Notes for Data Science using Python

By:-Ritika Bishnoi

Numpy

Numpy is a powerful library for numerical computations in Python, utilizing arrays for efficient data storage and manipulation. Its array data structures are essential in data science for handling large datasets and performing complex mathematical operations quickly as it has "ndarray" i.e N dimensional array which is capable of storing data in tabular format which was not possible using array from standard python library which only facilitated use of 1 Dimension Array.

NDimensional Array

Features:

- Store data in Contigous Memory location
- Can perform **Mathematical Operation** on data items stored in array unlike list
- Data is homogeneous i.e the datatype of element with highest hierarchy is considered as datatype of all elements unlike list that supported elements elements of different datatypes.
- Elements are not comma separated unlike list eg. [[2 2] [3 3]]

Array Creation:

To create a NumPy array from a list, you use the **np.array() constructor.**

Here is an example:

```
import numpy as np


# Creating a 1D array from a list

list_1d = [1, 2, 3, 4, 5]

array_1d = np.array(list_1d)

print("1D Array:", array_1d)


# Creating a 2D array from a list of lists

list_2d = [[1, 2, 3], [4, 5, 6]]

array_2d = np.array(list_2d)

print("2D Array:\n", array_2d)
```

Adding Data to an Array

Adding Data at the End

To add new data at the end of an array, you can use the **np.append()** function:

```
# Adding to a 1D array
array_1d = np.append(array_1d, [6, 7])
print("1D Array after append:", array_1d)

# Adding to a 2D array (note: this will flatten the array)
array_2d = np.append(array_2d, [[7, 8, 9]], axis=None)
print("2D Array after append (flattened):", array_2d)
```

Adding Data at a Specific Position

To insert data at a specific position in an array, you can use the **np.insert()** function:

```
# Inserting into a 1D array
array_1d = np.insert(array_1d, 2, [8, 9])
print("1D Array after insert:", array_1d)

# Inserting into a 2D array (note: this will flatten the array if axis is not specified)
array_2d = np.insert(array_2d, 1, [10, 11, 12])
print("2D Array after insert (flattened):", array_2d)

# Inserting into a specific axis of a 2D array
array_2d = np.array(list_2d)
array_2d = np.insert(array_2d, 1, [10, 11, 12], axis=0)
print("2D Array after insert with axis:\n", array_2d)
```

Deleting Data from a Specific Position

To delete data from a specific position in an array, you can use the **np.delete()** function:

```
# Deleting from a 1D array
array_1d = np.delete(array_1d, [2, 3])
print("1D Array after delete:", array_1d)


# Deleting from a 2D array
array_2d = np.delete(array_2d, 1, axis=0)
print("2D Array after delete:\n", array_2d)
```

Effect of Append and Insert on 2D Arrays

Using np.append() on a 2D Array

Appending data to a 2D array without specifying the axis parameter will flatten the array into a 1D array:

```
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_2d_flat = np.append(array_2d, [7, 8, 9])
print("2D Array after append (flattened):", array_2d_flat)
```

Using np.insert() on a 2D Array

Inserting data into a 2D array without specifying the axis parameter will also flatten the array into a 1D array:

```
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_2d_flat = np.insert(array_2d, 1, [7, 8, 9])
print("2D Array after insert (flattened):", array_2d_flat)
```

However, specifying the **axis parameter** maintains the 2D structure:

```
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_2d_inserted = np.insert(array_2d, 1, [7, 8, 9], axis=0)
print("2D Array after insert with axis:\n", array_2d_inserted)
```

Slicing in NumPy

It is a way to extract parts of an array using a range of indices. The slicing syntax is [start:end:jump], where:

- start: The index to start the slice (inclusive).
- end: The index to end the slice (exclusive).
- jump: The step size or interval between indices in the slice.

Slicing a 1D Array

A 1D array can be sliced to extract individual elements or subarrays.

Example:

```
import numpy as np

# Creating a 1D array
array_1d = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Slicing to get elements from index 2 to 5
subarray_1d = array_1d[2:6]
print("Sliced 1D Array (elements 2 to 5):", subarray_1d)

# Slicing to get every other element from the entire array
subarray_1d_step = array_1d[::2]
print("Sliced 1D Array (every other element):", subarray_1d_step)
```

Slicing a 2D Array

A 2D array can be sliced to extract individual elements, 1D arrays, or sub-2D arrays.

Example:

```
# Creating a 2D array
array_2d = np.array([[0, 1, 2, 3, 4],
                     [5, 6, 7, 8, 9],
                     [10, 11, 12, 13, 14],
                     [15, 16, 17, 18, 19],
                     [20, 21, 22, 23, 24]])

# Slicing to get a single element (row 2, column 3)
element_2d = array_2d[2, 3]
print("Sliced 2D Array (element at row 2, column 3):", element_2d)

# Slicing to get a 1D array (row 1)
subarray_1d_from_2d = array_2d[1, :]
print("Sliced 2D Array (row 1 as 1D array):", subarray_1d_from_2d)

# Slicing to get a sub-2D array (rows 1 to 3, columns 2 to 4)
subarray_2d = array_2d[1:4, 2:5]
```

```
print("Sliced 2D Array (subarray from rows 1 to 3 and columns 2 to 4):\n", subarray_2d)

# Slicing to get every other element in each row
subarray_2d_step = array_2d[:, ::2]
print("Sliced 2D Array (every other element in each row):\n", subarray_2d_step)
```

Summary of Slicing:

- **1D Array Slicing:**
  - Extract elements: array_1d[start:end]
  - Extract elements with a step: array_1d[start:end:step]
- **2D Array Slicing:**
  - Extract an element: array_2d[row_index, column_index]
  - Extract a row (1D array): array_2d[row_index, :]
  - Extract a column (1D array): array_2d[:, column_index]
  - Extract a sub-2D array: array_2d[row_start:row_end, col_start:col_end]
  - Extract every other element: array_2d[:, ::step]

Splitting Array

The numpy.array_split function in NumPy splits an array into sub-arrays along a specified axis. When the split size n exceeds the length of the array, the function creates as many sub-arrays as possible with size n, and any remaining sub-arrays are empty arrays ([]). This approach ensures that the structure of the original array is preserved, with empty sub-arrays generated when needed.

Function:

- numpy.array_split(array, num_sections, axis=0)

Explanation:

- Splits an array into multiple sub-arrays along a specified axis.

Example: Consider a NumPy array arr with 8 elements:

```
python
Copy code
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

Usage of numpy.array_split:

- If you use numpy.array_split(arr, 3):

  ```
  np.array_split(arr, 3) -> [array([1, 2, 3]), array([4, 5, 6]), array([7, 8])]
  ```

  - Here, arr is split into 3 sub-arrays.

Handling n > len(array)

- **Example:**
  - If you split arr with numpy.array_split(arr, 10):

    python
    Copy code
    ```
    np.array_split(arr, 10) -> [array([1, 2]), array([3, 4]), array([5]), array([6]), array([7]), array([8]),
    array([], dtype=int64), array([], dtype=int64), array([], dtype=int64), array([], dtype=int64)]
    ```

    - Here, the split size (10) is greater than the length of arr (8). As a result, NumPy creates as many sub-arrays as possible with the specified size (10), and the remaining sub-arrays are empty arrays ([]). This behavior ensures that the original dimension of the array is maintained, even if some resulting arrays are empty.

Concatenating Arrays with numpy.concatenate

- **Function:**
  - **numpy.concatenate((arrays, axis=0, out=None))**
- **Description:**
  - Concatenates arrays along a specified axis.
- **Parameters:**
  - arrays: Sequence of arrays to be concatenated.
  - axis: Axis along which arrays will be joined. Default is 0 (along the first dimension).
  - out: Optional output array.
- **Example:**

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Concatenate along axis 0 (rows)
result = np.concatenate((arr1, arr2), axis=0)
print(result)
```

Output:

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

Horizontal Stack (numpy.hstack) and Vertical Stack (numpy.vstack)

- **numpy.hstack:**
  - Stacks arrays horizontally (column-wise).
- **numpy.vstack:**
  - Stacks arrays vertically (row-wise).
- **Similarity with numpy.concatenate:**
  - Both hstack and vstack are specialized forms of concatenate.
  - They are convenient shortcuts for specific axis concatenations (hstack for axis=1 and vstack for axis=0).
- **Example:**

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Using hstack and vstack
hstack_result = np.hstack((arr1, arr2))
vstack_result = np.vstack((arr1, arr2))

print("Horizontal stack:")
print(hstack_result)
print("\nVertical stack:")
print(vstack_result)
```

Output:

```
Horizontal stack:
[[1 2 5]
 [3 4 6]]

Vertical stack:
[[1 2]
 [3 4]
 [5 6]]
```

Delivering Similar Results

- **Similarity:**
  - concatenate, hstack, and vstack all concatenate arrays but differ in the axis along which they concatenate.
  - They can be used interchangeably to achieve similar results by adjusting the axis parameter in concatenate.
- **Example (Using concatenate for hstack):**

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Using concatenate for hstack (axis=1) and vstack(axis=0)
concatenate_result = np.concatenate((arr1, arr2.T), axis=1)
```

```
print("Concatenate for horizontal stack:")
print(concatenate_result)
```

Output:

```
Concatenate for horizontal stack:
[[1 2 5]
 [3 4 6]]
```

Conclusion

NumPy provides several methods (concatenate, hstack, vstack) to concatenate arrays, allowing flexibility in how arrays can be combined either along rows or columns. Understanding these methods helps in efficiently manipulating array dimensions in data processing tasks.

Sorting Arrays (sort function)

- **Function:**
    - sort()
- **Description:**
    - Sorts elements of an array in-place in ascending order.
- **Example:**

```
arr = [3, 1, 4, 2]
arr.sort()
print(arr)  # Output: [1, 2, 3, 4]
```

- **Details:**
    - The sort() function modifies the original array arr and arranges its elements in ascending order by default. For descending order, you can use arr.sort(reverse=True).

Finding Elements (where function)

- **Function:**
    - There's where function for arrays in Python that use list methods or NumPy functions for specific tasks.
- **Example (Using NumPy where function for finding indices):**

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
indices = np.where(arr == 3)[0]
print(indices)  # Output: [2]
```

- **Details:**
    - NumPy's where function is commonly used to find indices where a condition is true (arr == 3 in this case). It returns an array of indices where the condition holds true.

Searching Sorted Arrays (searchsorted function)

- **Function:**
  - searchsorted()
- **Description:**
  - Finds the indices where elements should be inserted to maintain order in a sorted array.
- **Example:**

```
import numpy as np

arr = np.array([1, 3, 5, 7, 9])
index = np.searchsorted(arr, 6)
print(index)  # Output: 3
```

- **Details:**
  - searchsorted() returns the index where 6 would be inserted to keep arr sorted. In this case, it would be between 5 and 7, hence the index 3.

Conclusion

Understanding these operations (sort, where, searchsorted) is essential for efficiently manipulating and querying arrays in Python. They provide tools for sorting elements, locating specific values or conditions, and managing sorted arrays effectively.

Filtering Array

Filtering Data Based on Boolean Array

- **Concept:**
  - Use a boolean array of the same structure as the original 2D array to filter data.
- **Example:**

```
import numpy as np

arr = np.array([[10, 15, 20], [25, 30, 35]])
bool_arr = np.array([[True, False, True], [False, True, False]])

filtered_arr = arr[bool_arr]
print(filtered_arr)  # Output: [10 20 30]
```

- **Details:**
  - The boolean array bool_arr has the same structure as arr.
  - Only the elements in arr where the corresponding value in bool_arr is True are included in filtered_arr.
  - The result is a 1D array with the filtered elements.

Filtering Data Based on Logical Conditions

- **Concept:**
  - Use logical conditions to create a boolean array for filtering in a 2D array.
- **Example:**

```
import numpy as np

arr = np.array([[10, 15, 20], [25, 30, 35]])
condition = arr > 20

filtered_arr = arr[condition]
print(filtered_arr)  # Output: [25 30 35]
```

- **Details:**
  - The logical condition arr > 20 creates a boolean array of the same structure as arr.
  - Only the elements in arr where the condition is True are included in filtered_arr.
  - The result is a 1D array with the filtered elements.

Getting List of Indices Based on Logical Conditions

- **Concept:**
  - Use np.where() to get indices where the condition is True in a 2D array.
- **Example:**

```
python
Copy code
import numpy as np

arr = np.array([[10, 15, 20], [25, 30, 35]])
condition = arr > 20

indices = np.where(condition)
print(indices)  # Output: (array([1, 1, 1]), array([0, 1, 2]))
```

- **Details:**
  - np.where(condition) returns a tuple of arrays (one for each dimension) indicating the indices where the condition is True.
  - For a 2D array, the result is a tuple of two arrays: one for row indices and one for column indices.
  - These indices can be used to identify or extract the elements that satisfy the condition.

Complete Example with Explanation:

```
import numpy as np

arr = np.array([[10, 15, 20], [25, 30, 35]])

# Filtering using boolean array
bool_arr = np.array([[True, False, True], [False, True, False]])
```

```
filtered_arr = arr[bool_arr]
print("Filtered array using boolean array:", filtered_arr)  # Output: [10 20 30]

# Filtering using logical condition
condition = arr > 20
filtered_arr = arr[condition]
print("Filtered array using condition:", filtered_arr)  # Output: [25 30 35]

# Getting indices using logical condition
indices = np.where(condition)
print("Indices of elements > 20:", indices)  # Output: (array([1, 1, 1]), array([0, 1, 2]))

# Extracting elements using indices
filtered_elements = arr[indices]
print("Filtered elements using indices:", filtered_elements)  # Output: [25 30 35]
```

Explanation:

1. **Filtering Using Boolean Array:**
   - bool_arr is used to filter arr. The filtered array is [10, 20, 30] because these are the elements where bool_arr is True.
2. **Filtering Using Logical Condition:**
   - condition is created using arr > 20. The filtered array is [25, 30, 35] because these elements satisfy the condition.
3. **Getting Indices Using Logical Condition:**
   - np.where(condition) returns indices (array([1, 1, 1]), array([0, 1, 2])), indicating the positions of elements greater than 20 in arr.
   - Using these indices, we can extract the filtered elements, resulting in [25, 30, 35].

This example demonstrates how to filter 2D arrays based on boolean arrays and logical conditions, as well as how to retrieve the indices of elements that meet specific criteria. The filtered results are always 1D arrays.

Mathematical and Statistical Operation on Arrays

Using both operators and methods in NumPy allows for efficient implementation of mathematical operations. Statistical functions provide a powerful way to perform data analysis directly on arrays. The tabular summary helps to quickly reference how to use these operations and methods.

Operation	Operator	Method	Example Code (Method)
Addition	+	np.add()	np.add(a, b)
Subtraction	-	np.subtract()	np.subtract(a, b)
Multiplication	*	np.multiply()	np.multiply(a, b)
Division	/	np.divide()	np.divide(a, b)
Exponentiation	**	np.power()	np.power(a, 2)
Mean	N/A	np.mean()	np.mean(arr)
Median	N/A	np.median()	np.median(arr)
Standard Deviation	N/A	np.std()	np.std(arr)
Variance	N/A	np.var()	np.var(arr)
Sum	N/A	np.sum()	np.sum(arr)
Minimum	N/A	np.min()	np.min(arr)
Maximum	N/A	np.max()	np.max(arr)

Pandas

Pandas uses DataFrames, which are 2D labeled data structures, and Series, which are 1D labeled arrays, to efficiently organize, manipulate, and analyze data. These structures facilitate operations like filtering, aggregating, and transforming data, making complex data analysis tasks straightforward.

Data Frame Creation

Pandas DataFrames can be created from dictionaries, Series, nested lists, and ndarrays, providing flexibility for data manipulation. Importing data from CSV or Excel files into DataFrames is straightforward, with additional packages required for reading Excel files.

From Dictionary

```
import pandas as pd

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

From Series

```
import pandas as pd

s1 = pd.Series([1, 2, 3], name='A')
s2 = pd.Series([4, 5, 6], name='B')
df = pd.DataFrame({s1.name: s1, s2.name: s2})
print(df)
```

From Nested List

```
import pandas as pd

data = [[1, 2, 3], [4, 5, 6]]
df = pd.DataFrame(data, columns=['A', 'B', 'C'])
print(df)
```

From ndarray

```
import pandas as pd
import numpy as np

data = np.array([[1, 2, 3], [4, 5, 6]])
df = pd.DataFrame(data, columns=['A', 'B', 'C'])
print(df)
```

From CSV

```
import pandas as pd

df = pd.read_csv('path/file.csv')
print(df)
```

From Excel

```
import pandas as pd

# Make sure to install necessary packages:
# pip install openpyxl
# pip install xlrd
```

```
df = pd.read_excel('path/file.xlsx', engine='openpyxl')  # For .xlsx files
df = pd.read_excel('path/file.xls', engine='xlrd')       # For .xls files
print(df)
```

From SQL

1. **Setup:**
   - Install SQLAlchemy & PyMySQL:

     ```
     pip install sqlalchemy
     pip install PyMySQL
     ```

2. **Import Libraries:**

   ```
   import pandas as pd
   from sqlalchemy import create_engine
   ```

3. **Create Engine:**

   ```
   engine = create_engine("mysql+pymysql://username:password@localhost:3306/databaseName")
   ```

4. **Using read_sql_table:**
   - Load an entire table:

     ```
     df_table = pd.read_sql_table('table_name', engine)
     print(df_table)
     ```

5. **Using read_sql_query:**
   - Load data based on a query:

     ```
     query = 'SELECT * FROM table_name WHERE column_name > 10'
     df_query = pd.read_sql_query(query, engine)
     print(df_query)
     ```

Basic Data frame functions:

1. **describe() Function:**
   - Provides summary statistics.
   - Example:

     ```
     df.describe()
     ```

   - Output:

     ```
            A    B
     count  3.0  3.0
     mean   2.0  5.0
     std    1.0  1.0
     min    1.0  4.0
     25%    1.5  4.5
     ```

```
50%    2.0  5.0
75%    2.5  5.5
max    3.0  6.0
```

2. **info() Function:**
   - Provides a concise summary of the DataFrame.
   - Example:

     ```
     df.info()
     ```

   - Output:

     ```
     <class 'pandas.core.frame.DataFrame'>
     RangeIndex: 3 entries, 0 to 2
     Data columns (total 2 columns):
      #   Column  Non-Null Count  Dtype
     ---  ------  --------------  -----
      0   A       3 non-null      int64
      1   B       3 non-null      int64
     dtypes: int64(2)
     memory usage: 176.0 bytes
     ```

3. **Changing Data Type of Elements:**
   - Use astype() method.
   - Example:

     ```
     df['A'].astype(float)
     ```

   - Output:

     ```
          A  B
     0  1.0  4
     1  2.0  5
     2  3.0  6
     ```

4. **Defining Header Part:**
   - Specify columns parameter.
   - Example:

     ```
     pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['X', 'Y', 'Z'])
     ```

   - Output:

     ```
        X  Y  Z
     0  1  2  3
     1  4  5  6
     ```

5. **Finding Unique Values of Columns:**
   - Use unique() method.
   - Example:

     ```
     df['X'].unique()
     ```

   - Output:

     ```
     [1 4]
     ```

Handling Duplicate Data in Pandas

duplicated() Method

- **Purpose:**
  - Identifies duplicate rows or column values.
- **Example:**

```
import pandas as pd

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

# Identify duplicate rows
print(df.duplicated())
# Output:
# 0    False
# 1    False
# 2     True
# 3    False
# dtype: bool

# Identify duplicates in column 'A'
print(df['A'].duplicated())
# Output:
# 0    False
# 1    False
# 2     True
# 3    False
# Name: A, dtype: bool
```

drop_duplicates() Method

- **Purpose:**
  - Removes duplicate rows or column values.
- **Example:**

```
# Remove duplicate rows
df_no_dup = df.drop_duplicates()
print(df_no_dup)
# Output:
#    A  B
# 0  1  5
```

```
# 1  2  6
# 3  4  8

# Remove duplicates from column 'A'
df_no_dup_col = df.drop_duplicates(subset=['A'])
print(df_no_dup_col)
# Output:
#    A  B
# 0  1  5
# 1  2  6
# 3  4  8
```

Summary

1. **Identify Duplicate Rows or Columns:**
   - Using duplicated() method.

```
df.duplicated()         # For rows
df['A'].duplicated()    # For column 'A'
```

2. **Remove Duplicate Rows or Columns:**
   - Using drop_duplicates() method.

```
df.drop_duplicates()          # Remove duplicate rows
df.drop_duplicates(subset=['A'])  # Remove duplicates in column 'A'
```

Filling Missing Data

1. Using “fillna”

Forward Fill (ffill):

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Forward fill
df.fillna(method='ffill', inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  NaN
1  1.0  2.0
2  3.0  2.0
3  3.0  4.0
4  5.0  4.0
```

Backward Fill (bfill):

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Backward fill
df.fillna(method='bfill', inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  2.0
1  3.0  2.0
2  3.0  4.0
3  5.0  4.0
4  5.0  NaN
```

Filling with a Specific Value:

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Fill with 0
df.fillna(value=0, inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  0.0
1  0.0  2.0
2  3.0  0.0
3  0.0  4.0
4  5.0  0.0
```

2. Using "replace"

```
import numpy as np
import pandas as pd

# Sample DataFrame
data = {'A': [1, np.nan, 3, np.nan, 5], 'B': [np.nan, 2, np.nan, 4, np.nan]}
df = pd.DataFrame(data)

# Replace NaN with a specific value
df.replace(np.nan, -1, inplace=True)
```

```
print(df)
```

Output:

```
     A    B
0  1.0 -1.0
1 -1.0  2.0
2  3.0 -1.0
3 -1.0  4.0
4  5.0 -1.0
```

3. Interpolating Missing Data

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Interpolate
df.interpolate(inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  4.0  4.0
4  5.0  4.0
```

4. Using na_values in read_csv

```
import pandas as pd

# Sample data with specific values considered as NaN
data = """A,B
1,?
?,2
3,?
?,4
5,?
"""

# Read CSV with na_values
df = pd.read_csv(pd.compat.StringIO(data), na_values=['?'])
print(df)
```

Output:

```
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  NaN
3  NaN  4.0
4  5.0  NaN
```

5. Dropping Missing Data with dropna

Drop Rows with Any NaN:

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Drop rows with any NaN
df.dropna(axis=0, how='any', inplace=True)
print(df)
```

Output:

```
     A    B
```

(No rows are printed because all rows have at least one NaN.)

Drop Rows with All NaN:

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Drop rows with all NaN
df.dropna(axis=0, how='all', inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  NaN
3  NaN  4.0
4  5.0  NaN
```

Drop Rows with Specific Column NaN:

```
import pandas as pd

# Sample DataFrame
data = {'A': [1, None, 3, None, 5], 'B': [None, 2, None, 4, None]}
df = pd.DataFrame(data)

# Drop rows where 'A' is NaN
df.dropna(subset=['A'], inplace=True)
print(df)
```

Output:

```
     A    B
0  1.0  NaN
2  3.0  NaN
4  5.0  NaN
```

Renaming Column Names in Pandas

1. Using rename() Method

The rename() method allows you to change the column names of a DataFrame. You can rename columns by passing a dictionary with old column names as keys and new column names as values.

Example:

```
import pandas as pd

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

# Rename columns
df.rename(columns={'A': 'Alpha', 'B': 'Beta'}, inplace=True)
print(df)
```

Output:

```
   Alpha  Beta
0      1     4
1      2     5
2      3     6
```

2. Using columns Attribute

You can directly assign a list of new column names to the columns attribute of the DataFrame.

Example:

```
import pandas as pd

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

# Rename columns
df.columns = ['Alpha', 'Beta']
print(df)
```

Output:

```
   Alpha  Beta
0      1     4
1      2     5
2      3     6
```

Creating new (Calculated) columns

These methods provide various ways to create calculated columns in a pandas DataFrame, allowing you to perform both simple arithmetic and complex logic-based calculations efficiently.

1. Direct Formula-Based Method

You can create new columns in a DataFrame by performing arithmetic operations directly on existing columns.

Example:

```
import pandas as pd

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

# Create a new column 'C' as the sum of 'A' and 'B'
df['C'] = df['A'] + df['B']
print(df)
```

Output:

```
   A  B  C
0  1  4  5
1  2  5  7
2  3  6  9
```

2. Logic-Based Method using loc

You can use the loc method to create columns based on conditional logic.

Example:

```
import pandas as pd

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

# Create a new column 'C' with values based on a condition
df['C'] = 0  # Initialize with default value
df.loc[df['A'] > df['B'], 'C'] = 1
df.loc[df['A'] <= df['B'], 'C'] = -1
print(df)
```

Output:

```
   A  B  C
0  1  4 -1
1  2  3 -1
2  3  2  1
3  4  1  1
```

3. Using apply for Complex Formulas or Logic

The apply method allows you to use a function to create complex calculations or logic for new columns.

Example:

```
import pandas as pd

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

# Define a function for complex logic
```

```
def complex_logic(row):
    if row['A'] > row['B']:
        return row['A'] * 2
    else:
        return row['B'] * 3

# Apply the function to create a new column 'C'
df['C'] = df.apply(complex_logic, axis=1)
print(df)
```

Output:

```
   A  B  C
0  1  4 12
1  2  3  9
2  3  2  6
3  4  1  8
```

4. Using map for Mapping Values

The map method is useful for mapping values in a column based on a dictionary or function.

Example:

```
python
Copy code
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4]
})

# Define a mapping function
def map_logic(x):
    return x * 10

# Map the function to create a new column 'B'
df['B'] = df['A'].map(map_logic)
print(df)
```

Output:

```
   A   B
0  1  10
1  2  20
2  3  30
3  4  40
```

Deleting Existing Column

1. Using `del`

The `del` statement can be used to delete a column from a DataFrame.

Example:

```
import pandas as pd

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

# Delete column 'B'
del df['B']
print(df)
```

Output:

```
   A  C
0  1  7
1  2  8
2  3  9
```

2. Using `pop`

The `pop` method removes a column and returns it as a Series.

Example:

```
import pandas as pd

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

# Pop column 'B'
popped_column = df.pop('B')
print(df)
print(popped_column)
```

Output:

```
   A  C
0  1  7
1  2  8
2  3  9

0    4
1    5
2    6
Name: B, dtype: int64
```

3. Using drop

The drop method can delete one or more columns. It returns a new DataFrame by default and can modify the original DataFrame if inplace=True.

Example:

```
import pandas as pd

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

# Drop column 'B' and 'C'
df_dropped = df.drop(columns=['B', 'C'])
print(df_dropped)

# Drop column 'B' inplace
df.drop(columns='B', inplace=True)
print(df)
```

Output (new DataFrame):

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

Output (inplace):

```
   A  C
0  1  7
1  2  8
2  3  9
```

Differences Between del, pop, and drop

- **del**: Deletes the column permanently from the DataFrame. It does not return the column.
- **pop**: Deletes the column and returns it as a Series. Useful if you need to keep the deleted column data for further use.
- **drop**: Can delete one or multiple columns. By default, it returns a new DataFrame without the specified columns unless inplace=True is set.

Benefits of Setting an Index

1. **Improved Data Manipulation**: Setting an index allows for more efficient data manipulation. Indexes can be used to quickly look up, join, and merge data.
2. **Enhanced Data Selection**: Accessing rows by index values is faster and more intuitive. This is especially useful for time series data where the index might be dates.
3. **Better Grouping and Aggregation**: Indexes are essential for grouping and aggregation operations. They allow for more complex and efficient data analysis.
4. **Cleaner Data Representation**: Setting a meaningful index (e.g., a unique identifier) makes the DataFrame easier to understand and work with, as the index often represents a key aspect of the data.
5. **Avoiding Duplicate Rows**: When you set an index, you can enforce uniqueness, which helps prevent duplicate rows and maintains data integrity.

1. Using set_index Method

The set_index method allows you to set one or more columns as the index of the DataFrame.

Example:

```
import pandas as pd

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

# Set column 'A' as the index
df.set_index('A', inplace=True)
print(df)
```

Output:

```
   B  C
A
1  4  7
2  5  8
3  6  9
```

2. Using index_col Parameter in read_csv

You can set the index of a DataFrame when reading from a CSV file by using the index_col parameter.

Example:

```
import pandas as pd

# Sample CSV data
data = """A,B,C
1,4,7
2,5,8
3,6,9"""

# Read CSV with column 'A' as the index
df = pd.read_csv(pd.compat.StringIO(data), index_col='A')
print(df)
```

Output:

```
   B  C
A
1  4  7
2  5  8
3  6  9
```

3. Using set_axis Method

You can also set the index using the set_axis method with the axis=0 parameter.

Example:

```
import pandas as pd

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

# Set a new index using 'A' column values
df.set_axis(df['A'], axis=0, inplace=True)
print(df)
```

Output:

```
   A  B  C
1  1  4  7
2  2  5  8
```

```
3  3  6  9
```

4. Using reset_index Method

You can reset the index to the default integer index and optionally convert the index back to a column.

Example:

```
import pandas as pd

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

# Set column 'A' as the index
df.set_index('A', inplace=True)
print(df)

# Reset index
df.reset_index(inplace=True)
print(df)
```

Output:

```
   B  C
A
1  4  7
2  5  8
3  6  9

   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```

Concatenating Data Frames

1. Concatenating Along Rows (axis=0)

Concatenating DataFrames along rows means stacking them vertically. This is done by setting axis=0 in the concat function.

Example:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'A': [1, 2],
    'B': [3, 4]
})

df2 = pd.DataFrame({
    'A': [5, 6],
    'B': [7, 8]
})

# Concatenate along rows
result = pd.concat([df1, df2], axis=0)
print(result)
```

Output:

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

2. Concatenating Along Columns (axis=1)

Concatenating DataFrames along columns means stacking them side by side. This is done by setting `axis=1` in the `concat` function.

Example:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'A': [1, 2]
})

df2 = pd.DataFrame({
    'B': [3, 4]
})

# Concatenate along columns
result = pd.concat([df1, df2], axis=1)
print(result)
```

Output:

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

Summary

- **Concatenating along rows (axis=0)** stacks DataFrames vertically.
- **Concatenating along columns (axis=1)** stacks DataFrames side by side.

These methods allow you to combine DataFrames in a flexible manner, depending on the structure and requirements of your data.

Comparing Data Frames

1. Using compare Method

The compare method in pandas is used to compare two DataFrames and highlight their differences.

Basic Syntax:

```
df1.compare(df2, align_axis=0, keep_equal=False, keep_shape=False)
```

2. align_axis Argument

Determines how the results are aligned.

- **align_axis=0** (default): Differences are stacked vertically.
- **align_axis=1**: Differences are aligned horizontally.

Example:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'A': [1, 2],
    'B': [3, 4]
})

df2 = pd.DataFrame({
    'A': [1, 2],
    'B': [5, 4]
})

# Compare with align_axis=0
result = df1.compare(df2, align_axis=0)
```

```
print("align_axis=0:\n", result)

# Compare with align_axis=1
result = df1.compare(df2, align_axis=1)
print("\nalign_axis=1:\n", result)
```

Output:

```
align_axis=0:
    B
  self other
0    3     5

align_axis=1:
   A       B
  self other self other
0    1     1    3     5
1    2     2    4     4
```

3. keep_equal Argument

Specifies whether to keep or discard equal values in the result.

- **keep_equal=False** (default): Discards equal values.
- **keep_equal=True**: Keeps equal values in the comparison result.

Example:

```
# Compare with keep_equal=True
result = df1.compare(df2, keep_equal=True)
print("keep_equal=True:\n", result)
```

Output:

```
keep_equal=True:
    A        B
  self other self other
0    1     1    3     5
1    2     2    4     4
```

4. keep_shape Argument

Specifies whether to keep the original shape of the DataFrames.

- **keep_shape=False** (default): Only shows rows and columns with differences.
- **keep_shape=True**: Keeps the original DataFrame shape, filling unmatched elements with NaN.

Example:

```
# Compare with keep_shape=True
result = df1.compare(df2, keep_shape=True)
print("keep_shape=True:\n", result)
```

Output:

```
keep_shape=True:
    A         B
  self other self other
0  NaN   NaN  3.0   5.0
1  NaN   NaN  NaN   NaN
```

Summary

- **align_axis**: Controls alignment of differences (0 for vertical, 1 for horizontal).
- **keep_equal**: Decides whether to keep equal values in the result.
- **keep_shape**: Determines if the original shape should be retained, filling unmatched elements with NaN.

Without these arguments, the default behavior (align_axis=0, keep_equal=False, keep_shape=False) is used, which highlights only the differing values, stacked vertically.

Merging DataFrames in Pandas

1. Using merge Method

The merge method in pandas combines two DataFrames based on a key column or index. It is similar to SQL joins.

Basic Syntax:

```
pd.merge(left, right, how='inner', on='key_column')
```

how Argument Types

The how argument specifies the type of merge to perform.

1. Inner Join (**how='inner'**)
   - Only includes rows with matching keys in both DataFrames.

Example:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [3, 4], 'Age': [23, 25]})

# Inner Join
result = pd.merge(df1, df2, how='inner', on='ID')
print(result)
```

Output:

```
   ID     Name  Age
0   3  Charlie   23
```

2. Left Join (**how='left'**)
    - Includes all rows from the left DataFrame and matched rows from the right DataFrame. Unmatched rows get NaN.

Example:

```
# Left Join
result = pd.merge(df1, df2, how='left', on='ID')
print(result)
```

Output:

```
   ID     Name   Age
0   1    Alice   NaN
1   2      Bob   NaN
2   3  Charlie  23.0
```

3. **Right Join (how='right')**
    - Includes all rows from the right DataFrame and matched rows from the left DataFrame. Unmatched rows get NaN.

Example:

```
# Right Join
result = pd.merge(df1, df2, how='right', on='ID')
print(result)
```

Output:

```
   ID     Name  Age
0   3  Charlie   23
1   4      NaN   25
```

4. **Outer Join** (how='outer')
    - Includes all rows from both DataFrames. Unmatched rows get NaN.

Example:

```
# Outer Join
result = pd.merge(df1, df2, how='outer', on='ID')
print(result)
```

Output:

```
   ID     Name   Age
0   1    Alice   NaN
1   2      Bob   NaN
2   3  Charlie  23.0
3   4      NaN  25.0
```

Summary of SQL Join Equivalents

- **Inner Join**: Matches INNER JOIN in SQL.
- **Left Join**: Matches LEFT JOIN in SQL.
- **Right Join**: Matches RIGHT JOIN in SQL.
- **Outer Join**: Matches FULL OUTER JOIN in SQL.

These join types provide flexibility in combining DataFrames based on common keys, allowing for different inclusion rules for unmatched rows.

Group By Function

To see summarized form of data to get usefull insights from it which would have been difficult to retrieve if data is large and manual efforts would be too tiring.

1. Simple groupby by One Column

The groupby method allows you to group data based on a specific column and then perform aggregate functions on the grouped data.

Example:

```
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'B'],
    'Value1': [10, 20, 30, 40],
    'Value2': [50, 60, 70, 80]
})

# Group by 'Category' and calculate the mean for each group
grouped = df.groupby('Category').mean()
print(grouped)
```

Output:

```
          Value1  Value2
Category
A           20.0    60.0
B           30.0    70.0
```

2. Aggregate Function on Specific Column

You can apply aggregate functions to specific columns within the grouped data.

Example:

```
python
Copy code
# Group by 'Category' and calculate the sum of 'Value1'
grouped = df.groupby('Category')['Value1'].sum()
print(grouped)
```

Output:

```
css
Copy code
Category
A    40
B    60
Name: Value1, dtype: int64
```

3. Using Index and Values on Grouped Object

You can access the index and values of a grouped object for further analysis.

Example:

```
# Group by 'Category'
grouped = df.groupby('Category')

# Access index
print(grouped.indices)

# Iterate through groups
for name, group in grouped:
    print(f"Group name: {name}")
    print(group)
```

Output:

```
{'A': [0, 2], 'B': [1, 3]}
Group name: A
  Category  Value1  Value2
0        A      10      50
2        A      30      70
Group name: B
  Category  Value1  Value2
1        B      20      60
3        B      40      80
```

4. Group by List of Columns (Subgrouping)

You can group by multiple columns to create subgroups and then apply different aggregate functions to each column using a dictionary.

Example:

```
# Sample DataFrame
df = pd.DataFrame({
    'A': ['foo', 'foo', 'bar', 'bar'],
    'B': ['one', 'two', 'one', 'two'],
    'C': [1, 2, 3, 4],
    'D': [5, 6, 7, 8]
})

# Group by 'A' and 'B' and apply different aggregate functions
grouped = df.groupby(['A', 'B']).agg({'C': 'mean', 'D': 'max'})
print(grouped)
```

Output:

```
          C  D
A   B
bar one  3.0  7
    two  4.0  8
foo one  1.0  5
    two  2.0  6
```

Summary

- **Simple groupby by One Column**: Groups data by one column and applies aggregate functions to all or specific columns.
- **Aggregate Function on Specific Column**: Focuses on applying aggregate functions to specific columns.
- **Using Index and Values**: Accesses index and iterates through grouped data for further analysis.
- **Group by List of Columns (Subgrouping)**: Groups by multiple columns and applies different aggregate functions to each column.

These techniques allow you to efficiently summarize and analyze data based on grouping criteria.

Cross Tab

Usage: crosstab is used to compute cross-tabulations of two (or more) factors, showing the **frequency of their intersections**.

Example with Output:

```
import pandas as pd

# Sample data
data = {
    'A': ['foo', 'foo', 'foo', 'bar', 'bar', 'baz'],
    'B': ['one', 'one', 'two', 'two', 'one', 'two'],
    'C': [1, 2, 1, 2, 1, 2]
}
df = pd.DataFrame(data)

# Crosstab example
result = pd.crosstab(df['A'], df['B'])
print(result)
```

Output:

```
B    one  two
A
bar    1    1
baz    0    1
foo    2    1
```

Arguments:

```
import pandas as pd

# Sample data
data = {
    'A': ['foo', 'foo', 'foo', 'bar', 'bar', 'baz'],
    'B': ['one', 'one', 'two', 'two', 'one', 'two'],
    'C': [1, 2, 1, 2, 1, 2]
}
df = pd.DataFrame(data)

# Crosstab example with arguments
result = pd.crosstab(index=df['A'], columns=df['B'], values=df['C'], aggfunc=sum, margins=True)

print(result)
```

Output:

```
B      one  two  All
A
bar    1.0  2.0  3.0
baz    NaN  2.0  2.0
foo    3.0  1.0  4.0
All    4.0  5.0  9.0
```

In this example:

- index=df['A'] specifies 'A' as the row grouping.
- columns=df['B'] specifies 'B' as the column grouping.
- values=df['C'] uses 'C' values to aggregate with sum.
- aggfunc=sum aggregates values using the sum function.
- margins=True adds row and column margins ('All').

Pandas Pivot Table:

Usage: Pivot tables in pandas allow you to **summarize and aggregate data** inside a DataFrame.

Example with Output:

```
import pandas as pd

# Sample data
data = {
    'A': ['foo', 'foo', 'foo', 'bar', 'bar', 'baz'],
    'B': ['one', 'one', 'two', 'two', 'one', 'two'],
    'C': [1, 2, 1, 2, 1, 2],
    'D': [10, 20, 30, 40, 50, 60]
}
df = pd.DataFrame(data)

# Pivot table example with all arguments
result = df.pivot_table(values='D', index='A', columns='B', aggfunc='sum', margins=True, fill_value=0)

print(result)
```

Output:

```
B      one  two  All
A
bar     50   40   90
baz      0   60   60
foo     30   31   61
All     80  131  211
```

Arguments:

- values: Column to aggregate.
- index: Key(s) to group by on the rows.
- columns: Key(s) to group by on the columns.
- aggfunc: Function to use for aggregation ('sum' in this case).
- margins: Add row/column margins (totals).
- fill_value: Replace missing values with this value (default 0).

Pandas Pivot:

Usage: pivot in pandas **reshapes data**, converting **unique values** from **one column into multiple columns**.

Example with Output:

```
import pandas as pd

# Sample data
data = {
    'A': ['foo', 'foo', 'foo', 'bar', 'bar', 'baz'],
    'B': ['one', 'one', 'two', 'two', 'one', 'two'],
    'C': [1, 2, 1, 2, 1, 2],
    'D': [10, 20, 30, 40, 50, 60]
}
df = pd.DataFrame(data)

# Pivot example with all arguments
result = df.pivot(index='A', columns='B', values='D')

print(result)
```

Output:

```
B      one   two
A
bar   50.0  40.0
baz    NaN  60.0
foo   10.0  20.0
```

Difference from pivot_table:

- **pivot** directly **reshapes** data based on **unique values** in specified columns without aggregation.
- **pivot_table aggregates** data when there are **duplicate entries** for the same index/column pair.

Syntax and Arguments:

- index: Column to use to make new frame's index.
- columns: Column to use to make new frame's columns.
- values: Column(s) to use for populating new frame's values.
- No aggfunc or fill_value since pivot doesn't aggregate or fill missing values.

Pandas melt Function:

Usage: melt in pandas **unpivots** a DataFrame from wide format to long format, making it more suitable for analysis and plotting.

Example with Output:

```
import pandas as pd

# Sample data
data = {
    'A': ['foo', 'foo', 'foo', 'bar', 'bar', 'baz'],
    'B': ['one', 'two', 'three', 'one', 'two', 'three'],
    'C': [1, 2, 3, 4, 5, 6],
    'D': [10, 20, 30, 40, 50, 60]
}
df = pd.DataFrame(data)

# Melt example with all arguments
result = pd.melt(df, id_vars=['A', 'B'], value_vars=['C', 'D'], var_name='variable', value_name='value')

print(result)
```

Output:

```
      A      B variable  value
0   foo    one        C      1
1   foo    two        C      2
2   foo  three        C      3
3   bar    one        C      4
4   bar    two        C      5
5   baz  three        C      6
6   foo    one        D     10
7   foo    two        D     20
8   foo  three        D     30
9   bar    one        D     40
10  bar    two        D     50
11  baz  three        D     60
```

Relation to pivot:

- **melt** is essentially the **inverse operation of pivot**.
- pivot transforms long format data into wide format, while melt transforms wide format data into long format.

Arguments:

- id_vars: Columns to keep as identifier variables.
- value_vars: Columns to unpivot. If not specified, uses all columns not set in id_vars.
- var_name: Name to use for the variable column.
- value_name: Name to use for the value column.

Pandas stack Function:

- **Usage:** stack in pandas pivots a level of the column labels (headers) to the row index, similar to melt, especially useful when dealing with multi-level headers.

Example with Output:

```
import pandas as pd

# Sample data with multi-level columns
data = {
    ('A', 0, 'one'): [1, 2, 3],
    ('A', 0, 'two'): [4, 5, 6],
    ('A', 1, 'one'): [7, 8, 9],
    ('A', 1, 'two'): [10, 11, 12],
    ('B', 0, 'one'): [13, 14, 15],
    ('B', 0, 'two'): [16, 17, 18],
    ('B', 1, 'one'): [19, 20, 21],
    ('B', 1, 'two'): [22, 23, 24]
}
df = pd.DataFrame(data)

# Stack example with level=1
stacked = df.stack(level=1)
print(stacked)
```

Output:

```
           one  two
 A 0  one   1    4
      two   2    5
   1  one   7   10
      two   8   11
 B 0  one  13   16
      two  14   17
   1  one  19   22
      two  20   23
```

In this example:

- level=1 specifies to stack the second level of column headers (1), moving them to the innermost level of the row index.
- **By default, stack would stack the innermost level** (2), but with level=1, we specifically choose the second level (1).

Saving Data Frames into Computer as excel sheet / csv file / in Database

Sending DataFrame to XLSX Format:

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)

# Save to XLSX
df.to_excel('data.xlsx', index=False)
```

Sending DataFrame to CSV Format:

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)

# Save to CSV
df.to_csv('data.csv', index=False)
```

Sending DataFrame to SQL using SQLAlchemy:

```
import pandas as pd
from sqlalchemy import create_engine

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)

# Create SQLAlchemy engine
engine = create_engine("mysql+pymysql://username:password@localhost:3306/databaseName")

# Send DataFrame to SQL table
df.to_sql('users', con=engine, if_exists='replace', index=False)
```

```
# Example: Reading from SQL
df_from_sql = pd.read_sql('users', con=engine)
print(df_from_sql)
```

Explanation for SQLAlchemy to SQL Function (to_sql):

- create_engine: Establishes a connection to the SQL database (mysql+pymysql://username:password@localhost:3306/databaseName in this case).
- df.to_sql('users', con=engine, if_exists='replace', index=False): Saves the DataFrame df to a SQL table named users. If the table already exists, it replaces it (if_exists='replace'). The index=False argument ensures that the DataFrame index is not saved as a separate column in the SQL table.

Matplotlib

Matplotlib is a Python library used for creating static, animated, and interactive visualizations, making it essential for data visualization and exploration tasks.

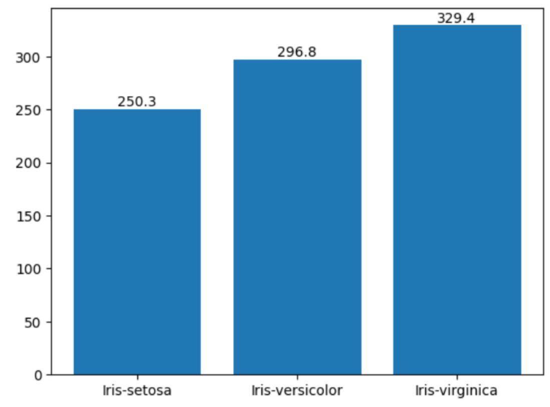
There are in Total 10 charts that I have studied while exploring this library .

***Note: Images used are different from the output of example , images are given just to show how each chart looks**

1. **Bar Chart**

- **Use** Displays categorical data where each category is represented by a bar. In a grouped bar chart, multiple bars are grouped together to compare different categories.

- **Function**: matplotlib.pyplot.bar()
- **Arguments**:
  - x: The x coordinates of the bars.
  - height: The height of the bars.
  - width: The width of the bars (default is 0.8).
  - color: The color of the bars.
  - label: Label for the bars.
  - tick_label: Labels for x-axis ticks.
  - align: Alignment of the bars (e.g., 'center', 'edge').



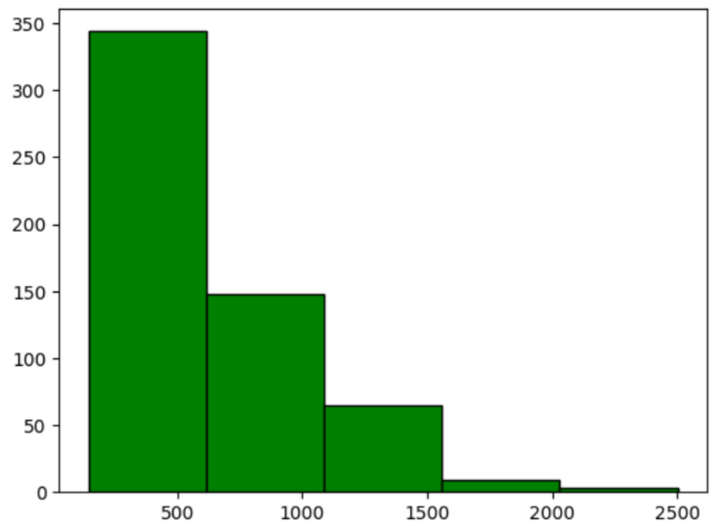
Example:

```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]

plt.bar(categories, values, color='blue', width=0.5, align='center')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart Example')
plt.show()
```

2. Histogram

- **Use** In a histogram, bars represent the frequency or distribution of continuous data within predefined bins.

- **Function**: matplotlib.pyplot.hist()
- **Arguments**:
    - x: Data values (usually a single column of numeric data).
    - bins: Number of bins for the histogram.
    - color: Color of the bars.
    - edgecolor: Color of the edges of bars.
    - alpha: Transparency of bars (0 for transparent, 1 for opaque).
    - label: Label for the histogram.
    - density: If True, the first element of the return tuple will be the counts normalized to form a probability density.

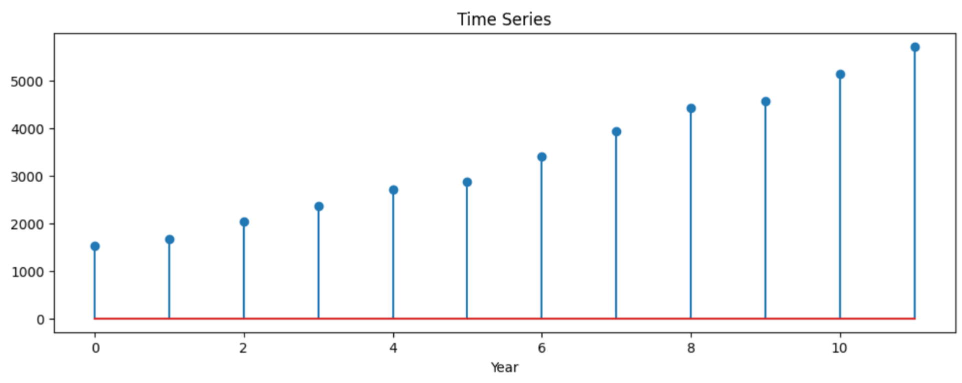


Example:

```
import matplotlib.pyplot as plt
import numpy as np

data = np.random.normal(0, 1, 100)  # Example data
plt.hist(data, bins=20, color='green', edgecolor='black', alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram Example')
plt.show()
```

3. Stem Plot:

- **Use**: Shows data values as markers along a vertical axis, typically used to visualize trends or distributions over time or categories. Each stem represents a category or data point, and leaves represent individual data values.
- **Function**: matplotlib.pyplot.stem()
- **Arguments**:
  - x: Data points for the x-axis.
  - y: Data points for the y-axis.
  - linefmt: Line format (e.g., color, line style).
  - markerfmt: Marker format.
  - basefmt: Base format for the stem lines.
  - use_line_collection: If True, use a LineCollection for improved performance.

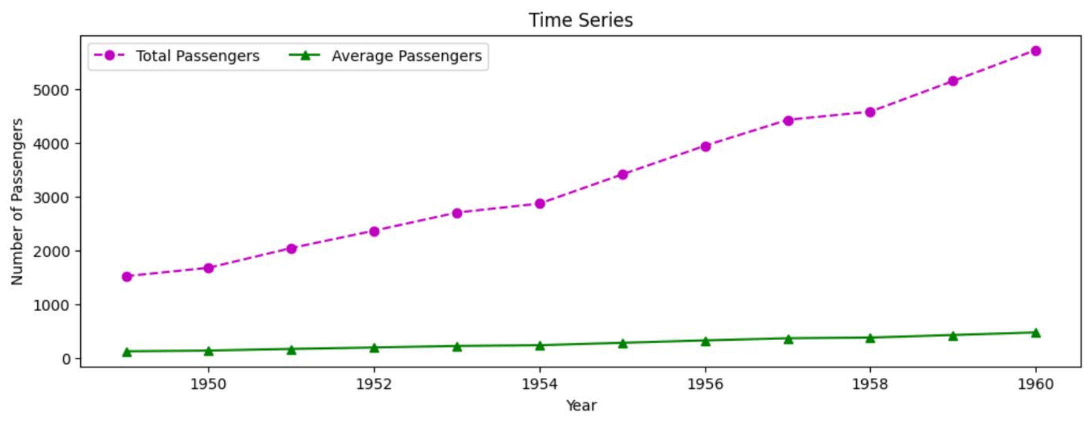


Example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2 * np.pi, 10)
y = np.cos(x)

plt.stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Stem Plot Example')
plt.show()
```

4. **Line Chart**:

- **Use**: Connects data points with straight line segments, ideal for showing trends in data over time or continuous variables. It helps identify patterns and trends, making it suitable for time series analysis or plotting continuous data points.
- **Function**: matplotlib.pyplot.plot()
- **Arguments**:
  - x: Data points for the x-axis.
  - y: Data points for the y-axis.
  - color: Color of the line.
  - linestyle: Line style (e.g., solid, dashed).
  - marker: Marker style (e.g., circle, square).
  - label: Label for the line.
  - alpha: Tranparency of the line (0 for transparent, 1 for opaque).

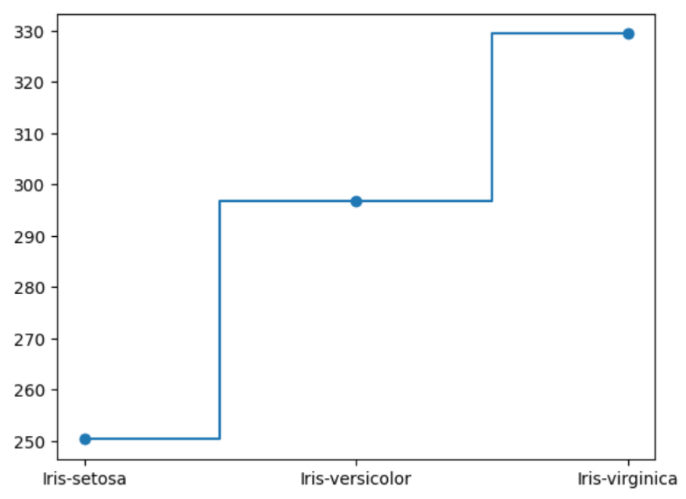


Example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y, color='red', linestyle='--', marker='o', label='Sine Wave')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Line Chart Example')
plt.legend()
plt.show()
```

5. Step Plot:

- **Use**: Similar to a line chart but with vertical steps connecting data points, indicating changes that occur at discrete intervals. Useful for showing data changes that occur abruptly, such as in signal processing or stepwise data trends.
- **Function**: matplotlib.pyplot.step()
- **Arguments**:
  - x: Data points for the x-axis.
  - y: Data points for the y-axis.
  - where: Where to place steps (e.g., 'pre', 'post').
  - color: Color of the steps.
  - linestyle: Line style for the steps.
  - label: Label for the step plot.

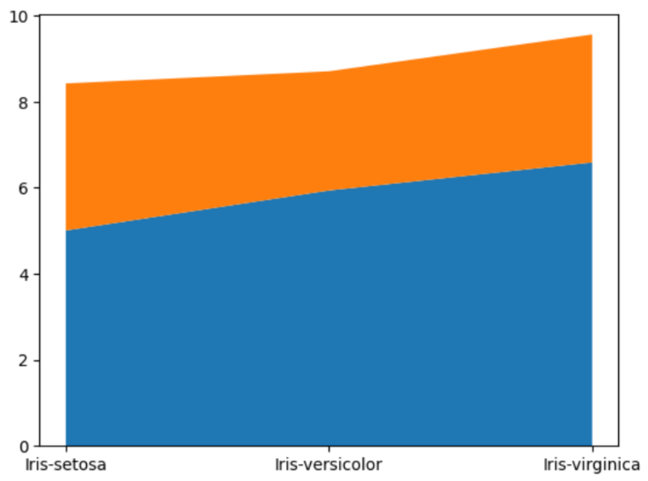


Example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 1)
y = np.random.randint(1, 10, size=10)

plt.step(x, y, where='mid', color='blue', linestyle='--', label='Step Plot')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Step Plot Example')
plt.legend()
plt.show()
```

6. Stack Plot:

- **Use**: Displays multiple datasets stacked on top of each other in segments, illustrating cumulative totals or proportions over time or categories. It visually represents how parts make up a whole, often used in financial analysis or multi-series comparisons.
- **Function**: matplotlib.pyplot.stackplot()
- **Arguments**:
  - x: Data points for the x-axis.
  - y: List of arrays containing data points for each stack.
  - labels: Labels for each stack.
  - colors: Colors for each stack.
  - baseline: Either 'zero' or 'sym' to define the baseline.



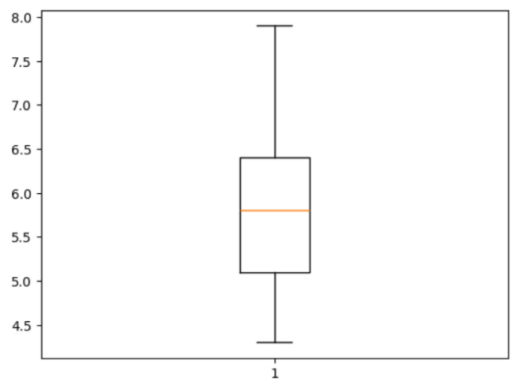
Example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(1, 6)
y1 = np.array([1, 2, 3, 4, 5])
y2 = np.array([1, 1, 2, 3, 5])
y3 = np.array([1, 3, 5, 7, 9])

plt.stackplot(x, y1, y2, y3, labels=['Stack 1', 'Stack 2', 'Stack 3'])
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Stack Plot Example')
plt.legend()
plt.show()
```

7. Box Plot:

- **Use:** Represents data distribution using five summary statistics: minimum, first quartile (Q1), median, third quartile (Q3), and maximum. It helps visualize outliers, variability, and skewness in data distributions across categories or variables.

- **Function**: matplotlib.pyplot.boxplot()
- **Arguments**:
  - x: Data points or array-like data to plot.
  - vert: Orientation of the box plot (True for vertical, False for horizontal).
  - patch_artist: If True, fill the box plot boxes with color.
  - showmeans: If True, display means as a point.
  - labels: Labels for each box plot.

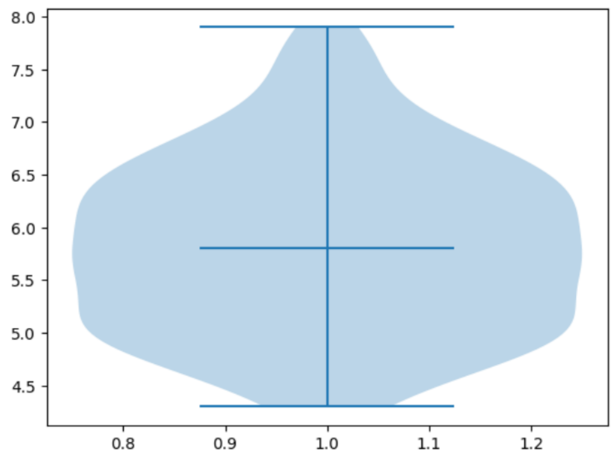


Example:

```
import matplotlib.pyplot as plt
import numpy as np

data = np.random.normal(0, 1, 100)  # Example data
plt.boxplot(data, vert=False, patch_artist=True, showmeans=True)
plt.xlabel('Value')
plt.title('Box Plot Example')
plt.show()
```

8. Violin Plot:

- **Use**: Combines aspects of a box plot and a density plot, showing the distribution of data across categories using kernel density estimation. It provides insights into data distribution, skewness, and the presence of multiple modes, suitable for comparing distributions across different groups.
- **Function**: matplotlib.pyplot.violinplot()
- **Arguments**:
  - dataset: List of data to plot as violins.
  - vert: Orientation of the violins (True for vertical, False for horizontal).
  - showmeans: If True, display means as a point.
  - showmedians: If True, display medians as a line.
  - showextrema: If True, display min and max as lines.

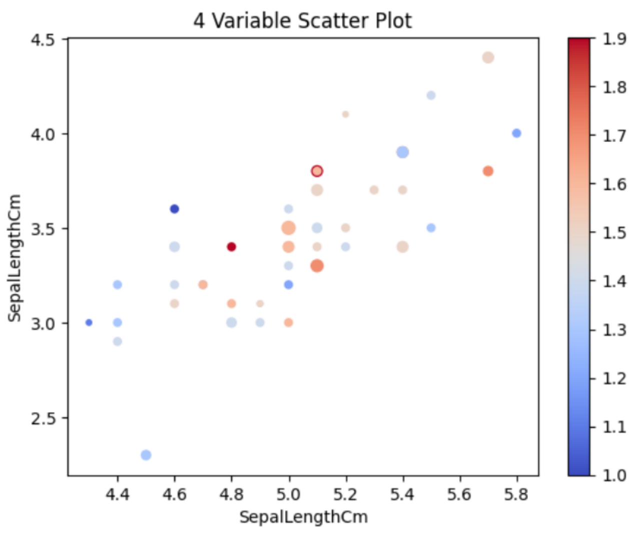


Example:

```
import matplotlib.pyplot as plt
import numpy as np

data = [np.random.normal(0, std, 100) for std in range(1, 4)]  # Example data
plt.violinplot(data, showmeans=True, showmedians=True)
plt.xlabel('Distribution')
plt.title('Violin Plot Example')
plt.show()
```

9. Scatter Plot:

- **Use**: Plots individual data points on a two-dimensional plane, where each point represents a value for two variables. It visually shows the relationship between variables, including patterns such as clusters, trends, correlations, or outliers. Optional features like color and size can represent additional dimensions of the data.
- **Function**: matplotlib.pyplot.scatter()
- **Arguments**:
  - x: Data points for the x-axis.
  - y: Data points for the y-axis.
  - s: Size of markers.
  - c: Color of markers.
  - marker: Marker style (e.g., 'o', '^', 's').
  - alpha: Transparency of markers (0 for transparent, 1 for opaque).



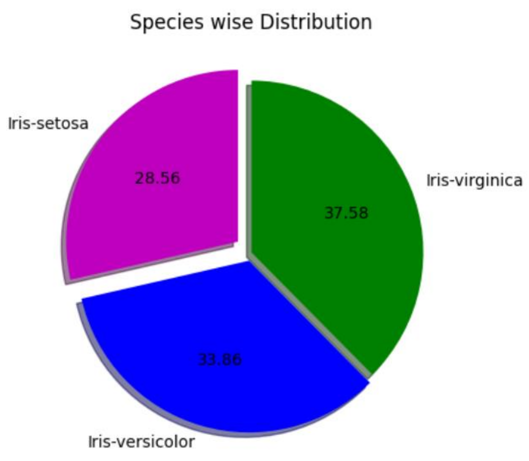
Example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(100)
y = np.random.randn(100)
sizes = np.random.rand(100) * 100
colors = np.random.rand(100)
```

```
plt.scatter(x, y, s=sizes, c=colors, alpha=0.5)
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Scatter Plot Example')
plt.show()
```

10. Pie Chart:

- **Use**: Divides a circle into sectors to illustrate numerical proportions, where each sector represents a category's contribution to the whole. It's effective for showing parts of a whole or comparing the relative sizes of categories, such as market shares or budget allocations.
- **Function**: matplotlib.pyplot.pie()
- **Arguments**:
  - x: Sizes of the slices (data).
  - labels: Labels for each slice.
  - colors: Colors for each slice.
  - autopct: Format string for percentage values.
  - shadow: If True, display shadow behind the pie.
  - startangle: Starting angle for the pie chart.



Example:

```
import matplotlib.pyplot as plt

sizes = [30, 20, 15, 35]  # Example data
labels = ['A', 'B', 'C', 'D']
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']

plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.title('Pie Chart Example')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

Subplots

Subplots in Matplotlib allow you to create multiple plots within the same figure. When using a 1-row, 2-column layout:

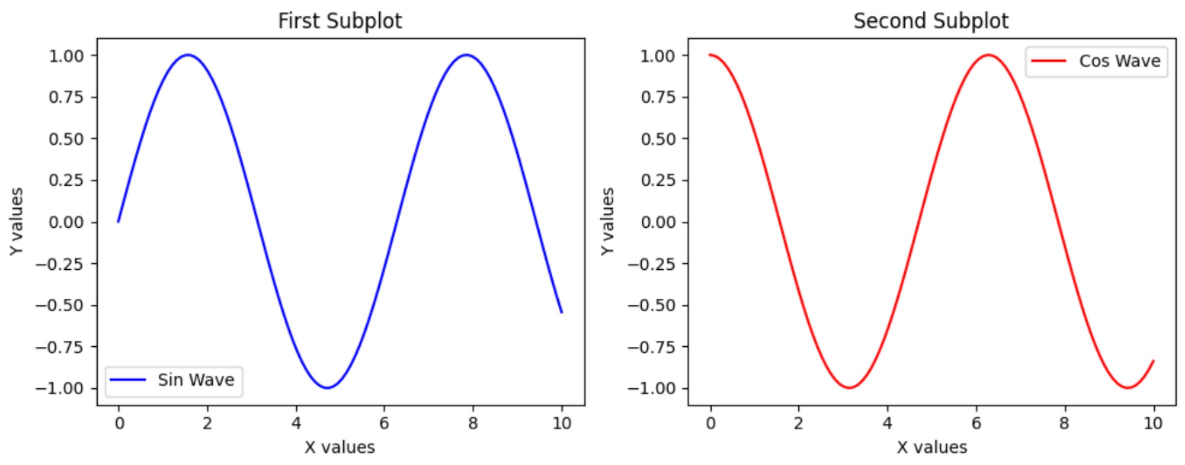
- **1 row, 2 columns**: This setup means you want to create two subplots side by side horizontally.
- **Index of column**: Specifies which subplot you're currently working on. For example, `index=1` refers to the first subplot (leftmost), and `index=2` refers to the second subplot (rightmost).

Here's an example:

```
import matplotlib.pyplot as plt
import numpy as np

# Generate some data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create a figure with 1 row and 2 columns
plt.figure(figsize=(10, 4))

# Plot on the first subplot (index 1)
plt.subplot(1, 2, 1)
plt.plot(x, y1, 'b-', label='Sin Wave')
plt.title('First Subplot')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.legend()

# Plot on the second subplot (index 2)
plt.subplot(1, 2, 2)
plt.plot(x, y2, 'r-', label='Cos Wave')
plt.title('Second Subplot')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.legend()

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```

In this example:

- `plt.subplot(1, 2, 1)` creates the first subplot in a 1-row, 2-column layout.
- `plt.subplot(1, 2, 2)` creates the second subplot next to the first one.
- Each subplot is then customized with its own title, labels, and legends as needed.



This approach allows for efficient organization and comparison of multiple plots within a single figure.

Creating a single plot with multiple lines on the same set of axes, rather than using subplots
Works for Both Line and Bar Chart

- **Single Plot with Multiple Lines**: This method uses a single set of axes (plt.plot()), where each plt.plot() command adds a new line to the existing plot.
- **Purpose**: It's used to compare multiple series or datasets within the same plot for direct visual comparison.

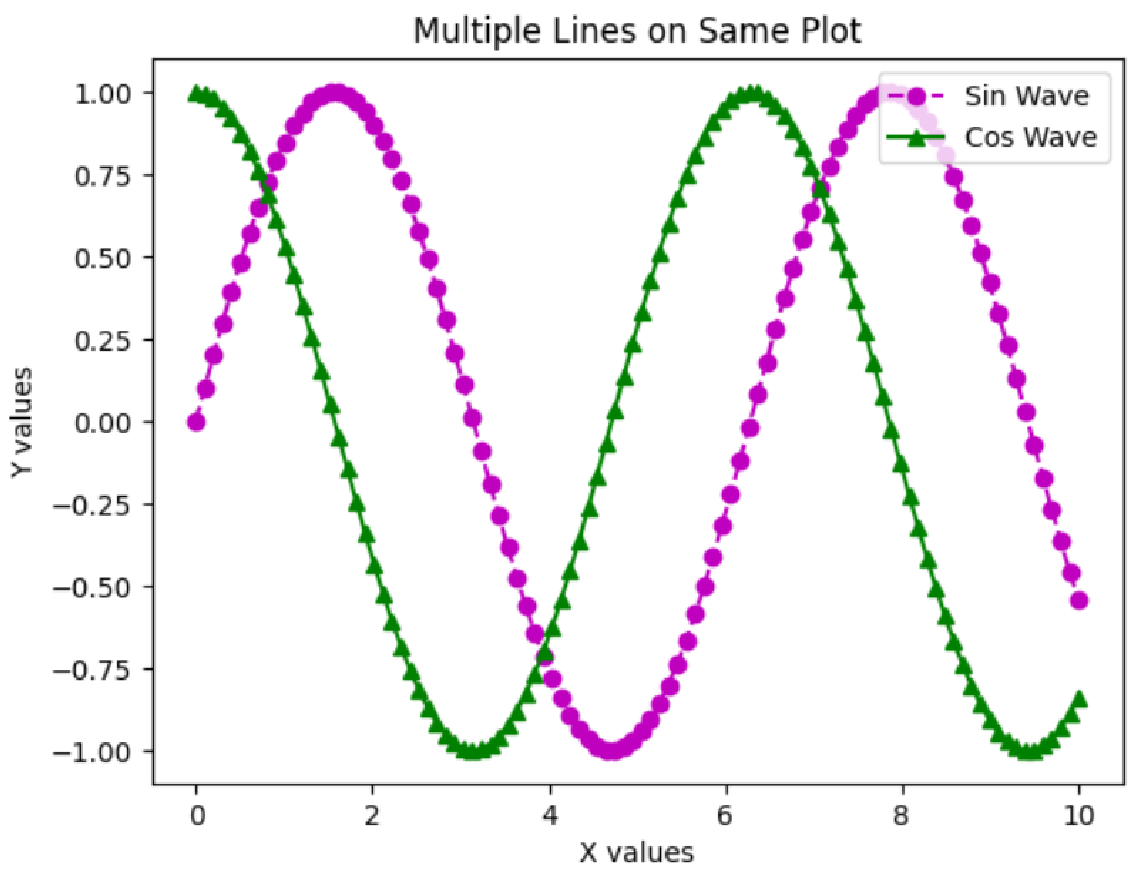
Here's another example using the provided code style:

```
import matplotlib.pyplot as plt
import numpy as np

# Example data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plotting multiple lines on the same plot
plt.plot(x, y1, color='m', marker='o', ls='--', label='Sin Wave')
plt.plot(x, y2, color='g', marker='^', label='Cos Wave')
plt.title('Multiple Lines on Same Plot')
plt.xlabel('X values')
plt.ylabel('Y values')
plt.legend(loc='upper right')  # Place legend in the upper right corner
plt.show()
```

In this example:

- plt.plot(x, y1, color='m', marker='o', ls='--', label='Sin Wave') plots the sine wave with a magenta color, dashed line style, and circular markers.

- plt.plot(x, y2, color='g', marker='^', label='Cos Wave') plots the cosine wave with a green color, solid line (default), and triangle markers.
- plt.legend(loc='upper right') positions the legend in the upper right corner of the plot.



This approach is useful for displaying multiple related datasets or comparing trends in data within the same plot area.

These notes encompass essential methods in NumPy, pandas, and Matplotlib, providing clear explanations and practical examples. From NumPy's array operations to pandas' powerful data manipulation techniques like pivot tables and melt function, and Matplotlib's versatile plotting capabilities including bar charts, histograms, and scatter plots, these resources aim to enhance understanding and proficiency in data analysis and visualization.

Creator: Ritika Bishnoi

LinkedIn: www.linkedin.com/in/ritikabishnoi1808

GitHub: https://github.com/RitikaBishnoi18