CSE1006
Foundations of Data Analytics

Module - 6

Introduction to Pandas

- Introducing Pandas Objects
- Data Indexing and Selection
- Operating on Data in Pandas
- Handling missing data
- Hierarchical Indexing
- Vectorized String Operations
- Visualization with Matplotlib

Introduction to Pandas

- Pandas is a powerful and open-source Python library.
- The Pandas library is used for data manipulation and analysis.
- Pandas consist of data structures and functions to perform efficient operations on data.
- Pandas is well-suited for working with tabular data, such as spreadsheets or SQL tables.
- The Pandas library is an essential tool for data analysts, scientists, and engineers working with structured data in Python.
- It is built on top of the NumPy library which means that a lot of the structures of NumPy are used or replicated in Pandas.
- Python's Pandas library is the best tool to analyze, clean, and manipulate data.

List of things that we can do using Pandas:

- Data set cleaning, merging, and joining.
- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data.
- Columns can be inserted and deleted from DataFrame and higher-dimensional objects.
- Powerful group by functionality for performing split-apply-combine operations on data sets.
- Data Visualization.

Getting started with Pandas:

Step 1: Type ‘cmd’ in the search box and open it.

Step 2: Locate the folder using the cd command where the python-pip file has been installed.

Step 3: After locating it, type the command: **pip install pandas**

After the Pandas have been installed in the system, you need to import the library. **Import pandas as pd**

Data Structures in Pandas Library

Pandas generally provide two data structures for manipulating data.

Series

DataFrame

Panels

Pandas Series:

- A Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, Python objects, etc.). The axis labels are collectively called indexes.
- The Pandas Series is nothing but a column in an Excel sheet. Labels need not be unique but must be of a hashable type.

Series

- We use Serries() to create a series in Pandas
- Create a simple Pandas Series from a list:

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

```
0    1
1    7
2    2
dtype: int64
```

- Note : If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.
- This label can be used to access a specified value.

```
print(myvar[0])        #1
```

- With the index argument, you can name your own labels.

- ✓ Create a simple Pandas Series with index attribute

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```

- ✓ This label can be used to access a specified value.

```
print(myvar[“y”])  #7
```

```
x    1
y    7
z    2
dtype: int64
```

- ✓ You can also use a key/value object, like a dictionary, when creating a Series.

```
calories={"day1": 420,"day2":380,"day3":390}
myvar = pd.Series(calories)
print(myvar)
```

```
day1    420
day2    380
day3    390
dtype: int64
```

- ✓ Within pandas, a missing value is denoted by NaN
- ✓ We can create a Series with missing values

```
import numpy as np
import pandas as pd
print(pd.Series([1,np.nan,'Hello', None]))
```

```
0        1
1      NaN
2    Hello
3     None
dtype: object
```

- ✓ isnull() method is used to detect missing values for an array-like object.

```
import numpy as np
import pandas as pd
data = pd.Series([1, np.nan, 'hello', None])
print(data.isnull())
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

- ✓ One popular method during the data cleansing stage is the .notnull method.
- ✓ .notnull is a general function of the pandas library in Python that detects if values are not missing for either a single value (scalar) or array-like objects.
- ✓ The function returns booleans to reflect whether the values evaluated are null (False) or not null (True).

```
import numpy as np
import pandas as pd
data = pd.Series([1, np.nan, 'hello', None])
print(data.notnull())
print(data[data.notnull()])
```

```
0     True
1    False
2     True
3    False
dtype: bool
0        1
2    hello
dtype: object
```

- ✓ The fillna() method replaces the NULL values with a specified value.

```
import numpy as np
import pandas as pd
data = pd.Series([1, np.nan, 'hello', None])
print(data.fillna(0))
```

```
0        1
1        0
2    hello
3        0
dtype: object
```

- ✓ numpy.nanmean() function can be used to calculate the mean of array ignoring the NaN value.

```
import numpy as np
import pandas as pd
data = pd.Series([1, np.nan, 3, None])
print(data.fillna(np.nanmean(data)))
```

```
0    1.0
1    2.0
2    3.0
3    2.0
dtype: float64
```

Accessing Element from Series with Position:

- ✓ In order to access the series element refers to the index number. Use the index operator [ ] to access an element in a series.
- ✓ The index must be an integer.
- ✓ In order to access multiple elements from a series, we use Slice operation. Slice operation is performed on Series with the use of the colon(:).
- ✓ To print elements from beginning to a range use [:Index].
- ✓ To print elements from end-use [:-Index].
- ✓ To print elements from specific Index till the end use [Index:].
- ✓ To print elements within a range, use [Start Index:End Index] and to print whole Series with the use of slicing operation, use [:].
- ✓ Further, to print the whole Series in reverse order, use [::-1].

```
# creating simple array
data = np.array(['V', 'I', 'T', 'A', 'A', 'P','U', 'N', 'I', 'V', 'E', 'R', 'S'])
ser = pd.Series(data)
# retrieve the first element
print(ser[0])                    #V
# retrieve the first five elements
print(ser[:5])
# retrieve the last 10 elements
print(ser[-10:])
# Access all elements in reverse
print(ser[::-1])
```

```
0    V
1    I
2    T
3    A
4    A
dtype: object
3     A
4     A
5     P
6     U
7     N
8     I
9     V
10    E
11    R
12    S
dtype: object
```

```
12    S
11    R
10    E
9     V
8     I
7     N
6     U
5     P
4     A
3     A
2     T
1     I
0     V
dtype: object
```

- In this example, the Pandas module is imported, and a DataFrame ‘df’ is created by reading data from a CSV file named “nba.csv” using `pd.read_csv`.
- A Pandas Series ‘ser’ is then created by selecting the ‘Name’ column from the DataFrame. Finally, the first 10 elements of the series are accessed and displayed using `ser.head(10)`.

```
# importing pandas module
import pandas as pd
# making data frame
df = pd.read_csv("nba.csv")
ser = pd.Series(df['Name'])
ser.head(10)
```

```
0    Avery Bradley
1      Jae Crowder
2     John Holland
3      R.J. Hunter
4    Jonas Jerebko
5     Amir Johnson
6    Jordan Mickey
7     Kelly Olynyk
8     Terry Rozier
9     Marcus Smart
Name: Name, dtype: object
```

Pandas DataFrames

- A Pandas DataFrame is a 2-dimensional data structure, like a 2-dimensional array, or a table with rows and columns.

	calories	duration
0	420	50
1	380	40
2	390	45

Create a simple Pandas DataFrame:

```
import pandas as pd
data = {   "calories": [420, 380, 390],
           "duration": [50, 40, 45]
        }
#load data into a DataFrame object
df = pd.DataFrame(data)
print(df)
```

As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas DataFrames

Locate Row:

Pandas use the loc attribute to return one or more specified row(s).

```
#refer to the row index:
print(df.loc[0])
```

```
calories    420
duration     50
Name: 0, dtype: int64
```

```
#use a list of indexes:
print(df.loc[[0, 1]])
```

```
   calories  duration
0       420        50
1       380        40
```

With the index argument, you can name your own indexes.

```
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
```

```
#refer to the named index:
print(df.loc["day1"])
```

```
calories    420
duration     50
Name: 0, dtype: int64
```

iloc:

- In the Python Pandas library, .iloc[] is an indexer used for integer-location-based indexing of data in a DataFrame.
- It allows users to select specific rows and columns by providing integer indices, making it a valuable tool for data manipulation and extraction based on numerical positions within the DataFrame.

•row_selection:

- It is an optional parameter.
- This parameter specifies the rows to be selected based on their integer index.
- It can be a single integer, a list of integers, or a slice object.
- For example, to select the first three rows of a ***DataFrame***, you can use **df.iloc[0:3]**.

•column_selection:

- It is an optional parameter.
- This parameter specifies the columns selected based on their integer index.
- It can be a single integer, a list of integers, or a slice object.
- For example, to select the first two columns of a DataFrame, you can use **df.iloc[:, 0:2]**.

```python
import pandas as pd
data = pd.DataFrame({
    'Name': ['Geek1', 'Geek2', 'Geek3', 'Geek4', 'Geek5'],
    'Age': [25, 30, 22, 35, 28],
    'Salary': [50000, 60000, 45000, 70000, 55000]})
 # Setting 'Name' column as the index for clarity
data.set_index('Name', inplace=True)
print(data)
# Extracting a single row by index
row_alice = data.iloc[0, :]
print(row_alice)
# Extracting multiple rows using a slice
rows_geek2_to_geek3 = data.iloc[1:3, :]
print("\nExtracted Rows (Geek2 to Geek3):")
print(rows_geek2_to_geek3)
```

```
Original DataFrame:
       Age  Salary
Name
Geek1   25   50000
Geek2   30   60000
Geek3   22   45000
Geek4   35   70000
Geek5   28   55000

Extracted Row (Geek1):
Age          25
Salary    50000
Name: Geek1, dtype: int64

Extracted Rows (Geek2 to Geek3):
       Age  Salary
Name
Geek2   30   60000
Geek3   22   45000
```

Hierarchical Indexing

- Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index levels on an axis.
- It provides a way for you to work with higher dimensional data in a lower dimensional form.
- Let's start with a simple example; create a Series with a list of lists (or arrays) as the index:

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
```

Partial listing of vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element

Example 1:

Create a Pandas Series and DataFrame and perform the following tasks:

a) Create a Series named population containing city names as index and population values as data.

b) Display only cities with population greater than 10 lakhs.

c) Create a DataFrame named students with columns: Student_ID, Name, Age, and CGPA.

d) Set Student_ID as the index.

e) Retrieve only the Name and CGPA of students having CGPA >= 8.5.

```
import pandas as pd
a) Create a Series with city population
population = pd.Series(
    [1500000, 850000, 1200000, 950000],
    index=['Mumbai', 'Pune', 'Delhi', 'Hyd']
)

b) Display only cities with population > 10 lakhs
print(population[population > 1000000])

c) Create DataFrame of students
students = pd.DataFrame({
    'Student_ID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'Daisy'],
    'Age': [20, 21, 19, 22],
    'CGPA': [8.6, 7.5, 9.2, 8.9]
})

d) Set Student_ID as index
students.set_index('Student_ID', inplace=True)

e) Retrieve Name and CGPA of students with CGPA >= 8.5
print(students[students['CGPA'] >= 8.5][['Name', 'CGPA']])
```

Example 2:

Consider a DataFrame containing employee details. Perform the following operations:

a) Create a DataFrame with some missing values in the Salary and Department columns.

b) Identify rows with missing values using isnull().

c) Replace all missing salaries with the average salary.

d) Drop rows where Department is missing.

e) Fill missing ages with a constant value 30.

```
import numpy as np
a) DataFrame with missing values
df = pd.DataFrame({
    'Employee_ID': [101, 102, 103, 104],
    'Name': ['John', 'Alice', 'Raj', 'Sara'],
    'Age': [28, np.nan, 32, 40],
    'Salary': [50000, np.nan, 55000, 60000],
    'Department': ['HR', 'Finance', np.nan, 'IT']
})

b) Identify rows with missing values
print(df[df.isnull().any(axis=1)])

c) Replace missing Salary with mean
df['Salary'].fillna(df['Salary'].mean(), inplace=True) or df.fillna(np.nanmean(df.salary))

d) Drop rows where Department is missing
df.dropna(subset=['Department'], inplace=True)

 e) Fill missing ages with 30
df['Age'].fillna(30, inplace=True)

print(df)
```

Example 3:

Given a DataFrame orders with columns OrderID, Customer, Product, and Amount, perform the following tasks:

a) Select only the first 3 rows using .iloc.

b) Retrieve all orders where the amount is greater than ₹1000.

c) Set OrderID as the index and display the updated DataFrame.

d) Use .loc[] to retrieve details of a specific order.

e) Sort the DataFrame by Amount in descending order.

Example 4:

Create a DataFrame products with columns Product_Name, Category, and Sales. Perform the following:

a) Convert all Product_Name values to uppercase using vectorized string functions.

b) Extract all products where the name contains the word "Smart".

c) Count the number of products in each category.

d) Plot a bar chart showing total sales per category using Matplotlib.

e) Add axis labels and a title to the plot.

Visualization with Matplotlib

Data Visualization is the process of presenting data in the form of graphs or charts. It helps to understand large and complex amounts of data very easily.
It allows the decision-makers to make decisions very efficiently and also allows them in identifying new trends and patterns very easily.
It is also used in high-level data analysis for Machine Learning and Exploratory Data Analysis (EDA). Data visualization can be done with various tools like Tableau, Power BI, Python.

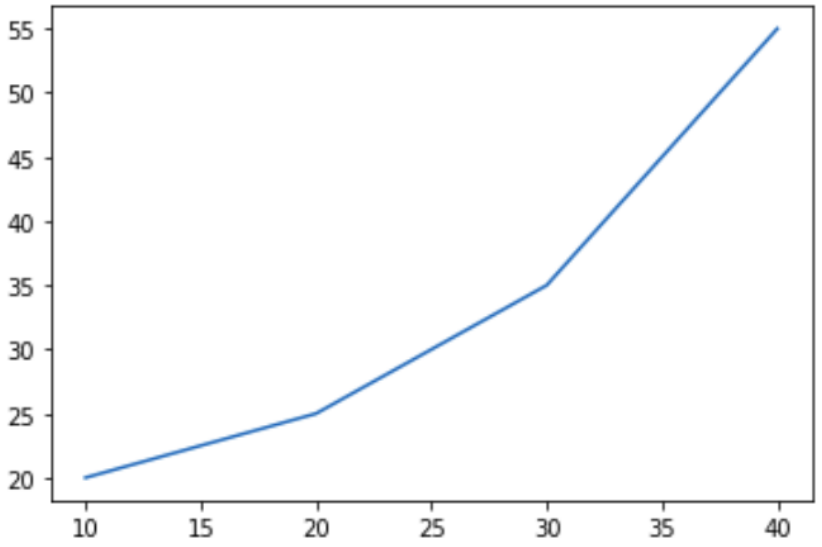
Matplotlib:

Matplotlib is a low-level library of Python which is used for data visualization.
It is easy to use and emulates MATLAB like graphs and visualization.
This library is built on the top of NumPy arrays and consist of several plots like line chart, bar chart, histogram, etc.
It provides a lot of flexibility but at the cost of writing more code.
To install Matplotlib type the below command in the terminal “pip install matplotlib”.

Pyplot:

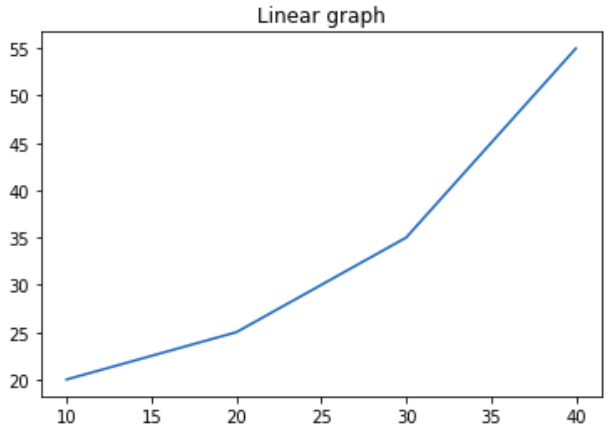
Pyplot is a Matplotlib module that provides a MATLAB-like interface. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. The various plots we can utilize using Pyplot are Line Plot, Histogram, Scatter, 3D Plot, Image, Contour, and Polar.

```
import matplotlib.pyplot as plt
# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]
# plotting the data
plt.plot(x, y)
plt.show()
```



The title() method in matplotlib module is used to specify the title of the visualization depicted and displays the title using various attributes.

```
import matplotlib.pyplot as plt
# initializing the data
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]
# ploting the data
plt.plot(x, y)
# Adding title to the plot
plt.title("Linear graph")
plt.show()
```



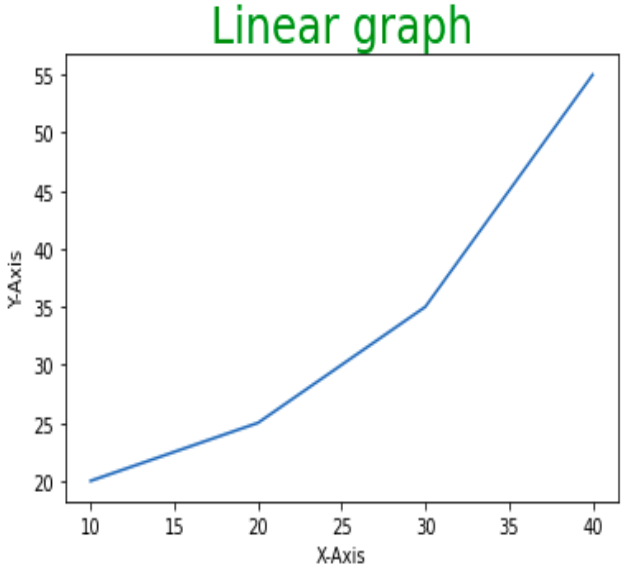
We can also change the appearance of the title by using the parameters of this function.

```
plt.title("Linear graph“,fontize=25,
color=“green”)
```

Adding X Label and Y Label:

the X label and the Y label are the titles given to X-axis and Y-axis respectively. These can be added to the graph by using the xlabel() and ylabel() methods.

```
import matplotlib.pyplot as plt
x = [10, 20, 30, 40]
y = [20, 25, 35, 55]
# plotting the data
plt.plot(x, y)
plt.title("Linear graph", fontsize=25, color="green")
# Adding label on the y-axis
plt.ylabel('Y-Axis')
# Adding label on the x-axis
plt.xlabel('X-Axis')
plt.show()
```



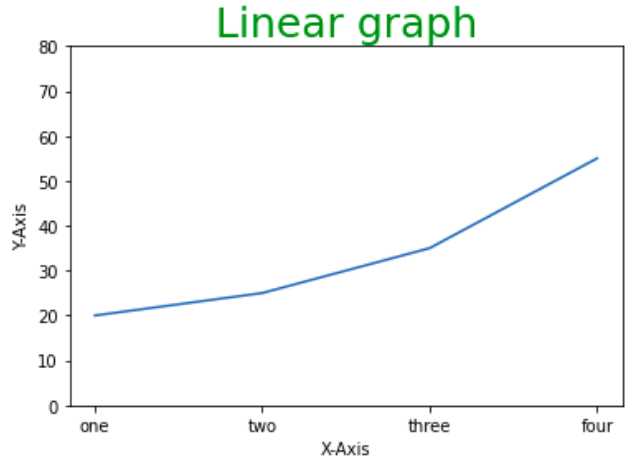
Setting Limits and Tick labels:

You might have seen that Matplotlib automatically sets the values and the markers(points) of the X and Y axis, however, it is possible to set the limit and markers manually.

xlim() and ylim() functions are used to set the limits of the X-axis and Y-axis respectively. Similarly, xticks() and yticks() functions are used to set tick labels.

```
# Setting the limit of y-axis
plt.ylim(0, 80)
# setting the labels of x-axis
plt.xticks(x, labels=["one", "two", "three", "four"])

plt.show()
```



plt.plot() – Draws line plots.

plt.scatter() – Creates scatter plots.

plt.bar() / plt.barh() – Draws vertical/horizontal bar charts.

plt.hist() – Plots histograms.

plt.pie() – Creates pie charts.

plt.figure() – Creates a new figure.

plt.subplot() / plt.subplots() – Adds subplots (multiple plots in one figure).

fig.add_subplot() – Adds a subplot to a figure using object-oriented interface.

plt.gca() – Gets current axes.

plt.gcf() – Gets current figure.

plt.title() – Adds a title to the plot.

plt.xlabel() / plt.ylabel() – Labels the X and Y axes.

plt.xticks() / plt.yticks() – Sets tick locations and labels.

plt.legend() – Displays a legend.

plt.grid() – Adds gridlines.

plt.style.use() – Applies a predefined style (like 'ggplot', 'seaborn', etc.).

plt.xlim() / plt.ylim() – Sets axis limits.

plt.axhline() / plt.axvline() – Draws horizontal/vertical lines.

plt.axhspan() / plt.axvspan() – Highlights horizontal/vertical spans.

plt.show() – Displays the plot.

plt.savefig() – Saves the plot as an image file (e.g., .png, .jpg, .pdf).

Example 5:

Write a Python program using Pandas and Matplotlib to perform the following tasks:

a) Create a Pandas DataFrame with Studentname, Math, English, Science with minimum 6 records.
b) Plot a line chart comparing student scores in Math, Science, and English.
c) Plot a bar chart for Math scores of all students.
d) Plot a pie chart showing the percentage share of Science scores.
e) Customize all plots with appropriate title, axis labels, and grid.