Experiment No. 3

Title: Exploratory data analysis using PANDAS

Batch: A3 **Roll No:** 16010423099 **Experiment No.:3**

Aim: To perform exploratory data analysis using python Pandas

Resources needed: Python IDE

Theory:

Pandas is a Python library that provides extensive means for data analysis. Data scientists often work with data stored in table formats like .csv, .tsv, or .xlsx. Pandas makes it very convenient to load, process, and analyze such tabular data using SQL-like queries. Python has long been great for data munging and preparation, but less so for data analysis and modeling. *pandas* helps fill this gap, enabling you to carry out your entire data analysis workflow in Python. In conjunction with Matplotlib and Seaborn, Pandas provides a wide range of opportunities for visual analysis of tabular data.

Installing pandas library:

Conda install pandas

Or

pip install pandas

The main data structures in Pandas are implemented with Series and DataFrame classes. The former is a one-dimensional indexed array of some fixed data type. The latter is a two-dimensional data structure - a table - where each column contains data of the same type. You can see it as a dictionary of Series instances. DataFrames are great for representing real data: rows correspond to instances (examples, observations, etc.), and columns correspond to features of these instances.

A series can be created using list ,dictionary etc. with index(implicit indexing) or without index(explicit indexing).

```
import pandas as pd
data1=pd.Series({2:'a', 1:'b', 3:'c'}) #implicit indexing
data2=pd.Series({2:'a', 1:'b', 3:'c'}, index=[1,2,3]) # explicit indexing
        #loc attribute allows indexing and slicing that always references the explicit index:
 data2.loc[2]
        #iloc attribute allows indexing and slicing that always references the implicit
```

```
    #Python-style index
data.iloc[1]
```

Following are the various series related operations

- Append(): s3.append(s1) # Stitch s1 to s3
- Drop: s4.drop('e') #Delete the value whose index is e
- Addition: s4.add(s3)#addition according to the index, and it would be filled with NaN (null value) if the indexes are different.
- Subtraction: s4.sub(s3) #substraction according to the index, and it would be filled with NaN (null value) if the indexes are different.
- Multiplication: s4.mul(s3) #multiplication according to the index, and it would be filled with NaN (null value) if the indexes are different.
- Division: s4.div(s3)
- Median: s4.median()
- Sum: s4.sum()
- Maximum & Minimum : s4.max() s4.min()

A data frame object keeps track of both data (numerical as well as text), and column and row headers. It can have multiple columns of data.
convert a numpy array to a pandas data frame with pd.Data frame().

```
import numpy as np
h = [[1,2],[3,4]]
df_h = pd.DataFrame(h)
print('Data Frame:', df_h)
```

data frame can read data from dictionary and files as well.

Reading and writing data from files:
CSVs don't have indexes like our DataFrames, so all we need to do is just designate the index_col when reading.

```
import pandas as pd
df=pd.read_csv("C:/Users/Admin/Desktop/ADVANCED
PYTHON/DATA/SalesJan2009.csv",index_col=0)
#Reading the dataset in a dataframe using Pandas
print(df)
```

To write data to a new csv file use to_csv()

```
df3.to_csv('animal.csv')
df3.to_excel('animal.xlsx', sheet_name='Sheet1')
```

Following functions of dataframe can be used to explore dataset to get summary of it.

- **info()** provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.
  df.info()
- **describe()** is used to get a summary of numeric values in your dataset. It calculates the mean, standard deviation, minimum value, maximum value, 1st percentile, 2nd percentile, 3rd percentile of the columns with numeric values. It also counts the number of variables in the dataset.
  df.describe()
  describe() can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category
  temp_df['product'].describe()
- **head()** outputs the first five rows of your DataFrame by default, but we could also pass a number as well
  print(df.head)
- **tail()** outputs last five rows by default.
  print(df.tail)
- **shape()** outputs just a tuple of (rows, columns):
  df.shape

Row and column selection:
Each row and column of dataframe is a series . following functions can be used for row selection and column selection

Extracting a column using square brackets will return a *Series.*

```
prod_col=temp_df['product']
#selecting multiple columns
subset=temp_df[['product','price']]
```

accessing rows:

.loc - locates by name

```
prom = movies_df.loc["Prometheus"]
prom
```

.iloc- locates by numerical index

```
prod = df.iloc[1]
prod
```

Further analysis using pandas dataframe:

value_counts() can tell us the frequency of all values in a column.

```
temp_df['product'].value_counts().head(10)
```

nunique() to count number of unique values that occur in dataset or in a column

```
df.nunique() #to see the counts of unique numbers in each column
```

```
df["Embarked"].nunique() #to get the unique count of a column
```

corr() generate the relationship between each continuous variable:

```
temp_df.corr()
```

Correlation tables are a numerical representation of the bivariate relationships in the dataset.

astype() can be used to change the datatype of that column

```
df["Embarked"] = df["Embarked"].astype("category")
df["Embarked"].dtype
```

column clean up funtions:

append() will return a copy after appending without affecting the original DataFrame(if inplace attribute is used).

```
temp_df = df.append(df)
temp_df.shape
```

drop_duplicates() method will return a copy of DataFramewith duplicates removed.

```
temp_df = temp_df.drop_duplicates()
#inorder to avoid reasignment it can be done
temp_df.drop_duplicates(inplace=True)
temp_df.shape
```

.columns print the column names of dataset also can be used for renaming

```
temp_df.columns
```

drop() can be used to delete columns

```
df.drop(columns=['A', 'C'])
```

rename() method is used to rename certain or all columns via a dict.

```
temp_df.rename(columns={
'Account_Created': 'Acc_Created',
'Last_Login': 'Lst_Login'
  }, inplace=True)
temp_df.columns
```

Handling null values using pandas:

Mostly Python's None or NumPy's np.nan indicates missing or null values.

- **isnull()** checks which cells in our DataFrame are null. It returns a DataFrame where each cell is either True or False depending on that cell's null status.

```
temp_df.isnull()
```

To count the number of nulls in each column we use an aggregate function .sum() for summing:

```
temp_df.isnull().sum()
```

To get rid of rows or columns with nulls. Removing null data is only suggested if you have a small amount of missing data. .dropna() will delete any row with at least a single null value, but it will return a new DataFrame without altering the original one.

```
temp_df.dropna()
```

- Or drop columns with null values by setting axis=1.

```
temp_df.dropna(axis=1)
```

- Replace nulls with non-null values, a technique known as imputation. Normally null value is replaced with mean or the median of that column.

Conditional selections/ Filtering

Comparison operators are used for filtering

Take a column from the DataFrame and apply a Boolean condition to it.

```
condition = (movies_df['Director'] == "Ridley Scott")
```

It returns a Series of True and False values.Some more examples on conditionals

Select movies_df where movies_df Director equals Ridley Scott.

```
movies_df[movies_df['Director'] == "Ridley Scott"]
movies_df[movies_df['Rating'] >= 8.6].head(3)
movies_df[(movies_df['Director'] == 'Christopher Nolan') | (movies_df['Director'] ==
'Ridley Scott')].head()
movies_df[movies_df['Director'].isin(['Christopher Nolan', 'Ridley Scott'])].head()
```

Summary statistics Functions/ Aggregate Functions

Aggregating functions are the ones that reduce the dimension of the returned objects. DataFrames include some aggregate functions to understand the overall properties of a dataset

df.count(): Count the number of rows /items

df.mean(): To find mean average of data frame

Synatx: data.Population.mean()#where Population is column name

df.median(): To find median of data frame

df.quantile():

df.sum(): Do a summation operation on any column in the DataFrame

df.prod(): To find Product of all items

df.std():To find standard deviation of a data frame

df.var():To find varianceof data frame

df.min(), df.max(): To find Minimum and maximum

df.first(), df.last(): First and last item

GROUP BY and aggregation

"group by" process can involve one or more of the following steps:

–Splitting the data into groups based on some criteria.

–Applying a function to each group independently.

–Combining the results into a data structure

Apply step involves following

–Aggregation: compute a summary statistic (or statistics) for each group. Some examples:

- •Compute group sums or means.
- •Compute group sizes / counts.

–Transformation: perform some group-specific computations and return a like-indexed object. Some examples:

- •Standardize data (zscore) within a group.
- •Filling NAs within groups with a value derived from each group.

–Filtration: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:

- •Discard data that belongs to groups with only a few members.
- •Filter out data based on the group sum or mean.

```
# group the data on team value.
    gk = df.groupby('Team')
# Finding the values contained in the "Boston Celtics" group
     gk.get_group('Boston Celtics')
```

Applying functions

To iterate over a DataFrame or Series we can use list, but doing so — especially on large datasets — is very slow.

An efficient alternative is to apply() a function to the dataset. Using apply() will be much faster than iterating manually over rows because pandas is utilizing vectorization.

Combining datasets:

Combining Datasets

Concat: s to append either columns or rows from one DataFrame to another.

Joining two dataframe on the index

merge two dataframes on key attribute

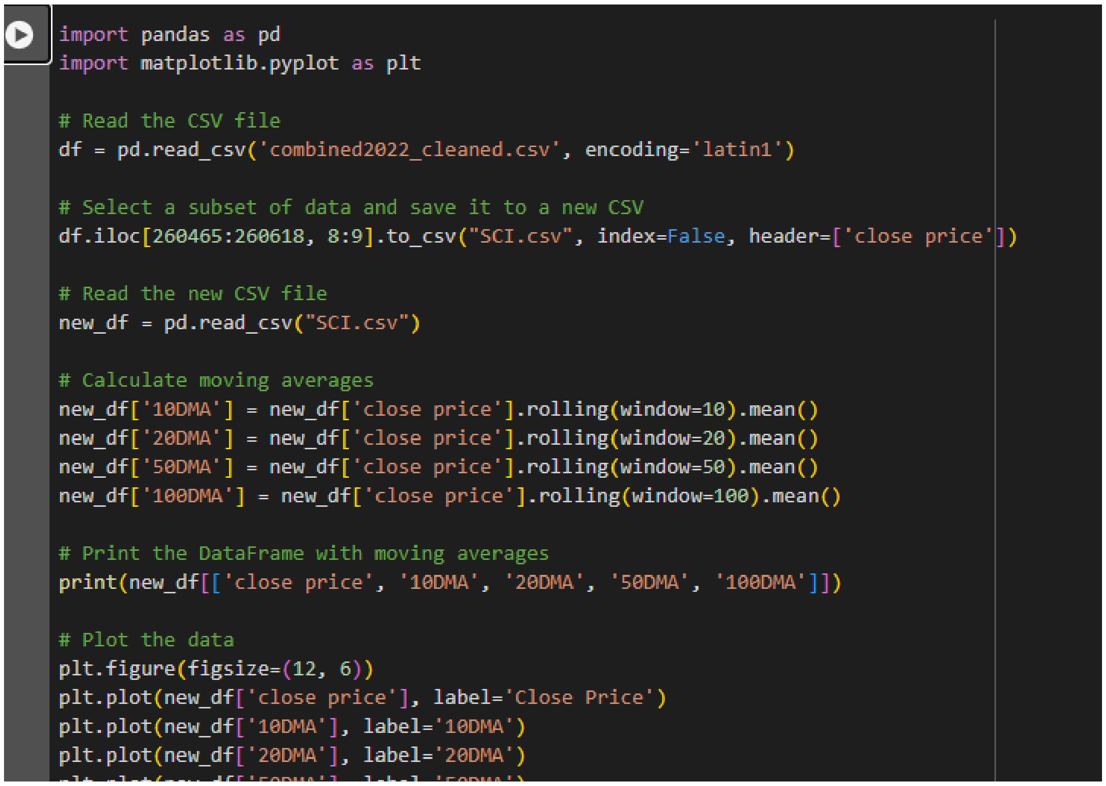
Activities:

1. Download data set with atleast 1500 rows and 10-20 columns(numeric and non numeric) from valid data sources
2. Read same in pandas DataFrame
3. Perform in detail Exploratory data analysis of this dataset
   - Get information and description of dataset.
   - See if any null values are present. Display count of null values.

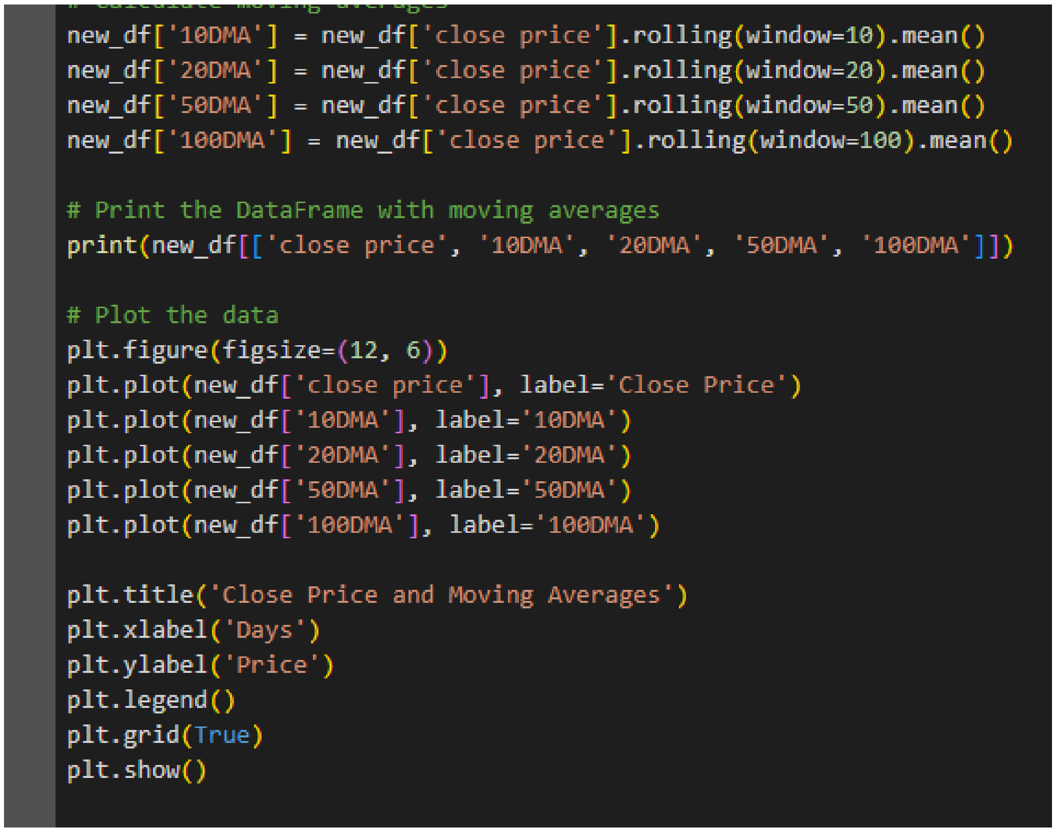
- Choose appropriate technique to handle missing values.(imputation with use of inplace)
- Use sorting of data in dataframe to display topmost 5 or 8 records based on one or more column values(conditional filtering)
- Get frequency listing of any one relavant column(2 cases)
- Sorting of rows and columns,(implicit and explicit indexing)
- Accessing particular row based on certain condition and displaying only one or few columns from it.(3 cases with compound conditions)
- Minimum and maximum values related analysis
- Use of group by on one or more columns(2 cases)
- Add new column to existing dataframe and populate same using existing columns data.
- Use of appropriate aggregate functions with groupby.(2 cases)
- Selection on particular groups based on name or condition
- Find correlation between any two columns values.
- Try transformation(normalization using any technique) on data set
- Joining , merging and concatenation of data in dataframe.

Write down observation for your dataset for each of above listed task of analysis.

Result: (script and output)

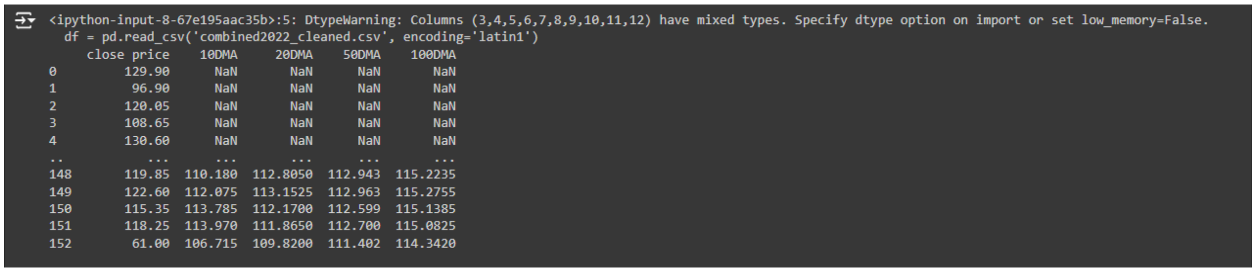


```python
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file
df = pd.read_csv('combined2022_cleaned.csv', encoding='latin1')

# Select a subset of data and save it to a new CSV
df.iloc[260465:260618, 8:9].to_csv("SCI.csv", index=False, header=['close price'])

# Read the new CSV file
new_df = pd.read_csv("SCI.csv")

# Calculate moving averages
new_df['10DMA'] = new_df['close price'].rolling(window=10).mean()
new_df['20DMA'] = new_df['close price'].rolling(window=20).mean()
new_df['50DMA'] = new_df['close price'].rolling(window=50).mean()
new_df['100DMA'] = new_df['close price'].rolling(window=100).mean()

# Print the DataFrame with moving averages
print(new_df[['close price', '10DMA', '20DMA', '50DMA', '100DMA']])

# Plot the data
plt.figure(figsize=(12, 6))
plt.plot(new_df['close price'], label='Close Price')
plt.plot(new_df['10DMA'], label='10DMA')
plt.plot(new_df['20DMA'], label='20DMA')
```

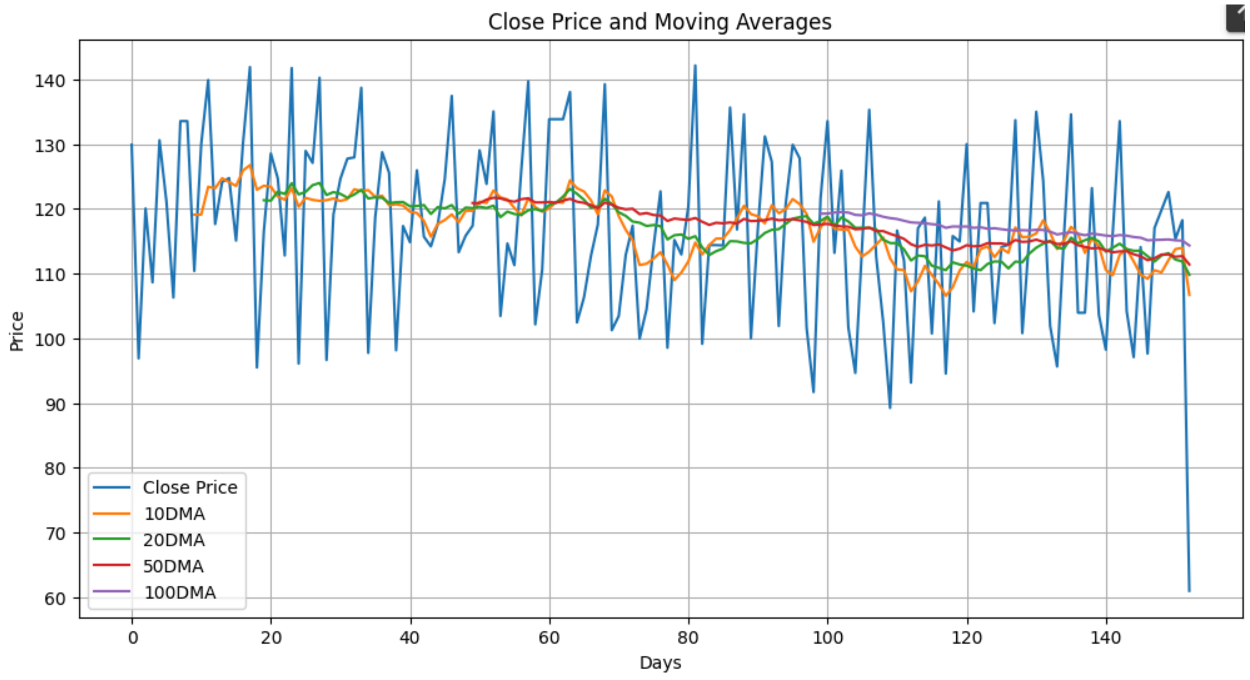


```python
new_df['10DMA'] = new_df['close price'].rolling(window=10).mean()
new_df['20DMA'] = new_df['close price'].rolling(window=20).mean()
new_df['50DMA'] = new_df['close price'].rolling(window=50).mean()
new_df['100DMA'] = new_df['close price'].rolling(window=100).mean()

# Print the DataFrame with moving averages
print(new_df[['close price', '10DMA', '20DMA', '50DMA', '100DMA']])

# Plot the data
plt.figure(figsize=(12, 6))
plt.plot(new_df['close price'], label='Close Price')
plt.plot(new_df['10DMA'], label='10DMA')
plt.plot(new_df['20DMA'], label='20DMA')
```

```python
plt.plot(new_df['50DMA'], label='50DMA')
plt.plot(new_df['100DMA'], label='100DMA')

plt.title('Close Price and Moving Averages')
plt.xlabel('Days')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()
```



```
<ipython-input-8-67e195aac35b>:5: DtypeWarning: Columns (3,4,5,6,7,8,9,10,11,12) have mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('combined2022_cleaned.csv', encoding='latin1')
     close price    10DMA     20DMA    50DMA    100DMA
0         129.90      NaN       NaN      NaN       NaN
1          96.90      NaN       NaN      NaN       NaN
2         120.05      NaN       NaN      NaN       NaN
3         108.65      NaN       NaN      NaN       NaN
4         130.60      NaN       NaN      NaN       NaN
..           ...      ...       ...      ...       ...
148       119.85  110.180  112.8050  112.943  115.2235
149       122.60  112.075  113.1525  112.963  115.2755
150       115.35  113.785  112.1700  112.599  115.1385
151       118.25  113.970  111.8650  112.700  115.0825
152        61.00  106.715  109.8200  111.402  114.3420
```



Outcomes:

CO3: Inculcate the knowledge of python libraries like NumPy, pandas, Matplotlib for scientific- computing and data visualization.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

References:

1. 1. Daniel Arbuckle, Learning Python Testing, Packt Publishing, 1st Edition, 2014
2. Wesly J Chun, Core Python Applications Programming, O’Reilly, 3rd Edition, 2015
3. Wes McKinney, Python for Data Analysis, O’Reilly, 1st Edition, 2017
4. Albert Lukaszewsk, MySQL for Python, Packt Publishing, 1st Edition, 2010
5. Eric Chou, Mastering Python Networking, Packt Publishing, 2nd Edition, 2017