Course End Assessment

Python For Data Science

Section- I (MCQ) [Solve any 6 Questions]

1. Ava is working on a data analysis project and needs to perform mathematical operations on large arrays efficiently. Which Python library should she use to handle these tasks?

c. Numpy

2. Ela is working on a program that involves counting the number of elements in various data structures. Which built-in Python function should she use to determine the number of elements in a list, tuple, or string?

d. len()

3. Misha is working on a project that involves creating visualizations to better understand her data. She is considering using Python libraries for data visualization. Which of the following libraries is commonly used in Python for creating interactive and static visualizations?

c. Matplotlib

5. In Matplotlib, what function is commonly used to create a scatter plot, where data points are represented as individual dots on a two-dimensional plane?

a. plt.scatter()

6. Mudit is developing a Python application that needs to interact with a MySQL database. He wants to connect to the database and execute queries using Python. Which of the following Python libraries should Mudit use to interact with the MySQL database?

a. mysql-connector

8. Sonia is working on a dataset in Pandas and wants to remove a column named "Age" from the DataFrame. Which Pandas operation should she use to delete this column?

c. df.drop("Age", axis=1)

Section - II (Short Answer Questions) [Solve any 8 questions]

1. Explain the difference between Python lists and NumPy arrays for scientific computing applications.

Python Lists	NumPy Arrays
Can store elements of different data types.	Stores elements of the same data type.
Slower for large-scale computations.	Faster due to optimized C-based implementation.
No built-in support for element-wise operations.	Supports element-wise operations and broadcasting.
Less memory efficient due to heterogeneous elements.	More memory efficient due to homogeneity.
General-purpose data storage.	Ideal for scientific computing and numerical tasks.

2. Explain the concept of indexing and slicing in one-dimensional NumPy arrays, providing examples.

Indexing: Accessing a single element using its position (index). Indexing starts from 0.

```
import numpy as np

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

print(arr[2])  # Output: 30 (3rd element)
```

Slicing: Extracting a subset of elements using a range of indices. The syntax is start:stop:step.

- start: Starting index (inclusive).
- stop: Ending index (exclusive).
- step: Increment between indices (optional).

```
print(arr[1:4])     # Output: [20 30 40] (elements from index 1 to 3)
print(arr[::2])     # Output: [10 30 50] (every 2nd element)
print(arr[-3:])     # Output: [30 40 50] (last 3 elements)
```

3. Explain the difference between a Pandas Series and a NumPy array in the context of data analysis.

Pandas Series	**NumPy Array**
One-dimensional, with labeled indices.	One-dimensional or multi-dimensional, no labels.
Supports custom indexing for data alignment.	Uses positional indexing only.
Can hold mixed data types, similar to Python lists.	Requires all elements to be of the same type.
Built-in methods for data wrangling and analysis.	Primarily focuses on numerical computations.
Ideal for labeled data and data frames in analysis.	Best for fast numerical operations and computations.

4. Explain the difference between accessing elements by label and position in a Pandas Series.

Accessing by Label	**Accessing by Position**
Accessing elements using the Series' index labels.	Accessing elements using integer-based positions.
Uses series[label] or .loc[label].	Uses .iloc[position].
Depends on custom or default index labels (e.g., strings, integers).	Always uses integer-based indexing starting from 0.
Raises a Key Error if the label does not exist.	Raises an `Index Error` if the position is out of range.

5. Describe the purpose of the read_csv() function in pandas. Provide an example of using this function to read a CSV file into a DataFrame.

Purpose of read_csv() in Pandas:
The read_csv() function in Pandas is used to read data from a CSV (Comma-Separated Values) file and load it into a Pandas **DataFrame**. This is commonly used for data analysis, as it enables easy handling of structured datasets.

Key Features:

- Handles large datasets efficiently.
- Supports customization options like setting column names, skipping rows, handling missing values, etc.

Example:

```
import pandas as pd

# Reading a CSV file

df = pd.read_csv('filename.csv')

# Display the first few rows of the DataFrame

print(df.head())
```

7. Explain the difference between inner, outer, left, and right joins/merges, and provide examples of when each might be used.

Type of Join	Description	**Use Case**	Example
Inner Join	Returns rows that have matching keys in **both DataFrames**.	Use when you only need data that exists in both tables.	Merging sales and customers tables to get records where customers made purchases.
Outer Join	Returns **all rows** from both DataFrames, filling unmatched entries with NaN.	Use when you need a complete dataset with unmatched entries included.	Combining product lists from two vendors to create a master catalog, including items unique to each vendor.
Left Join	Returns **all rows from the left DataFrame** and matched rows from the right DataFrame (unmatched rows get NaN).	Use when you need all data from the left table and optional matches from the right table.	Adding optional sales data to a master list of customers, even if some customers haven't made purchases.
Right Join	Returns **all rows from the right DataFrame** and matched rows from the left DataFrame (unmatched rows get NaN).	Use when you need all data from the right table and optional matches from the left table.	Keeping a complete inventory list while merging supplier details for matched items.

Examples:

```
import pandas as pd

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

# Inner Join
inner = pd.merge(df1, df2, on='ID', how='inner')
print("Inner Join:\n", inner)
# Output: Rows with ID 2 and 3 (common in both DataFrames)

# Outer Join
outer = pd.merge(df1, df2, on='ID', how='outer')
print("Outer Join:\n", outer)
# Output: All rows from both DataFrames, unmatched rows filled with NaN.

# Left Join
left = pd.merge(df1, df2, on='ID', how='left')
print("Left Join:\n", left)
# Output: All rows from df1, matched data from df2 (NaN for unmatched).

# Right Join
right = pd.merge(df1, df2, on='ID', how='right')
print("Right Join:\n", right)
# Output: All rows from df2, matched data from df1 (NaN for unmatched).
```

9. Describe the most appropriate use cases for each of the following plot types:
• Line graph • Pie chart • Vertical bar chart • Horizontal bar chart

Plot Type	**Use Case**	**Explanation**
Line Graph	Used for visualizing trends over time (time series data).	Ideal for showing continuous data and trends over a period. E.g., stock prices, temperature changes.
Pie Chart	Used to show proportions or percentages of a whole.	Best for displaying parts of a whole, e.g., market share, budget distribution.
Vertical Bar Chart	Used for comparing discrete categories or groups.	Suitable for visualizing categorical data with distinct groups (e.g., sales by region, survey results).
Horizontal Bar Chart	Used when category names are long or when comparing categories that have longer labels.	Useful when the labels are too long for vertical bars or for comparing many categories.

10. What is the difference fenchone() and fetchall() methods?

Method	Description	Use Case
fetchone()	Retrieves **one row** from the result set. If no more rows are available, it returns None.	Use when you want to retrieve a single record from a database (e.g., retrieving a specific user).
fetchall()	Retrieves **all rows** from the result set as a list of tuples. Returns an empty list if no rows are found.	Use when you need to retrieve all results from a query, especially when working with multiple records.

Section - III (Long Answer Questions) [Solve any 5 questions]

1. Write the steps to connect Mysql database to the python application.

MySQL Database Connection in Python (contd.)

1. Install the MySQL Connector module: Use the pip command to install MySQL Connector Python.

2. Import the MySQL Connector module: Use the methods of the MySQL Connector module to communicate with the MySQL database.

3. Use the connect() method: Use the connect() method of the MySQL Connector class with the required arguments to connect MySQL. It will return the MySQLConnection object if the connection is established successfully.

4. Use the cursor() method: Use this method to create a cursor object to perform various SQL operations.

5. Use the execute() method: Use this method to run SQL queries.

6. Extract a result using the fetchall(), fetchone(), or fetchmany() method: Use these methods to read query results.

7. Close cursor and connection object: Use close() to close the database connection after the work is completed and you no longer want to interact with the database server.

2. what is Kurtosis and Types of Kurtosis?

Kurtosis is a statistical measure that describes the **shape of the distribution** of data, specifically the **tail behavior**. It provides insights into the "tailedness" or the presence of extreme values (outliers) in the distribution. In simple terms, kurtosis quantifies how much data points cluster in the tails (far ends) of the distribution compared to a normal distribution.

1. **High kurtosis** indicates that the distribution has **heavy tails** or outliers.
2. **Low kurtosis** suggests that the distribution has **light tails** or fewer outliers.
3. **Normal kurtosis** (a kurtosis of 3) corresponds to a normal distribution.

Types of Kurtosis:

1. **Leptokurtic (Positive Kurtosis)**:
    - **Definition**: A distribution that has **heavy tails** and a **sharp peak** near the mean.
    - **Kurtosis Value**: Greater than 3 (excess kurtosis greater than 0).
    - **Characteristics**:
        - More frequent extreme values or outliers.
        - More concentration around the mean, resulting in a "tall" peak.
    - **Example**: Financial data often shows leptokurtic distributions due to occasional large fluctuations (e.g., stock returns).

Visual Representation: The curve is sharper and more peaked than a normal distribution.

2. **Platykurtic (Negative Kurtosis)**:
    - **Definition**: A distribution that has **light tails** and a **flatter peak** compared to a normal distribution.

- **Kurtosis Value**: Less than 3 (excess kurtosis less than 0).
- **Characteristics**:
  - Fewer extreme values or outliers.
  - The distribution is more spread out with a flatter peak.
- **Example**: Uniform distributions (where all values are equally likely) exhibit platykurtic behavior.

Visual Representation: The curve is flatter than a normal distribution with lighter tails.

3. **Mesokurtic (Normal Kurtosis)**:
   - **Definition**: A distribution that has a **normal level of kurtosis** similar to the normal distribution.
   - **Kurtosis Value**: Equal to 3 (excess kurtosis equal to 0).
   - **Characteristics**:
     - The distribution follows a standard bell curve.
     - The tails are moderate, not too heavy or light.
   - **Example**: The **normal distribution** is mesokurtic by definition.

Visual Representation: The curve has a normal bell-shaped form with moderate tails.

3. List at least five different ways to customize a plot's appearance in Matplotlib (e.g., titles, labels, colors, styles).

1. Adding Titles and Labels:

- **Title**: Use plt.title() to set the plot's title.
- **X-axis label**: Use plt.xlabel() to label the x-axis.
- **Y-axis label**: Use plt.ylabel() to label the y-axis.

```
plt.plot(x, y)

plt.title("My Plot Title")

plt.xlabel("X-axis Label")
```

plt.ylabel("Y-axis Label")

2. Changing Line Styles and Colors:

You can customize the line style, color, and marker types using various arguments in the plot() function.

- **Line style**: '-' for solid, '--' for dashed, ':' for dotted, etc.
- **Line color**: You can use color names ('red', 'blue', etc.) or hexadecimal color codes ('#FF5733').
- **Marker style**: Use o, x, ^, etc. to specify marker types.

plt.plot(x, y, linestyle='--', color='green', marker='o')

3. Changing Axis Limits:

You can adjust the limits of the x and y axes to zoom in or out of your plot using plt.xlim() and plt.ylim().

plt.plot(x, y)

plt.xlim(0, 10) # Set x-axis range

plt.ylim(0, 20) # Set y-axis range

4. Customizing Gridlines:

You can add gridlines to your plot using plt.grid(). You can also adjust the grid style, color, and line width.

plt.plot(x, y)

plt.grid(True, linestyle='-', color='gray', linewidth=0.5)

5. Adding Legends:

Use plt.legend() to display a legend, which is helpful for distinguishing different data series. You can specify labels for each plot line.

plt.plot(x, y, label="Data Series 1")

plt.plot(x, z, label="Data Series 2")

```
plt.legend(loc='upper left')  # Display legend in the upper left corner
```

4. Describe three different ways to check for missing values in a DataFrame.

1. Using isnull() or isna() Method

The isnull() (or isna()) method returns a DataFrame of the same shape as the original, but with Boolean values indicating whether each value is missing (NaN). **True** indicates a missing value, and **False** indicates a non-missing value.

- **Syntax**: df.isnull() or df.isna()

```
import pandas as pd

# Sample DataFrame

data = {'A': [1, 2, None, 4], 'B': [None, 2, 3, 4]}

df = pd.DataFrame(data)

# Check for missing values

print(df.isnull())  # or df.isna()
```

2. Using sum() to Count Missing Values

After using the isnull() or isna() method, you can chain sum() to count the number of missing values in each column. This will give you a quick summary of missing data in each column.

- **Syntax**: df.isnull().sum() or df.isna().sum()

```
# Count the number of missing values per column

print(df.isnull().sum())
```

3. Using info() Method

The info() method provides a concise summary of the DataFrame, including the number of non-null entries in each column. The difference between the total

number of rows and the number of non-null entries gives the count of missing values.

- **Syntax**: df.info()

Display summary of DataFrame

df.info()

6. Explain the steps involved in converting a Pandas Series to a DataFrame and accessing specific data elements from the resulting DataFrame.

Step 1: Import Pandas

First, ensure that you have the necessary library, **Pandas**, imported into your environment.

import pandas as pd

Step 2: Create a Pandas Series

Create a Pandas **Series** from a list or any other iterable. This series will be converted to a DataFrame.

Create a Pandas Series

data = [10, 20, 30, 40, 50]

series = pd.Series(data, name='Numbers')

0 10

1 20

2 30

3 40

4 50

Name: Numbers, dtype: int64

Step 3: Convert the Pandas Series to a DataFrame

Use the pd.DataFrame() function to convert the **Series** into a **DataFrame**. If you want to assign the Series values as a column in the DataFrame, you can provide a column name.

```
# Convert Series to DataFrame
df = pd.DataFrame(series)
```

```
   Numbers
0      10
1      20
2      30
3      40
4      50
```

Step 4: Access Specific Data Elements from the DataFrame

Once the Series has been converted to a DataFrame, you can access specific data elements using various methods:

1. **Access a Column by Name**:
    - Use the column name to access data in the form of a **Series**.

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

      ```
      0    10
      1    20
      2    30
      3    40
      4    50
      Name: Numbers, dtype: int64
      ```

2. **Access a Specific Row by Index**:
    - Use the .iloc[] or .loc[] method to access a specific row.
    - **.iloc[]** accesses rows by integer index.
    - **.loc[]** accesses rows by label index.

Access the first row using iloc (integer-location based indexing)

print(df.iloc[0])

Numbers 10

Name: 0, dtype: int64

Access the first row using loc (label-based indexing)

print(df.loc[0])

3.Access a Specific Element (Cell):

- You can access a specific element using a combination of row and column.

Access the element at the first row and 'Numbers' column

print(df.loc[0, 'Numbers'])

10