UNIT-I: Introduction to Data Science, Numpy & Pandas

1. Introduction to Data science: Facets of data

Data science is an interdisciplinary field that extracts meaningful insights from data using scientific methods, processes, algorithms, and systems. At its core, it combines mathematics, statistics, programming, domain expertise, and machine learning to analyze and interpret complex datasets.

A key concept in data science is understanding the **facets of data**, which form the foundation for analysis and decision-making. These facets represent the diverse characteristics and dimensions of data. Below are the primary facets of data:

Types of Data

- **Structured Data**: Data organized in rows and columns, typically stored in relational databases. Examples include customer records, sales data, and financial transactions.
- **Unstructured Data**: Data that does not follow a predefined format. Examples include text, images, videos, and social media posts.
- **Semi-Structured Data**: Data with some organizational properties but no fixed schema, such as JSON, XML, or CSV files.

Sources of Data

- **Primary Data**: Data collected firsthand for a specific purpose, such as surveys, experiments, or interviews.
- **Secondary Data**: Data collected by others and repurposed for analysis, such as publicly available datasets, research studies, or government reports.
- **Real-Time Data**: Data that is generated and processed instantly, often used in IoT systems, stock trading platforms, or monitoring systems.

Dimensions of Data

- **Volume**: Refers to the amount of data generated, often measured in terabytes or petabytes.
- **Variety**: Indicates the different types and formats of data, such as images, text, or numerical data.
- **Velocity**: The speed at which data is generated and processed, critical for real-time analytics.
- **Veracity**: The quality and reliability of data, considering inaccuracies, biases, or inconsistencies.
- **Value**: The usefulness of the data for generating insights and driving decisions.

Characteristics of Data

- **Granularity**: The level of detail in the data. High granularity means detailed data, while low granularity refers to summarized or aggregated data.
- **Sparsity**: The ratio of missing or zero values in a dataset. Sparse data often needs specialized techniques for analysis.

- **Dimensionality**: The number of variables or features in a dataset, critical for determining suitable analysis methods.

Lifecycle of Data

- **Collection**: Gathering raw data from various sources, such as sensors, web scraping, or manual entry.
- **Storage**: Organizing and storing data in databases, data lakes, or cloud platforms.
- **Processing**: Cleaning and transforming data to make it suitable for analysis.
- **Analysis**: Applying statistical and machine learning techniques to uncover patterns and insights.
- **Visualization**: Communicating insights through charts, graphs, and dashboards.
- **Decision-Making**: Using insights to guide business or research decisions.

Ethical Considerations of Data

- **Privacy**: Ensuring data is used responsibly and in compliance with laws like GDPR or CCPA.
- **Bias**: Identifying and mitigating biases in data collection or analysis.
- **Security**: Protecting sensitive data from breaches or unauthorized access.

Understanding the facets of data is critical in data science, as it lays the groundwork for effective analysis and insight generation. By considering aspects like data type, source, dimension, and ethical use, data scientists can harness the power of data to address real-world challenges and drive innovation.

2. **Data Science Process**

The **data science process** is a systematic approach to solving problems and extracting insights from data. It outlines the steps required to turn raw data into actionable knowledge. This process ensures that data scientists work efficiently and effectively, maintaining consistency and quality in their analyses. Below is a detailed overview of the key stages in the data science process:

Problem Definition

- **Objective Setting**: Clearly define the problem or question to be answered. Understand the business or research goals and identify the key performance indicators (KPIs).
- **Stakeholder Involvement**: Collaborate with domain experts to align expectations and ensure that the analysis addresses practical needs.
- **Outcome Specification**: Determine the expected outputs and their implications for decision-making.

Data Collection

- **Identify Data Sources**: Gather data from relevant sources, such as databases, APIs, surveys, or external datasets.
- **Primary and Secondary Data**: Determine whether new data collection is needed or if existing datasets suffice.

- **Data Acquisition Tools**: Use tools like web scraping, SQL queries, or cloud platforms to extract data.

Data Preparation

- **Data Cleaning**: Handle missing values, duplicate records, and outliers. Standardize formats and resolve inconsistencies.
- **Data Transformation**: Convert raw data into a usable format through normalization, scaling, or encoding categorical variables.
- **Feature Engineering**: Create new variables or features that enhance the model's predictive power.
- **Exploratory Data Analysis (EDA)**: Use statistical and visualization techniques to understand data distribution, relationships, and trends.

Data Exploration and Analysis

- **Hypothesis Testing**: Formulate and test hypotheses to validate assumptions about the data.
- **Statistical Analysis**: Apply statistical methods to identify patterns, correlations, and anomalies.
- **Dimensionality Reduction**: Reduce the number of features while retaining important information (e.g., using PCA or t-SNE).

Modeling

- **Model Selection**: Choose appropriate models based on the problem type (e.g., regression, classification, clustering, or time-series analysis).
- **Model Training**: Train the model on the prepared dataset using algorithms like linear regression, decision trees, or neural networks.
- **Model Evaluation**: Assess the model's performance using metrics like accuracy, precision, recall, F1-score, or mean squared error (MSE).
- **Hyperparameter Tuning**: Optimize the model by adjusting hyperparameters to improve performance.

Validation and Testing

- **Cross-Validation**: Divide the dataset into training, validation, and test sets to ensure robustness and generalizability.
- **Overfitting/Underfitting Analysis**: Check for signs of overfitting (poor performance on new data) or underfitting (insufficient learning).
- **Performance Metrics**: Compare models and select the best-performing one for deployment.

Deployment

- **Integration**: Implement the final model into production systems, such as applications, APIs, or dashboards.

- **Automation**: Automate data ingestion, model retraining, and reporting to streamline processes.
- **Monitoring**: Continuously track model performance in the real world and retrain as needed.

Communication and Reporting

- **Data Visualization**: Present findings using charts, graphs, or dashboards to convey insights clearly.
- **Stakeholder Communication**: Translate technical results into actionable recommendations for non-technical stakeholders.
- **Documentation**: Record the process, assumptions, and findings to ensure reproducibility and knowledge transfer.

Iteration and Improvement

- **Feedback Loop**: Gather feedback from stakeholders to refine the model or analysis.
- **Continuous Improvement**: Update models and processes as new data becomes available or as requirements evolve.

3. **Introduction to Numpy**
   NumPy (short for **Numerical Python**) is one of the most popular and fundamental libraries in Python for numerical computing. It provides support for working with large, multidimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays efficiently.

Key Features of NumPy

1. **N-Dimensional Arrays**: Provides the ndarray object, which allows for efficient storage and operations on large datasets.
2. **Mathematical Functions**: Includes a wide range of functions for statistical, linear algebra, and Fourier analysis.
3. **Broadcasting**: Allows for vectorized operations, making it unnecessary to write loops for element-wise operations.
4. **Interoperability**: Works seamlessly with other Python libraries such as Pandas, Matplotlib, and Scikit-learn.
5. **High Performance**: Built on C, ensuring speed and efficiency for numerical computations.

Installing NumPy

To install NumPy, use pip:

pip install Numpy

Core Concepts in NumPy

1. The ndarray Object

The core of NumPy is the ndarray (n-dimensional array), which is a grid of values of the same data type.

```
import numpy as np #

Create a NumPy array

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Array Creation

NumPy provides several functions to create arrays:

- **From Lists or Tuples**:

```
arr = np.array([1, 2, 3])  # 1D array

mat = np.array([[1, 2], [3, 4]])  # 2D array
```

Using Helper Functions:

```
np.zeros((2, 3))     # Array of zeros
np.ones((3, 3))      # Array of ones
np.arange(0, 10, 2)   # Array of evenly spaced values
np.linspace(0, 1, 5)  # Array of 5 values evenly spaced between 0 and 1
```

Array Operations

- **Element-Wise Operations**:

```
arr = np.array([1, 2, 3])
print(arr + 2)  # [3, 4, 5]
print(arr * 2)  # [2, 4, 6]
```

Matrix Multiplication:

```
mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[5, 6], [7, 8]])
print(np.dot(mat1, mat2))  # Matrix product
```

Array Indexing and Slicing

- **Indexing**

```
arr = np.array([10, 20, 30, 40])

print(arr[1])  # 20
```

Slicing:

```
print(arr[1:3])  # [20, 30]
```

Shape and Size

```
arr = np.array([[1, 2], [3, 4]])

print(arr.shape)  # (2, 2)

print(arr.size)   # 4
```

Reshape

```
print(arr.reshape(4, 1))  # Reshape to a 4x1 array
```

Aggregation

```
print(arr.sum())  # Sum of all elements

print(arr.mean())  # Mean value

print(arr.max())  # Maximum value
```

- **Attributes**
  NumPy arrays, represented by the ndarray object, come with various attributes that provide useful metadata about the array. These attributes allow you to better understand and manipulate arrays in an efficient and organized manner. Below are the key attributes of a NumPy array:

ndarray.shape

- **Description**: Returns a tuple representing the dimensions of the array (number of rows, columns, etc.).
- **Example**:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # Output: (2, 3) (2 rows, 3 columns)
```

ndarray.ndim

- **Description**: Returns the number of dimensions (axes) of the array.
- **Example**

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.ndim)  # Output: 2 (2D array)
```

ndarray.size

- **Description**: Returns the total number of elements in the array (product of all dimensions).
- **Example**

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.size)  # Output: 6
```

ndarray.dtype

- **Description**: Returns the data type of the elements in the array.
- **Example**:

```
arr = np.array([1, 2, 3])
print(arr.dtype)  # Output: int64 (or int32, depending on the platform)
```

ndarray.itemsize

- **Description**: Returns the size (in bytes) of each element in the array.
- **Example**:

```
arr = np.array([1, 2, 3])
print(arr.itemsize)  # Output: 8 (if dtype is int64, 4 if int32)
```

ndarray.nbytes

- **Description**: Returns the total number of bytes consumed by the array (size × itemsize).
- **Example**

```
arr = np.array([1, 2, 3])
print(arr.nbytes)  # Output: 24 (3 elements × 8 bytes each)
```

ndarray.T (Transpose)

- **Description**: Returns the transposed version of the array (only for 2D or higher dimensions).
- **Example**:

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

ndarray.flags

- **Description**: Provides information about the memory layout of the array.
- **Example**:

```
arr = np.array([1, 2, 3])
print(arr.flags)
```

ndarray.real and ndarray.imag

- **Description**: Return the real and imaginary parts of the array, respectively (useful for complex numbers).
- **Example**

```
arr = np.array([1 + 2j, 3 + 4j])
print(arr.real)  # Output: [1. 3.]
print(arr.imag)  # Output: [2. 4.]
```

ndarray.strides

- **Description**: Returns the number of bytes to step in each dimension when traversing an array.
- **Example**

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.strides)  # Output: (24, 8) (depending on platform)
```

ndarray.base

- **Description**: If the array is a view of another array, this attribute points to the original array.
- **Example**

```
arr = np.array([1, 2, 3])
view = arr[1:]
print(view.base is arr)  # Output: True
```

ndarray.flat

- **Description**: Returns a 1D iterator over the array.
- **Example**

```
arr = np.array([[1, 2], [3, 4]])
for element in arr.flat:
  print(element)  # Outputs: 1 2 3 4
```

4. **Numpy Arrays objects:** The core of NumPy is its array object, called the **ndarray (N-dimensional array)**. This object is a powerful and versatile data structure for handling numerical data efficiently. It is the backbone of most numerical operations in NumPy.

- **Creating Arrays**

a. From Python Lists or Tuples

You can create a NumPy array using np.array():

```
import numpy as np

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

# From a nested list (2D array)
arr_2d = np.array([[1, 2], [3, 4]])
```

b. Using NumPy Helper Functions

NumPy provides several functions to create arrays:

- **np.zeros()**: Creates an array filled with zeros.

```
zeros_arr = np.zeros((2, 3))  # 2 rows, 3 columns
```

- **np.ones()**: Creates an array filled with ones.

```
ones_arr = np.ones((3, 3))
```

- **np.arange()**: Creates an array with evenly spaced values.

```
arr = np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
```

- **np.linspace()**: Creates an array of evenly spaced values between a start and end point.

  arr = np.linspace(0, 1, 5) # [0., 0.25, 0.5, 0.75, 1.]

c. Random Array Creation

- **np.random.rand()**: Creates an array of random values between 0 and 1.

  rand_arr = np.random.rand(2, 3)

- **np.random.randint()**: Creates an array of random integers.

  rand_int_arr = np.random.randint(1, 10, (3, 3))

Dimensions in NumPy Arrays

- **1D Array**: A single list of elements.

  arr_1d = np.array([1, 2, 3])

- **2D Array**: A list of lists (like a matrix).

  arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

- **3D Array**: A list of matrices.

  arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

- **basic operations (Array Join, split, search, sort)**

Joining Arrays

Joining arrays means combining multiple arrays into a single array.

a. Using np.concatenate()

Joins two or more arrays along a specified axis.

```
import numpy as np

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

# Join along the default axis (1D arrays)
joined = np.concatenate((arr1, arr2))
print(joined)  # Output: [1 2 3 4 5 6]
```

b. Using np.vstack() (Vertical Stack)

Stacks arrays vertically (row-wise).

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

vstacked = np.vstack((arr1, arr2))
print(vstacked)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

c. Using np.hstack() (Horizontal Stack)

Stacks arrays horizontally (column-wise).

```
hstacked = np.hstack((arr1, arr2))
print(hstacked)  # Output: [1 2 3 4 5 6]
```

d. Using np.dstack() (Depth Stack)

Stacks arrays along the third axis.

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

dstacked = np.dstack((arr1, arr2))
print(dstacked)
# Output:
# [[[1 4]
#   [2 5]
#   [3 6]]]
```

---

2. Splitting Arrays

Splitting divides an array into multiple smaller arrays.

a. Using np.split()

Splits an array into equal parts.

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

# Split into 3 parts
split = np.split(arr, 3)
```

```
print(split)  # Output: [array([1, 2]), array([3, 4]), array([5, 6])]
```

b. Using np.array_split()

Splits an array into unequal parts.

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

# Split into 3 parts (last part may be uneven)
split = np.array_split(arr, 3)
print(split)  # Output: [array([1, 2, 3]), array([4, 5]), array([6, 7])]
```

c. Splitting Multi-Dimensional Arrays

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

# Horizontal split
h_split = np.hsplit(arr, 2)
print(h_split)
# Output: [array([[1], [3], [5]]), array([[2], [4], [6]])]

# Vertical split
v_split = np.vsplit(arr, 3)
print(v_split)
# Output: [array([[1, 2]]), array([[3, 4]]), array([[5, 6]])]
```

---

3. Searching in Arrays

NumPy provides methods to find elements and their indices.

a. Using np.where()

Finds the indices of elements that satisfy a condition.

```
arr = np.array([10, 20, 30, 40, 50])

# Find indices where values are greater than 25
indices = np.where(arr > 25)
print(indices)  # Output: (array([2, 3, 4]),)
```

b. Using np.searchsorted()

Searches for a value and returns the index where it should be inserted to maintain order.

```
arr = np.array([1, 3, 5, 7])

# Find the index to insert 4
index = np.searchsorted(arr, 4)
```

```
print(index)  # Output: 2
```

c. Using np.isin()

Checks if elements exist in another array.

```
arr = np.array([1, 2, 3, 4, 5])
result = np.isin(arr, [2, 4, 6])
print(result)  # Output: [False  True False  True False]
```

---

4. Sorting Arrays

Sorting arranges elements in ascending or descending order.

a. Using np.sort()

Sorts an array along a specified axis.

```
arr = np.array([5, 2, 9, 1, 7])
sorted_arr = np.sort(arr)
print(sorted_arr)  # Output: [1 2 5 7 9]
```

b. Sorting Multi-Dimensional Arrays

Sorts along each row or column.

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

# Sort along rows
row_sorted = np.sort(arr, axis=1)
print(row_sorted)
# Output:
# [[1 2 3]
#  [4 5 6]]

# Sort along columns
col_sorted = np.sort(arr, axis=0)
print(col_sorted)
# Output:
# [[3 2 1]
#  [6 5 4]]
```

c. Using np.argsort()

Returns the indices that would sort the array.

```
arr = np.array([5, 2, 9, 1, 7])
```

```
indices = np.argsort(arr)
print(indices)  # Output: [3 1 0 4 2]
```

- **Iterating**

NumPy arrays support iteration just like Python lists, but with additional functionality for multidimensional arrays.

a. Iterating Through 1D Arrays

You can iterate over the elements of a 1D array using a simple for loop:

```
import numpy as np

arr = np.array([1, 2, 3, 4])
for element in arr:
  print(element)
# Output: 1, 2, 3, 4
```

b. Iterating Through 2D Arrays

Iteration in 2D arrays occurs row by row.

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
for row in arr:
  print(row)
# Output:
# [1 2]
# [3 4]
# [5 6]
```

c. Iterating Through Each Element in a Multi-Dimensional Array

To iterate over each element in a multidimensional array, use the nditer() function:

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
for element in np.nditer(arr):
  print(element)
# Output: 1, 2, 3, 4, 5, 6
```

d. Iterating With Indices

Use the ndenumerate() function to get both the index and the value during iteration.

```
arr = np.array([[1, 2], [3, 4]])
for index, value in np.ndenumerate(arr):
  print(index, value)
# Output:
```

```
# (0, 0) 1
# (0, 1) 2
# (1, 0) 3
# (1, 1) 4
```

- **Copying arrays**

NumPy arrays can be copied in two main ways: **shallow copy** and **deep copy**.

a. Shallow Copy (View)

A shallow copy creates a new array object that refers to the same data as the original array. Changes in one array will affect the other.

```
arr = np.array([1, 2, 3])
shallow_copy = arr.view()

shallow_copy[0] = 99
print(arr)         # Output: [99  2  3]
print(shallow_copy)  # Output: [99  2  3]
```

b. Deep Copy

A deep copy creates a new array object with a completely independent copy of the data. Changes in one array do not affect the other.

```
arr = np.array([1, 2, 3])
deep_copy = arr.copy()

deep_copy[0] = 99
print(arr)         # Output: [1  2  3]
print(deep_copy)     # Output: [99  2  3]
```

c. Check for Shared Memory

You can check whether two arrays share memory using the np.may_share_memory() function:

```
arr = np.array([1, 2, 3])
shallow_copy = arr.view()
deep_copy = arr.copy()

print(np.may_share_memory(arr, shallow_copy))  # Output: True
print(np.may_share_memory(arr, deep_copy))     # Output: False
```

- **Identity array**

An **identity array** is a square matrix in which all the diagonal elements are 1 and all other elements are 0. NumPy provides the np.identity() and np.eye() functions to create identity arrays.

a. Using np.identity()

Creates a square identity matrix of a specified size.

```
identity_matrix = np.identity(4)
print(identity_matrix)
# Output:
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]
```

b. Using np.eye()

Allows you to specify the number of rows and columns, and the position of the diagonal.

```
eye_matrix = np.eye(3, 4, k=1)  # 3 rows, 4 columns, diagonal shifted by 1
print(eye_matrix)
# Output:
# [[0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]
```

5. **Eye function Pandas: Exploring Data using Series**

Part 1: Eye Function in NumPy

The **np.eye()** function in NumPy is used to create a 2D array with ones on the diagonal and zeros elsewhere. It is similar to the identity matrix but offers more flexibility.

Syntax

numpy.eye(N, M=None, k=0, dtype=float, order='C')

Parameters

- **N**: Number of rows.
- **M**: (Optional) Number of columns. Defaults to N (square matrix).
- **k**: Index of the diagonal:
  - k=0 (default): Main diagonal.
  - k>0: Above the main diagonal.
  - k<0: Below the main diagonal.
- **dtype**: Data type of the returned array. Default is float.
- **order**: Memory layout ('C' for row-major, 'F' for column-major).

Examples

a. Basic Usage

```
import numpy as np
```

```
matrix = np.eye(4)
print(matrix)
# Output:
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]
```

b. Non-Square Matrix

```
python
Copy code
matrix = np.eye(3, 5)
print(matrix)
# Output:
# [[1. 0. 0. 0. 0.]
#  [0. 1. 0. 0. 0.]
#  [0. 0. 1. 0. 0.]]
```

c. Offset Diagonal (k)

```
matrix = np.eye(4, k=1)  # Diagonal above the main diagonal
print(matrix)
# Output:
# [[0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]
#  [0. 0. 0. 0.]]
```

Part 2: Exploring Data Using Pandas Series

The **Pandas Series** is a one-dimensional labeled array capable of holding any data type (e.g., integers, floats, strings, etc.). It is similar to a NumPy array but with labels (index).

Creating a Pandas Series

a. From a List

```
import pandas as pd

data = [10, 20, 30, 40]
series = pd.Series(data)
print(series)
# Output:
# 0   10
# 1   20
# 2   30
```

```
# 3   40
# dtype: int64
```

b. From a Dictionary

```
data = {'a': 10, 'b': 20, 'c': 30}
series = pd.Series(data)
print(series)
# Output:
# a   10
# b   20
# c   30
# dtype: int64
```

c. Custom Index

```
data = [10, 20, 30]
index = ['x', 'y', 'z']
series = pd.Series(data, index=index)
print(series)
# Output:
# x   10
# y   20
# z   30
# dtype: int64
```

Accessing and Manipulating Data in a Series

a. Accessing Elements

- By index:

  ```
  print(series['x'])  # Output: 10
  ```

- By position:

  ```
  print(series[0])  # Output: 10
  ```

b. Slicing

```
print(series['x':'y'])  # Output: x   10
                #       y   20
```

c. Filtering

```
print(series[series > 15])  # Output:
# y   20
# z   30
```

Performing Operations on a Series

a. Arithmetic Operations

Pandas Series supports element-wise arithmetic operations:

```
series1 = pd.Series([1, 2, 3])
series2 = pd.Series([4, 5, 6])

result = series1 + series2
print(result)
# Output:
# 0    5
# 1    7
# 2    9
# dtype: int64
```

b. Statistical Operations

- Mean:

  ```
  print(series.mean())  # Output: 20.0
  ```

- Maximum:

  ```
  print(series.max())  # Output: 30
  ```

- Sum:

  ```
  print(series.sum())  # Output: 60
  ```

6. **Exploring Data using Data Frames, Index objects, Re index, Drop Entry**
   Pandas **DataFrames** are 2-dimensional labeled data structures with columns of potentially different types. They are the primary structure for handling and analyzing data in Python.

Creating a DataFrame

a. From a Dictionary of Lists

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'San Francisco', 'Los Angeles']
}

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

```
# Output:
#      Name  Age         City
# 0    Alice   25      New York
# 1      Bob   30  San Francisco
# 2  Charlie   35   Los Angeles
```

b. From a List of Dictionaries

```
data = [
   {'Name': 'Alice', 'Age': 25, 'City': 'New York'},
   {'Name': 'Bob', 'Age': 30, 'City': 'San Francisco'},
   {'Name': 'Charlie', 'Age': 35, 'City': 'Los Angeles'}
]

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

---

2. Index Objects

The **Index** in a DataFrame uniquely identifies rows and columns.

Accessing the Index

```
print(df.index)  # Output: RangeIndex(start=0, stop=3, step=1)
```

Setting a Custom Index

```
df.set_index('Name', inplace=True)
print(df)
# Output:
#          Age         City
# Name
# Alice      25      New York
# Bob        30  San Francisco
# Charlie    35   Los Angeles
```

Resetting the Index

```
df.reset_index(inplace=True)
print(df)
```

---

3. Reindexing

Reindexing changes the row or column labels of a DataFrame.

Changing the Row Index

```
new_index = ['a', 'b', 'c']
df.index = new_index
print(df)
# Output:
```

```
#    Name  Age         City
# a  Alice   25      New York
# b    Bob   30  San Francisco
# c Charlie   35   Los Angeles
```

Using reindex()

You can align a DataFrame to a new index:

```
new_index = [0, 1, 2, 3]
reindexed_df = df.reindex(new_index)
print(reindexed_df)
# Output:
#      Name   Age         City
# 0    Alice  25.0      New York
# 1      Bob  30.0  San Francisco
# 2  Charlie  35.0   Los Angeles
# 3      NaN   NaN          NaN
```

---

4. Dropping Entries

Dropping Rows

To drop a row, use drop() with the row label:

```
df = pd.DataFrame(data)
df_dropped = df.drop(1)  # Drops the second row
print(df_dropped)
# Output:
#      Name  Age         City
# 0    Alice   25      New York
# 2  Charlie   35   Los Angeles
```

Dropping Columns

Use axis=1 to drop columns:

```
df_dropped_col = df.drop('City', axis=1)
print(df_dropped_col)
# Output:
#      Name  Age
# 0    Alice   25
# 1      Bob   30
# 2  Charlie   35
```

In-Place Drop

Perform the operation in-place:

```
df.drop(0, inplace=True)  # Drops the first row directly in df
print(df)
```

---

5. Exploring the Data

a. Viewing Basic Information

- df.head(n): Displays the first n rows (default is 5).

  ```
  print(df.head(2))
  ```

- df.tail(n): Displays the last n rows.

  ```
  print(df.tail(2))
  ```

- df.info(): Summary of the DataFrame, including data types and memory usage.

  ```
  print(df.info())
  ```

b. Descriptive Statistics

- df.describe(): Summarizes numerical columns.

  ```
  print(df.describe())
  ```

c. Shape and Size

- df.shape: Returns (rows, columns).

  ```
  print(df.shape)  # Output: (3, 3)
  ```

- df.size: Total number of elements.

  ```
  print(df.size)  # Output: 9
  ```

---

6. Selecting and Accessing Data

a. Selecting Columns

```
print(df['Name'])  # Access the 'Name' column
```

b. Selecting Rows

- Using loc[] (label-based):

  ```
  print(df.loc[0])  # Access row with index 0
  ```

- Using iloc[] (integer-location based):

```
print(df.iloc[1])  # Access the second row
```

c. Boolean Indexing

```
print(df[df['Age'] > 25])  # Filter rows where Age > 25
```

7. Data Alignment

Data alignment in Pandas ensures that operations between DataFrames or Series align on their labels (indexes) by default.

Automatic Alignment

When performing arithmetic operations between objects with different indexes, Pandas aligns the data based on the index.

```
import pandas as pd

s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])

result = s1 + s2
print(result)
# Output:
# a    NaN
# b    6.0
# c    8.0
# d    NaN
dtype: float64
```

2. Filling Missing Data

You can fill missing values resulting from misaligned indexes using the fill_value parameter.

```
result = s1.add(s2, fill_value=0)
print(result)
# Output:
# a    1.0
# b    6.0
# c    8.0
# d    6.0
dtype: float64
```

3. Aligning Data Explicitly

Use the align() method to align two objects explicitly.

```
aligned_s1, aligned_s2 = s1.align(s2, fill_value=0)
print(aligned_s1)
print(aligned_s2)
```

8. Rank and Sort

Sorting

a. Sorting by Index

Use sort_index() to sort rows or columns by their index.

```
df = pd.DataFrame({'A': [3, 1, 2]}, index=['b', 'c', 'a'])
sorted_df = df.sort_index()
print(sorted_df)
# Output:
#    A
# a  2
# b  3
# c  1
```

b. Sorting by Values

Use sort_values() to sort rows by column values.

```
sorted_df = df.sort_values(by='A')
print(sorted_df)
# Output:
#    A
# c  1
# a  2
# b  3
```

c. Sorting in Descending Order

```
sorted_df = df.sort_values(by='A', ascending=False)
print(sorted_df)
```

---

2. Ranking

Ranking assigns a rank to elements based on their value.

a. Default Ranking

```
s = pd.Series([7, 1, 2, 7])
ranked = s.rank()
print(ranked)
# Output:
# 0    3.5
# 1    1.0
# 2    2.0
# 3    3.5
dtype: float64
```

b. Handling Ties

You can specify how to handle ties using the method parameter:

- **average**: Default, assigns the average rank to tied values.
- **min**: Assigns the minimum rank.
- **max**: Assigns the maximum rank.
- **first**: Assigns ranks in the order they appear.

```
ranked_min = s.rank(method='min')
print(ranked_min)
# Output:
# 0    3.0
# 1    1.0
# 2    2.0
# 3    3.0
dtype: float64
```

9. Summary Statistics

Pandas provides several functions to compute descriptive statistics.

1. Aggregation Functions

These return a single value for each column or row.

- **mean()**: Average.
- **sum()**: Sum of values.
- **min() and max()**: Minimum and maximum values.
- **std()**: Standard deviation.

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

print(df.mean())  # Column-wise mean
# Output:
# A    2.0
# B    5.0
# C    8.0
dtype: float64

print(df.mean(axis=1))  # Row-wise mean
```

2. Cumulative Functions

Cumulative functions return cumulative sums, products, or counts.

```
print(df['A'].cumsum())
# Output:
# 0    1
# 1    3
# 2    6
Name: A, dtype: int64
```

3. Describe Method

The describe() method generates a summary of statistics for numerical columns.

```
print(df.describe())
# Output:
#              A    B    C
# count  3.000000  3.0  3.0
# mean   2.000000  5.0  8.0
# std    1.000000  1.0  1.0
# min    1.000000  4.0  7.0
# 25%    1.500000  4.5  7.5
# 50%    2.000000  5.0  8.0
# 75%    2.500000  5.5  8.5
# max    3.000000  6.0  9.0
```

4. Value Counts

To count occurrences of unique values:

```
s = pd.Series([1, 2, 2, 3, 3, 3])
print(s.value_counts())
# Output:
# 3    3
# 2    2
# 1    1
dtype: int64
```

10. **Index Hierarchy Data Acquisition: Gather information from different sources, Web APIs, Open Data Sources, Web Scrapping.**
    Data acquisition refers to the process of collecting data from various sources to be analyzed. There are multiple ways to gather data, ranging from using **Web APIs**, **Open Data Sources**, to performing **Web Scraping**. Each method has its use case depending on the source and the desired format.

Index Hierarchy

An **Index Hierarchy** (or MultiIndex) in Pandas allows for multiple levels of indexing in rows or columns, which can be used to manage complex data and hierarchies. The index is like a key or label that can represent hierarchical relationships between different rows or columns of data.

Creating a MultiIndex

To create a **MultiIndex**, you can pass a list of tuples or arrays to the pd.MultiIndex.from_tuples() or pd.MultiIndex.from_arrays() methods.

```
import pandas as pd

# Creating a MultiIndex from tuples
index = pd.MultiIndex.from_tuples([('A', 1), ('A', 2), ('B', 1), ('B', 2)], names=['Letter', 'Number'])
df = pd.DataFrame({'Data': [10, 20, 30, 40]}, index=index)
print(df)
```

Output:

```
Data
Letter Number
A      1      10
       2      20
B      1      30
       2      40
```

Accessing Data from a MultiIndex

You can access rows using the loc[] method:

```
print(df.loc['A'])  # Access all rows where 'Letter' is 'A'
print(df.loc[('A', 1)])  # Access row where 'Letter' is 'A' and 'Number' is 1
```

---

2. Data Acquisition

a. Gathering Information from Web APIs

Web APIs (Application Programming Interfaces) provide a way to retrieve data from external servers. They are often used for acquiring real-time data from various services (such as weather, finance, social media, etc.).

Using requests Library

The requests library is commonly used for interacting with web APIs.

Example: Fetching Data from a JSON API

```
import requests

# API endpoint (example: OpenWeatherMap API)
url = "https://api.openweathermap.org/data/2.5/weather?q=London&appid=your_api_key"

response = requests.get(url)

# Check if the request was successful (status code 200)
if response.status_code == 200:
   data = response.json()  # Parse the JSON response
   print(data)
else:
   print("Failed to retrieve data")
```

Note: Replace your_api_key with an actual API key from the OpenWeatherMap API.

---

b. Open Data Sources

Open Data refers to publicly available datasets that are provided by governments, organizations, or research institutions. These datasets can be used for analysis, research, or machine learning purposes.

- **Government Open Data Portals**: Many governments provide open datasets, such as the US Data.gov and EU Open Data Portal.
- **Research Datasets**: Sites like Kaggle and UCI Machine Learning Repository provide datasets for machine learning and research.

To acquire open data, you can either download the files manually from the portals or use APIs, depending on availability.

---

c. Web Scraping

Web Scraping is the process of extracting data from websites that don't provide an API. It involves fetching the HTML content of a webpage and extracting specific data points.

Using BeautifulSoup for Web Scraping

To scrape data from a website, you need the requests and BeautifulSoup libraries. Here's an example of scraping a simple webpage:

```
import requests
from bs4 import BeautifulSoup

# Send GET request to the website
url = "https://example.com"
```

```
response = requests.get(url)

# Parse the content of the webpage
soup = BeautifulSoup(response.text, 'html.parser')

# Find the specific element you want (example: all paragraphs)
paragraphs = soup.find_all('p')

# Extract the text from each paragraph
for para in paragraphs:
   print(para.get_text())
```

Important Notes:

- Be respectful when scraping, adhere to the website's robots.txt rules.
- Consider adding delays between requests to avoid overloading the website.
- Some websites may block IPs or implement measures to prevent scraping.

---

3. Combining Data from Multiple Sources

Once you have gathered data from different sources (such as Web APIs, Open Data, or Web Scraping), you can combine and clean the data using Pandas.

Concatenating DataFrames

If you have multiple DataFrames, you can concatenate them:

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

result = pd.concat([df1, df2], ignore_index=True)
print(result)
```

Output:

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

Merging DataFrames

You can merge DataFrames on common columns, similar to SQL joins.

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'Age': [25, 30, 35]})

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

Output:

```
   ID    Name  Age
0   1   Alice   25
1   2     Bob   30
```