumPy:

- NumPy provides the fundamental array object that allows efficient numerical computing and mathematical operations on arrays. It is well-suited for handling large numerical datasets and performing mathematical computations.
- Pandas, on the other hand, builds on top of NumPy and provides higher-level data structures like Series and DataFrame, which are more convenient for data manipulation and analysis. Pandas is designed for working with structured and tabular data, offering functionality for data cleaning, filtering, merging, reshaping, and analysis.
- While NumPy focuses on numerical computing and array operations, Pandas extends NumPy's capabilities by adding data alignment, handling missing values, powerful indexing and slicing, and integration with other data formats and libraries.

- In summary, NumPy is primarily used for numerical computations and arrays, while Pandas is used for data manipulation, analysis, and working with structured data. They are often used together, with Pandas leveraging the array operations and efficient data structures provided by NumPy.

Convert Pandas DataFrame to Numpy array

To convert a Pandas DataFrame to a NumPy array, you can use the `values` attribute of the DataFrame.

Here's an example:

```
import pandas as pd
import numpy as np
# Create a sample DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Convert DataFrame to NumPy array
array = df.values
print(array)
```

Output 👇

```
[[1 4]
 [2 5]
 [3 6]]
```

In this example, the `values` attribute is used to retrieve the underlying data of the DataFrame as a NumPy array. The resulting `array` variable will contain the values of the DataFrame in a two-dimensional array format.

Convert Pandas DataFrame to CSV

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

# Convert DataFrame to CSV file
df.to_csv('output.csv', index=False)

# Read the CSV file to verify the data
df_read = pd.read_csv('output.csv')
print(df_read)
```

Output 👇

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

In this example, the `to_csv()` function is applied on the DataFrame `df` to convert it to a CSV file named 'output.csv'. The `index=False` parameter is used to exclude the index column from the CSV file.

You can specify other parameters such as `sep` (delimiter), `header` (whether to include the column names), and `columns` (select specific columns) according to your requirements.

Make sure to provide the appropriate file path and filename when using `to_csv()`.

Python Pandas Reading Files

To read various types of files using Pandas, you can use the following functions:

✅ Reading from CSV File:

👉 You can use the read_csv() function to read data from a CSV file.

Example:

```
import pandas as pd
# Read CSV file
df = pd.read_csv('data.csv')
```

✅ Reading from Excel File:

👉 To read data from an Excel file, you can use the read_excel() function.

Example:

```
import pandas as pd
# Read Excel file
df = pd.read_excel('data.xlsx')
```

✅ Reading from JSON:

👉 You can use the read_json() function to read data from a JSON file.

Example:

```
import pandas as pd
# Read JSON file
df = pd.read_json('data.json')
```

✅ Reading from a SQL Database:

👉 Pandas provides the read_sql() function to read data from a SQL database. You need to establish a connection to the database and pass the SQL query or table name.

Example:

```
import pandas as pd
import sqlite3

# Establish a connection to the database
conn = sqlite3.connect('database.db')

# Read data from a SQL query
query = 'SELECT * FROM table_name'
df = pd.read_sql(query, conn)

# Read data from a table
table_name = 'table_name'
df = pd.read_sql_table(table_name, conn)
```

Note: The above example uses SQLite as an example database. You need to install the appropriate database driver and modify the connection parameters according to your specific database.

Ensure that you have the necessary libraries installed (e.g., `pandas`, `xlrd`, `openpyxl`, `pyodbc`, etc.) to read files in the desired format.

Nitish Mehta