Week 1: Introduction to Jupyter Notebook, NumPy, and Pandas

Jupyter Notebook, NumPy, and Pandas:

Why it is important to learn them:

- **Jupyter Notebook**: Interactive Data Exploration
- **NumPy**: Efficient Numerical Computing
- **Pandas**: Data Manipulation and Analysis

Understanding Jupyter Notebooks

- Jupyter Notebook is a web-based interactive computing environment that allows users to create and share documents containing live code, equations, visualizations, and narrative text.

- Jupyter Notebook is not just a code editor; it revolutionizes the way we work with code, data, and explanations.

- Jupyter Notebook is particularly well-suited for various data-related tasks, including data exploration, analysis, and visualization.

Jupyter Notebooks Features

Live Code Execution:

- Jupyter allows users to execute code in a step-by-step manner, making it easy to understand and visualize each stage of data analysis.
- Code cells can be executed individually, allowing for an iterative and interactive coding experience.
- Execute code cells one by one to see immediate results, making it easier to debug and understand the impact of each code segment.

Jupyter Notebooks Features

Rich Media Integration:

- Supports the inclusion of images, charts, and interactive visualizations directly in the notebook, enhancing the storytelling aspect of data analysis.

Narrative Text Support:

- Enables the inclusion of descriptive text alongside code, facilitating the creation of a comprehensive and well-documented analysis.

Introduction to NumPy

- NumPy is a powerful library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices.

NumPy arrays: Efficient Handling of Numeric Data

- NumPy arrays are multi-dimensional, and resizable data structures.
- NumPy provides a powerful array object that allows efficient manipulation of large datasets, making numerical operations more straightforward and faster.
- NumPy includes a wide range of mathematical functions for array manipulation, linear algebra, and statistical operations, crucial for data manipulation and analysis.

Why Use NumPy Arrays?

- **Homogeneous Data Storage**:

NumPy arrays are homogeneous, meaning they contain elements of the same data type.

Arrays enforce homogeneity, ensuring consistent data types within the same array.

- **Vectorized Operations**:

Operations on NumPy arrays are vectorized, eliminating the need for explicit loops and making computations faster.

- **Broadcasting**:

Broadcasting allows for operations on arrays of different shapes and sizes, providing flexibility in array manipulation.

Example

Creating a NumPy array and performing a mathematical operation (squaring) on its elements.

```
import numpy as np

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

# Perform a mathematical operation
arr_squared = arr ** 2

print(arr_squared)
```

```
[ 1  4  9 16 25]
```

Lists VS NumPy arrays

- While lists in Python and arrays in NumPy may seem similar, there are key differences between the two. Here's an overview of the distinctions:

1. Homogeneity:

List (Python):

Lists can contain elements of different data types.

You can have a mix of integers, floats, strings, etc., in a single list.

Array (NumPy):

NumPy arrays are homogeneous, meaning they hold elements of the same data type.

This homogeneity allows for more efficient storage and operations.

Lists VS NumPy arrays

2. Vectorized Operations:

List (Python):

Operations on lists generally require explicit loops.

Performing mathematical operations on each element of a list typically involves iterating through the list using a loop.

Array (NumPy):

NumPy arrays support vectorized operations.

Operations can be applied to entire arrays without the need for explicit loops, leading to faster and more concise code.

Lists VS NumPy arrays

3. Memory Efficiency:

List (Python):

Lists are generally less memory-efficient.

Each element in a list carries additional information such as type and reference information.

Array (NumPy):

NumPy arrays are more memory-efficient.

They store elements in contiguous memory locations, reducing the overhead associated with each element.

Lists VS NumPy arrays

4. Functionality and Methods:

List (Python):

Lists offer a basic set of functionalities and methods for general-purpose use.

They lack specialized methods for numerical operations.

Array (NumPy):

NumPy arrays come with an extensive set of functions and methods optimized for numerical computations.

NumPy provides tools for linear algebra, statistical operations, and more.

Lists VS NumPy arrays

5. Broadcasting:

List (Python):

Lists do not support broadcasting.

Operations on lists typically require compatible shapes or explicit looping.

Array (NumPy):

NumPy arrays support broadcasting, enabling operations on arrays of different shapes and sizes.

Example

In the example, array B is broadcasted to the shape of A, allowing the addition operation to be performed element-wise.

The value B is effectively treated as if it were replicated along a new axis to match the shape of A.

```
import numpy as np

# Example of Broadcasting
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = np.array([10, 20, 30])

# Broadcasting automatically adjusts the shape of B to (2, 3) to match A
result = A + B

print(result)
```

```
[[11 22 33]
 [14 25 36]]
```

Introduction to Pandas

- **DataFrame Structure**: Pandas introduces two primary data structures: Series (1-dimensional) and DataFrames (2-dimensional), making it easy to work with structured data.

- **Data Cleaning and Transformation**: Pandas provides powerful tools for handling missing data, filtering, and transforming datasets, ensuring data is in a suitable format for analysis.

Pandas Series

➢ A One-Dimensional Labeled Array

- The Pandas Series is a one-dimensional labeled array that can hold any data type.
- It consists of data and associated labels (index).
- It's particularly useful when working with single-dimensional datasets or when extracting a single column from a DataFrame.

Why Use Series?

- **Labeled Indexing**:

  Series provides labeled indexing, making it easy to access and manipulate data using meaningful labels.

- **Homogeneous Data**:

  Ensures homogeneity within a column, simplifying operations on the entire set of data.

- **Integration with NumPy**:

  Series seamlessly integrates with NumPy, allowing for the application of various mathematical and statistical operations.

Example

Creating a Pandas Series to represent a one-dimensional array of ages.

```python
import pandas as pd

# Create a Pandas Series
ages = pd.Series([25, 30, 35], name='Age')

print(ages)
```

```
0    25
1    30
2    35
Name: Age, dtype: int64
```

DataFrame

➢ A Two-Dimensional Tabular Data Structure

- The DataFrame is a two-dimensional, tabular data structure similar to a spreadsheet or SQL table.
- It consists of rows and columns, where each column can have a different data type.
- The DataFrame is a tabular representation of data, where information is organized into rows and columns.
- Each column is essentially a Pandas Series.

Why Use DataFrames?

- **Flexibility and Versatility**:

DataFrames provide a flexible and versatile structure to handle diverse datasets with ease.

Columns can be of different data types, allowing for mixed data within the same structure.

- **Efficient Data Manipulation**:

Operations on columns, rows, and the entire DataFrame are vectorized, leading to efficient and fast data manipulations.

- **Integration with Other Tools**:

DataFrames seamlessly integrate with other libraries such as NumPy and Matplotlib, forming a comprehensive ecosystem for data analysis.

Example

Creating a Pandas DataFrame to organize and display structured data.

```python
import pandas as pd

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

df = pd.DataFrame(data)

print(df)
```

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

What is a CSV File?

- CSV stands for Comma-Separated Values.
- It is a plain text file where each line represents a row, and values are separated by commas (or other delimiters).
- CSV files are a common format for storing structured data.

CSV Structure:

- Rows represent records, and columns represent fields.
- The first row often contains headers, defining the names of the fields.

Read CSV file using *read_csv* function

- Before using this function, we must import the Pandas library.

- Like any other function, it can take parameters. Here is the Pandas read CSV syntax with its parameters.

pandas.read_csv(filepath_or_buffer, sep= ’ , ’ , header= ’infer’ , index_col= None, usecols= None, nrows= None)

Parameters: (there are more other parameters)

- filepath_or_buffer: Location of the csv file. It accepts any string path or URL of the file.
- sep: It stands for separator, default is ‘, ‘.
- header: It accepts int, a list of int, row numbers to use as the column names, and the start of the data. If no names are passed, i.e., header=None, then, it will display the first column as 0, the second as 1, and so on.
- usecols: Retrieves only selected columns from the CSV file.
- nrows: Number of rows to be displayed from the dataset.
- index_col: If None, there are no index numbers displayed along with records.

Example

Creating a Pandas DataFrame object from a CSV file

In [1]:
```python
# Import pandas library
import pandas as pd

# Creating a dataframe by reading a CSV file
df = pd.read_csv('./dataset.csv')
```

In [2]:
```python
df
```

Out[2]:

	Year	Age	Role	City	Education	Experience	Salary
0	2018	47	Helpdeskmanager	Apeldoorn	HBO	25	4950
1	2018	42	Applicatiebeheerder	Almelo	HAVO	10	4086
2	2018	42	BI Consultant	NaN	MBO	25	4400
3	2018	39	Software Engineer	Amsterdam	MBO	6	3600
4	2018	38	IT Consultant	Mierlo	HBO	15	4900
...	...	...	...	...	...	...	...
95	2018	24	Software Engineer	Zuid Holland	HBO	2	2413
96	2018	23	Servicedeskmedewerker	Randstad	MBO	3	2200
97	2018	23	Software Engineer	Friesland	MBO	5	3250
98	2018	23	Infrastructure Specialist	NaN	HBO	0	2200
99	2018	23	Web Developer	Nijmegen	MBO	2	1850

100 rows × 7 columns

Example

Methods and attributes of DataFrame class

```
In [3]:  df.shape

Out[3]: (100, 7)
```

```
In [4]:  # Display the first 5 rows of the dataset
         df.head()

Out[4]:
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	47	Helpdeskmanager	Apeldoorn	HBO	25	4950
1	2018	42	Applicatiebeheerder	Almelo	HAVO	10	4086
2	2018	42	BI Consultant	NaN	MBO	25	4400
3	2018	39	Software Engineer	Amsterdam	MBO	6	3600
4	2018	38	IT Consultant	Mierlo	HBO	15	4900

Integration of Jupyter, NumPy, and Pandas

- Loading data
- Performing basic operations
- Visualizing results

➢ The provided example demonstrates the integration by loading a dataset, performing basic analysis using NumPy and Pandas, and displaying the results within the Jupyter Notebook.

Example

Integrating Jupyter, NumPy, and Pandas to load a dataset, perform basic analysis, and display the results directly in the notebook.

In [5]:
```
# Import necessary libraries
import pandas as pd
import numpy as np

# Creating a dataframe by reading a CSV file
df = pd.read_csv('./dataset.csv')

# Perform basic analysis using Pandas and NumPy
max_value = df['Salary'].max()
mean_value = np.mean(df['Salary'])

# Display results in the Jupyter Notebook
mean_value, max_value
```

Out[5]:
```
(3510.75, 10000)
```

Exercise

- Install Anaconda on your computer. Open Anaconda navigator and run Jupyter Notebook.

- Jupyter:

- Create a new Jupyter Notebook.
- Create markdown cells and code cells
- Write and execute a code cell to print a message of your choice.

Exercise

- NumPy:

- Create a NumPy array with values from 1 to 10.
- Calculate the mean and standard deviation of the array.
- Feel free to experiment and test your understanding of the concepts we've covered.

Exercise

- Pandas:

- Create a Pandas DataFrame with at least three columns.
- Use DataFrame methods to display summary statistics and the first few rows.

Exercise

- Load given dataset into Pandas and conduct initial data exploration

Explore the data, check the first few rows, and try some basic operations to familiarize yourself with Pandas functionalities.

Here's a list of attributes and methods that you can try on your own created Pandas DataFrames.

These exercises will help you get hands-on experience with data manipulation and exploration using Pandas.

Exercise: DataFrame Attributes:

shape: Returns the dimensions of the DataFrame (number of rows, number of columns).

Example: df.shape

columns: Returns the column labels of the DataFrame.

Example: df.columns

index: Returns the row labels of the DataFrame.

Example: df.index

dtypes: Returns the data type of each column.

Example: df.dtypes

Exercise: DataFrame Methods:

head(n): Returns the first n rows of the DataFrame.

Example: df.head(3)

tail(n): Returns the last n rows of the DataFrame.

Example: df.tail(3)

info(): Provides a concise summary of the DataFrame, including data types and non-null values.

Example: df.info()

describe(): Generates descriptive statistics for numerical columns.

Example: df.describe()