EXPERIMENT – 01

AIM:

Implement indexing, slicing and reshaping on numpy arrays.

DESCRIPTION:

This experiment focuses on implementing indexing, slicing, and reshaping operations on NumPy arrays in Python, which are essential techniques for efficient data manipulation. Indexing allows direct access to individual elements or groups of elements in arrays, ensuring precise retrieval of data. Slicing extends this by enabling the extraction of continuous or step-based sub-arrays using the start, stop, and step parameters, which makes handling large datasets easier and more flexible. Reshaping further demonstrates the power of NumPy by allowing one-dimensional arrays to be reorganized into multidimensional forms without altering the actual data, thereby facilitating structured storage, matrix operations, and compatibility with machine learning models. By performing these operations, the experiment highlights the versatility of NumPy arrays in scientific computing, data analysis, and real-world problem-solving.

PROGRAM:

1. **Slicing and Reshaping**

```
import numpy as np
a = np.arange(1, 13)
a_reshaped = a[:8].reshape((2, 4))
print(a_reshaped)
```

```
[[1 2 3 4]
 [5 6 7 8]]
```

2. **Reshaping (3d array)**

```
import numpy as np
a = np.arange(24)
a_3d = a.reshape((2, 3, 4))
print(a_3d)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

3. Matrix Slicing

```
matrix = np.arange(25).reshape((5, 5))
center_block = matrix[1:4, 1:4]
print(center_block)
```

```
[[ 6  7  8]
 [11 12 13]
 [16 17 18]]
```

4. Reversing array using Slicing

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

```
[[3 2 1]
 [6 5 4]]
```

5. Concatenation of Numpy arrays – Row Wise

```
a1 = np.array([[1, 2, 3], [4, 5, 6]])
a2 = np.array([[7, 8, 9], [10, 11, 12]])
concat_rows = np.concatenate((a1, a2), axis=0)
print(concat_rows)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

6. Concatenation of Numpy arrays – Column Wise

```
concat_cols = np.concatenate((a1, a2), axis=1)
print(concat_cols)
```

```
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

7. Vertical, Horizontal and Depth-wise Stacking

```
a = np.array([[1, 2], [3, 4]])
print(np.vstack((a, a)))
print(np.hstack((a, a)))
print(np.dstack((a, a)))
print(np.stack((a, a)))
print(a.shape)
print(np.arange(10))
print(a.size)
```

```
[[1 2]
 [3 4]
 [1 2]
 [3 4]]
[[1 2 1 2]
 [3 4 3 4]]
[[[1 1]
  [2 2]]

 [[3 3]
  [4 4]]]
[[[1 2]
  [3 4]]

 [[1 2]
  [3 4]]]
(2, 2)
[0 1 2 3 4 5 6 7 8 9]
4
```

8. Vertical, Horizontal and Depth-wise Stacking

```
arr = np.arange(1, 17).reshape((4, 4))
print("Element at (1,2):", arr[1, 2])
print("First 2 rows:\n", arr[:2])
reshaped = arr.reshape((2, 8))
print("Reshaped:\n", reshaped)
```

```
Element at (1,2): 7
First 2 rows:
 [[1 2 3 4]
 [5 6 7 8]]
Reshaped:
 [[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]]
```

OUTPUT ANALYSIS:

The experiment successfully demonstrated fundamental NumPy operations such as indexing, slicing, reshaping, concatenation, and stacking. These operations highlight the flexibility of NumPy arrays in accessing elements, extracting sub-arrays, reorganizing data into different dimensions, and combining arrays in multiple ways. Such array manipulations form the foundation for efficient data handling, numerical computations, and advanced applications in scientific computing and machine learning.

EXPERIMENT – 02

AIM:

Implement various operations on NumPy arrays, including arithmetic calculations, broadcasting.

DESCRIPTION:

This experiment focuses on implementing various operations on NumPy arrays, including element-wise arithmetic calculations and broadcasting. Arithmetic operations such as addition, subtraction, multiplication, and division are performed directly on arrays, enabling fast and efficient numerical computations. Broadcasting further extends these operations by allowing arrays of different shapes to interact without explicit looping, automatically stretching dimensions to make computations possible. Together, these features demonstrate the power of NumPy in handling large-scale numerical data, simplifying complex operations, and providing a foundation for data analysis, machine learning, and scientific computing.

PROGRAM:

1. **Element wise - Arithmetic Operations (+,-,*,/)**

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
```

2. **1D array and 2D array addition using broadcasting feature**

```
a = np.array([[1, 2], [3, 4]])
b = np.array([10, 20])
print(a + b)
```

```
[[11 22]
 [13 24]]
```

3. **Universal Functions on Numpy Arrays**

```
a = np.array([1, -2, 3])
np.abs(a)
np.sqrt([1, 4, 9])
np.exp(a)
np.log([1, np.e])
```

```
array([0., 1.])
```

4. **Aggregations on Numpy Arrays**

```
a = np.array([[1, 2, 3], [4, 5, 6]])
np.sum(a)
np.sum(a, axis=0)
np.sum(a, axis=1)
np.mean(a)
np.min(a), np.max(a)
np.std(a)
```

```
np.float64(1.707825127659933)
```

5. **Logical Operations on Numpy Arrays**

```
a = np.array([1, 2, 3])
b = np.array([2, 2, 2])
a == b
a < b
np.all(a < b)
np.any(a < b)
```

```
np.True_
```

6. **Matrix Multiplications Operations**

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
np.dot(A, B)
np.matmul(A, B)
A.T
np.linalg.inv(A)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

CONCLUSION:

The experiment successfully demonstrated arithmetic operations and broadcasting in NumPy arrays. It showed how element-wise calculations can be performed efficiently and how broadcasting simplifies computations between arrays of different shapes, making numerical processing more powerful and flexible.

EXPERIMENT – 03

AIM:

Apply Boolean operations on a Numpy array to filter specific elements, sort the array, and manipulate it using Boolean masks.

DESCRIPTION:

This experiment demonstrates the application of Boolean operations on NumPy arrays to filter, sort, and manipulate data efficiently. Boolean indexing is used to create masks that select only those elements which satisfy specific conditions, enabling precise filtering without explicit loops. Sorting operations arrange array elements in ascending or descending order for better organization and analysis. Boolean masks further enhance manipulation by allowing conditional updates or extraction of subsets, making NumPy a powerful tool for handling large datasets in data science and numerical computing.

PROGRAM:

1. **Boolean masks filter students into pass and fail groups based on averages.**

```
import numpy as np
```

```
students = np.array(["Alice", "Bob", "Charlie", "David", "Eva", "Frank", "Grace", "Hannah"])
scores = np.array([
    [78, 85, 80, 82],
    [35, 42, 37, 39],
    [55, 60, 50, 52],
    [20, 25, 15, 18],
    [88, 92, 85, 90],
    [40, 38, 45, 42],
    [67, 70, 75, 72],
    [30, 33, 28, 31]
])
```

```
averages = scores.mean(axis=1)
passed_mask = averages >= 40
failed_mask = averages < 40
print(" Passed students:", students[passed_mask])
print(" Failed students:", students[failed_mask])
```

```
 Passed students: ['Alice' 'Charlie' 'Eva' 'Frank' 'Grace']
 Failed students: ['Bob' 'David' 'Hannah']
```

2. Sorting, score updates, and Boolean filters identify top students.

```
[6] sorted_indices = np.argsort(-averages)
    print("\n Students sorted by average (high → low):")
    print(students[sorted_indices])
    print(averages[sorted_indices])
```

```
 Students sorted by average (high → low):
['Eva' 'Alice' 'Grace' 'Charlie' 'Frank' 'Bob' 'Hannah' 'David']
[88.75 81.25 71.   54.25 41.25 38.25 30.5  19.5 ]
```

```
[7] scores[failed_mask] += 5
    new_averages = scores.mean(axis=1)
```

```
[8] math_high_mask = scores[:, 0] > 80
    print("\n Scored >80 in Math:", students[math_high_mask])
```

```
 Scored >80 in Math: ['Eva']
```

```
[9] sci_high_pass_mask = (passed_mask) & (scores[:, 1] > 70)
    print("\n Passed & Science > 70:", students[sci_high_pass_mask])
```

```
 Passed & Science > 70: ['Alice' 'Eva']
```

3. Masks used to filter, update, and check student scores.

```
[11] fail_or_low_hist = (failed_mask) | (scores[:, 3] < 35)
     print("\n Failed OR History < 35:", students[fail_or_low_hist])
```

```
 Failed OR History < 35: ['Bob' 'David' 'Hannah']
```

```
eng_range_mask = (scores[:, 2] >= 50) & (scores[:, 2] <= 80)
print("\n English 50-80:", students[eng_range_mask])
```

```
 English 50-80: ['Alice' 'Charlie' 'Grace']
```

```
[13] low_scores_mask = scores < 40
     scores[low_scores_mask] = 40
```

```
all_above_50 = np.all(scores >= 50, axis=1)
print("\n All subjects ≥50:", students[all_above_50])
```

```
 All subjects ≥50: ['Alice' 'Charlie' 'Eva' 'Grace']
```

```
[15] any_above_90 = np.any(scores >= 90, axis=1)
     print("\n Any subject ≥90:", students[any_above_90])
```

```
 Any subject ≥90: ['Eva']
```

4. NumPy is used to generate, modify, and analyze student scores.

```
print("\n Updated scores after modifications:")
for i in range(len(students)):
    print(f"{students[i]}: {scores[i]} (Avg: {scores[i].mean():.2f})")
```

```
 Updated scores after modifications:
Alice: [78 85 80 82] (Avg: 81.25)
Bob: [40 47 42 44] (Avg: 43.25)
Charlie: [55 60 50 52] (Avg: 54.25)
David: [40 40 40 40] (Avg: 40.00)
Eva: [88 92 85 90] (Avg: 88.75)
Frank: [40 40 45 42] (Avg: 41.75)
Grace: [67 70 75 72] (Avg: 71.00)
Hannah: [40 40 40 40] (Avg: 40.00)
```

```
num_students = 1000
num_subjects = 5

student_names = [f"Student_{i}" for i in range(num_students)]
scores = np.random.randint(0, 101, size=(num_students, num_subjects))

students = np.array(student_names)

subjects = np.array(["Math", "Science", "English", "History", "Art"])

print("Generated scores shape:", scores.shape)
```

```
Generated scores shape: (1000, 5)
```

CONCLUSION:

This experiment demonstrates that Boolean operations in NumPy provide a powerful and efficient way to filter, sort, and manipulate arrays. By applying Boolean masks, elements meeting specific conditions can be selectively extracted or modified, ensuring precise data handling without explicit loops. Sorting operations further organize the data for easier interpretation, while conditional updates through Boolean indexing allow targeted manipulation of subsets. Thus, Boolean operations and masks greatly simplify array processing and are essential tools in data science, machine learning, and numerical analysis.

EXPERIMENT – 04

AIM:

Demonstrate identifying and handling missing data within Pandas DataFrames using various techniques, such as filling missing values, dropping rows or columns with missing data.

DESCRIPTION:

This experiment demonstrates techniques for identifying and handling missing data within Pandas DataFrames, a common challenge in data analysis. Using functions like isnull() and notnull(), missing values can be detected and analyzed. Various strategies are then applied, including filling missing values with constants, statistical measures (mean, median, mode), or forward/backward fill methods to maintain data consistency. Alternatively, rows or columns containing missing values can be dropped to ensure dataset integrity. These techniques highlight the flexibility of Pandas in preprocessing datasets, ensuring cleaner and more reliable data for analysis and machine learning tasks.

PROGRAM:

1. **DataFrame with student marks containing missing values.**

```
[1]  import pandas as pd
     import numpy as np
```

```
data = {
    "Student": ["A", "B", "C", "D", "E"],
    "Maths": [85, np.nan, 78, 92, np.nan],
    "Science": [88, 76, np.nan, 95, 67],
    "English": [np.nan, 74, 80, 90, 65]
}
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
  Student  Maths  Science  English
0       A   85.0     88.0      NaN
1       B    NaN     76.0     74.0
2       C   78.0      NaN     80.0
3       D   92.0     95.0     90.0
4       E    NaN     67.0     65.0
```

2. Missing values are detected and removed.

```
[3] print("\nCheck for missing values:")
    print(df.isnull())
    print("\nCount of missing values per column:")
    print(df.isnull().sum())
```

```
Check for missing values:
   Student  Maths  Science  English
0    False  False    False     True
1    False   True    False    False
2    False  False     True    False
3    False  False    False    False
4    False   True    False    False

Count of missing values per column:
Student    0
Maths      2
Science    1
English    1
dtype: int64
```

```
print("\nDrop rows with any missing value:")
print(df.dropna())
print("\nDrop columns with any missing value:")
print(df.dropna(axis=1))
```

```
Drop rows with any missing value:
  Student  Maths  Science  English
3       D   92.0     95.0     90.0
```

3. Missing values handled using fixed value.

```
print("\nFill missing values with a fixed number (e.g., 0):")
print(df.fillna(0))
print("\nFill missing values with column mean:")
print(df.fillna(df.mean(numeric_only=True)))
```

```
Fill missing values with a fixed number (e.g., 0):
  Student  Maths  Science  English
0       A   85.0     88.0      0.0
1       B    0.0     76.0     74.0
2       C   78.0      0.0     80.0
3       D   92.0     95.0     90.0
4       E    0.0     67.0     65.0

Fill missing values with column mean:
  Student  Maths  Science  English
0       A   85.0     88.0    77.25
1       B   85.0     76.0    74.00
2       C   78.0     81.5    80.00
3       D   92.0     95.0    90.00
4       E   85.0     67.0    65.00
```

```
print("\nForward fill (propagate previous value):")
print(df.fillna(method="ffill"))
```

```
Forward fill (propagate previous value):
  Student  Maths  Science  English
0       A   85.0     88.0      NaN
1       B   85.0     76.0     74.0
2       C   78.0     76.0     80.0
3       D   92.0     95.0     90.0
```

4. Missing values handled using backward fill or column-specific replacement.

```
print("\nBackward fill (propagate next value):")
print(df.fillna(method="bfill"))
```

```
Backward fill (propagate next value):
  Student  Maths  Science  English
0       A   85.0     88.0     74.0
1       B   78.0     76.0     74.0
2       C   78.0     95.0     80.0
3       D   92.0     95.0     90.0
4       E    NaN     67.0     65.0
/tmp/ipython-input-1986577143.py:2: FutureWarning: DataFrame
  print(df.fillna(method="bfill"))
```

```
print("\nFill only English column NaN with 'Absent':")
df2 = df.copy()
df2["English"] = df2["English"].fillna("Absent")
print(df2)
```

```
Fill only English column NaN with 'Absent':
  Student  Maths  Science English
0       A   85.0     88.0  Absent
1       B    NaN     76.0    74.0
2       C   78.0      NaN    80.0
3       D   92.0     95.0    90.0
4       E    NaN     67.0    65.0
```

CONCLUSION:

This experiment demonstrates that Pandas provides efficient techniques to identify and handle missing data within DataFrames. Methods such as fillna() allow replacing null values with constants, computed values, or propagated entries, while dropna() enables the removal of incomplete rows or columns. These operations ensure data consistency and reliability, making datasets cleaner and better suited for accurate analysis and decision-making.

EXPERIMENT – 07

AIM:

Combine and analyze datasets using Pandas operations including merge, join, concatenate, grouping and aggregation.

DESCRIPTION:

This experiment demonstrates how Pandas enables combining and analyzing multiple datasets. Different operations are explored:

- **Concatenate:** Stacking datasets either row-wise or column-wise.
- **Merge:** Combining datasets based on common columns (like SQL joins).
- **Join:** Combining datasets using index or key columns.
- **Grouping:** Splitting data into groups based on column values.
- **Aggregation:** Applying statistical functions (sum, mean, count, etc.) on grouped data.

These operations are widely used in data analysis and preprocessing, enabling analysts to build larger, cleaner, and more meaningful datasets from multiple sources.

PROGRAM:

[1]
```
import pandas as pd
```

[2]
```
file_path = "/content/BMW sales data (2010-2024) (1).csv"
df = pd.read_csv(file_path)
```

[ ]
```
df1 = df.head(3)
df2 = df.tail(3)
concat_rows = pd.concat([df1, df2], axis=0)
print("Row-wise concatenation:\n", concat_rows, "\n")
```

```
Row-wise concatenation:
           Model  Year         Region  Color Fuel_Type Transmission  \
0      5 Series  2016           Asia    Red    Petrol       Manual
1            i8  2013  North America    Red    Hybrid    Automatic
2      5 Series  2022  North America   Blue    Petrol    Automatic
49997  5 Series  2010    Middle East    Red    Petrol    Automatic
49998        i3  2020           Asia  White  Electric    Automatic
49999        X1  2020  North America   Blue    Diesel       Manual

       Engine_Size_L  Mileage_KM  Price_USD  Sales_Volume Sales_Classification
0                3.5      151748      98740          8300                 High
1                1.6      121671      79219          3428                  Low
2                4.5       10991     113265          6994                  Low
49997            4.5      174939      46126          8280                 High
49998            3.8        3379      58566          9486                 High
49999            3.3      171003      77492          1764                  Low
```

1. Concatenation

```
df1 = df.head(3).reset_index(drop=True)
df2 = df.tail(3).reset_index(drop=True)
concat_cols = pd.concat([df1, df2], axis=1)
print("Column-wise concatenation:\n", concat_cols, "\n")
```

```
Column-wise concatenation:
      Model  Year         Region  Color Fuel_Type Transmission Engine_Size_L  \
0  5 Series  2016           Asia    Red    Petrol       Manual           3.5
1        i8  2013  North America    Red    Hybrid    Automatic           1.6
2  5 Series  2022  North America   Blue    Petrol    Automatic           4.5

  Mileage_KM  Price_USD  Sales_Volume  ...  Year         Region  Color  \
0     151748      98740          8300  ...  2010    Middle East    Red
1     121671      79219          3428  ...  2020           Asia  White
2      10991     113265          6994  ...  2020  North America   Blue

  Fuel_Type Transmission Engine_Size_L Mileage_KM  Price_USD  Sales_Volume  \
0    Petrol    Automatic           4.5     174939      46126          8280
1  Electric    Automatic           3.8       3379      58566          9486
2    Diesel       Manual           3.3     171003      77492          1764

   Sales_Classification
0                  High
1                  High
2                   Low

[3 rows x 22 columns]
```

2. Merge

```
region_info = pd.DataFrame({
    "Region": ["Asia", "Europe", "North America", "South America", "Middle East"],
    "Manager": ["Amit", "Sophie", "John", "Carlos", "Fatima"]
})

merged_left = pd.merge(df.head(5), region_info, on="Region", how="left")
print("Left Merge:\n", merged_left, "\n")


merged_inner = pd.merge(df.head(5), region_info, on="Region", how="inner")
print("Inner Merge:\n", merged_inner, "\n")
```

```
Left Merge:
      Model  Year         Region  Color Fuel_Type Transmission  Engine_Size_L  \
0  5 Series  2016           Asia    Red    Petrol       Manual            3.5
1        i8  2013  North America    Red    Hybrid    Automatic            1.6
2  5 Series  2022  North America   Blue    Petrol    Automatic            4.5
3        X3  2024    Middle East   Blue    Petrol    Automatic            1.7
4  7 Series  2020  South America  Black    Diesel       Manual            2.1

   Mileage_KM  Price_USD  Sales_Volume Sales_Classification Manager
0      151748      98740          8300                 High    Amit
1      121671      79219          3428                  Low    John
2       10991     113265          6994                  Low    John
3       27255      60971          4047                  Low  Fatima
4      122131      49898          3080                  Low  Carlos

Inner Merge:
      Model  Year         Region  Color Fuel_Type Transmission  Engine_Size_L  \
0  5 Series  2016           Asia    Red    Petrol       Manual            3.5
1        i8  2013  North America    Red    Hybrid    Automatic            1.6
2  5 Series  2022  North America   Blue    Petrol    Automatic            4.5
3        X3  2024    Middle East   Blue    Petrol    Automatic            1.7
4  7 Series  2020  South America  Black    Diesel       Manual            2.1

   Mileage_KM  Price_USD  Sales_Volume Sales_Classification Manager
0      151748      98740          8300                 High    Amit
1      121671      79219          3428                  Low    John
2       10991     113265          6994                  Low    John
3       27255      60971          4047                  Low  Fatima
4      122131      49898          3080                  Low  Carlos
```

3. Join

```
df_left = df.set_index("Model").head(5)
df_right = df.groupby("Model")[["Sales_Volume"]].sum()
join_default = df_left.join(df_right, rsuffix="_Total")
print("Default Join on index:\n", join_default, "\n")
join_inner = df_left.join(df_right, rsuffix="_Total", how="inner")
print("Inner Join on index:\n", join_inner, "\n")
```

```
Default Join on index:
            Year         Region  Color Fuel_Type Transmission  Engine_Size_L  \
Model
5 Series  2016           Asia    Red    Petrol       Manual            3.5
i8        2013  North America    Red    Hybrid    Automatic            1.6
5 Series  2022  North America   Blue    Petrol    Automatic            4.5
X3        2024    Middle East   Blue    Petrol    Automatic            1.7
7 Series  2020  South America  Black    Diesel       Manual            2.1

          Mileage_KM  Price_USD  Sales_Volume Sales_Classification  \
Model
5 Series      151748      98740          8300                 High
i8            121671      79219          3428                  Low
5 Series       10991     113265          6994                  Low
X3             27255      60971          4047                  Low
7 Series      122131      49898          3080                  Low
Inner Join on index:
            Year         Region  Color Fuel_Type Transmission  Engine_Size_L  \
Model
5 Series  2016           Asia    Red    Petrol       Manual            3.5
i8        2013  North America    Red    Hybrid    Automatic            1.6
5 Series  2022  North America   Blue    Petrol    Automatic            4.5
X3        2024    Middle East   Blue    Petrol    Automatic            1.7
7 Series  2020  South America  Black    Diesel       Manual            2.1

          Mileage_KM  Price_USD  Sales_Volume Sales_Classification  \
Model
5 Series      151748      98740          8300                 High
i8            121671      79219          3428                  Low
5 Series       10991     113265          6994                  Low
X3             27255      60971          4047                  Low
7 Series      122131      49898          3080                  Low
```

4. Grouping

```
group_avg_price = df.groupby("Region")["Price_USD"].mean()
print("Average Price by Region:\n", group_avg_price, "\n")
group_sales_by_year = df.groupby("Year")["Sales_Volume"].sum()
print("Total Sales by Year:\n", group_sales_by_year, "\n")
```

```
Average Price by Region:
 Region
Africa           74885.771598
Asia             75554.925006
Europe           74988.356851
Middle East      74726.788487
North America    75070.054709
South America    74973.598837
Name: Price_USD, dtype: float64

Total Sales by Year:
 Year
2010    16933445
2011    16758941
2012    16751895
2013    16866733
2014    16958960
2015    17010207
2016    16957550
2017    16620811
```

5. Aggregation

```
agg_fuel = df.groupby("Fuel_Type").agg({
    "Price_USD": ["mean", "max"]
})
print("Aggregation by Fuel Type:\n", agg_fuel, "\n")
agg_model = df.groupby("Model").agg({
    "Mileage_KM": ["mean", "min"],
    "Sales_Volume": "sum"
})
print("Aggregation by Model:\n", agg_model.head(), "\n")
```

```
Aggregation by Fuel Type:
                Price_USD
                    mean     max
Fuel_Type
Diesel      75079.809671  119997
Electric    75276.313207  119998
Hybrid      74797.551746  119994
Petrol      74990.419841  119996

Aggregation by Model:
              Mileage_KM     Sales_Volume
                    mean min          sum
Model
3 Series  100159.898368  36     23281303
5 Series  101356.634582  21     23097519
7 Series  100792.165881  23     23786466
M3         99729.770451  69     22349694
M5        102342.995757  42     22779688
```

CONCLUSION:

This experiment demonstrates that Pandas provides powerful operations to combine and analyze datasets. Concatenation allows stacking datasets, while merge and join enable SQL-like table operations. Grouping and aggregation support advanced analysis by summarizing data across categories. These techniques make data manipulation more flexible and efficient for real-world analytics tasks.

EXPERIMENT – 05

AIM:

Demonstrate Interpolation Techniques to estimate missing values.

DESCRIPTION:

This experiment demonstrates the use of interpolation techniques to estimate and fill missing values in a dataset. Interpolation is a method used to predict unknown data points within the range of a discrete set of known data points. It is commonly used in data preprocessing to handle incomplete datasets without discarding valuable information.

Different interpolation methods such as linear, polynomial, and spline interpolation can be applied depending on the data pattern.

- **Linear interpolation** assumes a straight-line relationship between known data points.
- **Polynomial interpolation** fits a polynomial curve through the data for smoother estimates.
- **Spline interpolation** uses piecewise polynomials to maintain smoothness and accuracy.

By applying these techniques, missing values are estimated realistically based on the surrounding data, ensuring dataset consistency and reliability for further analysis.

PROGRAM:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import interpolate
```

```
# Create a sample dataset with missing values
x = np.arange(0, 10)
y = np.sin(x)
y[[2, 5, 7]] = np.nan   # introduce missing values

# Create a DataFrame
data = pd.DataFrame({'X': x, 'Y': y})
print("Original Data with Missing Values:")
print(data)
```

```
Original Data with Missing Values:
   X         Y
0  0  0.000000
1  1  0.841471
2  2       NaN
3  3  0.141120
4  4 -0.756802
5  5       NaN
6  6 -0.279415
7  7       NaN
8  8  0.989358
9  9  0.412118
```

Pandas Interpolation

1. Linear interpolation

```
data['Linear'] = data['Y'].interpolate(method='linear')
```

2. Polynomial interpolation (order=2)

```
data['Polynomial'] = data['Y'].interpolate(method='polynomial', order=2)
```

3. Spline interpolation (order=3)

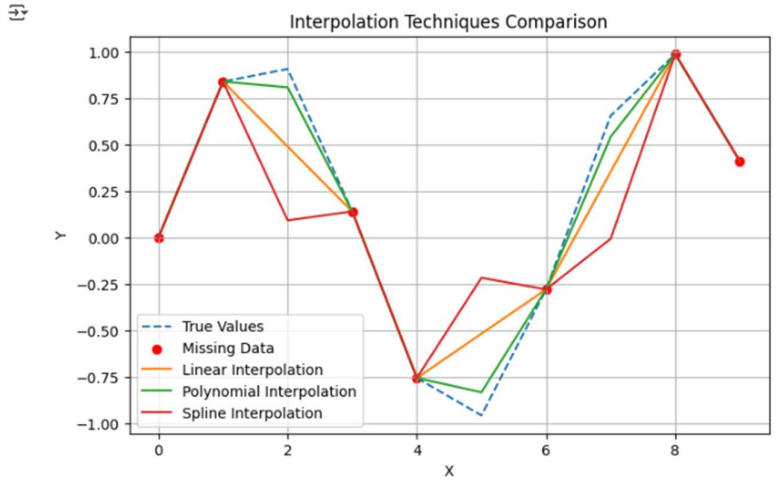
```
data['Spline'] = data['Y'].interpolate(method='spline', order=3)
```

```
print("\nAfter Interpolation:")
print(data)
```

```
After Interpolation:
   X         Y    Linear  Polynomial    Spline
0  0  0.000000  0.000000    0.000000  0.000000
1  1  0.841471  0.841471    0.841471  0.841471
2  2       NaN  0.491295    0.810343  0.092972
3  3  0.141120  0.141120    0.141120  0.141120
4  4 -0.756802 -0.756802   -0.756802 -0.756802
5  5       NaN -0.518109   -0.834084 -0.216721
6  6 -0.279415 -0.279415   -0.279415 -0.279415
7  7       NaN  0.354971    0.543623 -0.006340
8  8  0.989358  0.989358    0.989358  0.989358
9  9  0.412118  0.412118    0.412118  0.412118
```

--- Visualization ---

```
plt.figure(figsize=(8,5))
plt.plot(x, np.sin(x), label='True Values', linestyle='--')
plt.scatter(x, y, color='red', label='Missing Data')
plt.plot(x, data['Linear'], label='Linear Interpolation')
plt.plot(x, data['Polynomial'], label='Polynomial Interpolation')
plt.plot(x, data['Spline'], label='Spline Interpolation')
plt.legend()
plt.title("Interpolation Techniques Comparison")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```



OUTPUT ANALYSIS:

The experiment showed that interpolation effectively estimates missing values.

- **Linear interpolation** filled gaps smoothly using straight-line connections.
- **Polynomial interpolation** captured curved trends but may overfit fluctuating data.
- **Spline interpolation** provided smooth and accurate estimates. Overall, interpolation restored missing data realistically, improving dataset completeness and reliability for analysis.

CONCLUSION:

The experiment concludes that interpolation is an effective technique for estimating missing values in datasets. By using methods like linear, polynomial, and spline interpolation, missing data can be filled accurately without distorting the overall trend. This ensures data continuity, enhances dataset quality, and supports more accurate analysis and modeling.

EXPERIMENT – 06

AIM:

Demonstrate Hierarchical Indexing and Multi-Criteria based data retrieval using pandas.

DESCRIPTION:

This experiment demonstrates the use of hierarchical indexing and multi-criteria based data retrieval in Pandas for advanced data organization and analysis.

- **Hierarchical indexing (MultiIndex)** allows a DataFrame to have multiple levels of row or column labels, enabling the representation of higher-dimensional data in a 2D structure. It helps in grouping, summarizing, and performing complex data operations efficiently.
- **Multi-criteria data retrieval** involves accessing data based on multiple conditions or index levels. This can be done using combinations of labels, Boolean conditions, or logical operators to filter data precisely.

Together, these techniques enhance data manipulation flexibility, making it easier to handle complex datasets, perform detailed analysis, and extract meaningful insights from multi-level data structures.

PROGRAM:

```
import pandas as pd

data = {
    'City': ['Hyderabad', 'Hyderabad', 'Hyderabad',
             'Chennai', 'Chennai', 'Chennai',
             'Bangalore', 'Bangalore', 'Bangalore'],
    'Year': [2022, 2023, 2024,
             2022, 2023, 2024,
             2022, 2023, 2024],
    'Temperature': [30, 32, 33, 29, 31, 34, 28, 30, 31],
    'Rainfall': [120, 140, 110, 150, 130, 125, 160, 155, 170]
}

df = pd.DataFrame(data)
print("Original DataFrame:\n")
print(df)
```

```
Original DataFrame:

        City  Year  Temperature  Rainfall
0  Hyderabad  2022           30       120
1  Hyderabad  2023           32       140
2  Hyderabad  2024           33       110
3    Chennai  2022           29       150
4    Chennai  2023           31       130
5    Chennai  2024           34       125
6  Bangalore  2022           28       160
7  Bangalore  2023           30       155
8  Bangalore  2024           31       170
```

Setting Hierarchical Index

```
df = df.set_index(['City', 'Year'])
print("\nDataFrame with Hierarchical Index:\n")
print(df)
```

```
DataFrame with Hierarchical Index:

               Temperature  Rainfall
City      Year
Hyderabad 2022           30       120
          2023           32       140
          2024           33       110
Chennai   2022           29       150
          2023           31       130
          2024           34       125
Bangalore 2022           28       160
          2023           30       155
          2024           31       170
```

Access data using hierarchical indexing

```
# Example 1: Retrieve data for a specific city
print("\nData for Hyderabad:\n")
print(df.loc['Hyderabad'])
```

```
Data for Hyderabad:

      Temperature  Rainfall
Year
2022           30       120
2023           32       140
2024           33       110
```

```
# Example 2: Retrieve data for a specific city and year
print("\nData for Chennai, 2023:\n")
print(df.loc[('Chennai', 2023)])
```

```
Data for Chennai, 2023:

Temperature     31
Rainfall       130
Name: (Chennai, 2023), dtype: int64
```

```
# Example 3: Retrieve data for multiple cities
print("\nData for Hyderabad and Bangalore:\n")
print(df.loc[['Hyderabad', 'Bangalore']])
```

```
Data for Hyderabad and Bangalore:

               Temperature  Rainfall
City      Year
Hyderabad 2022           30       120
          2023           32       140
          2024           33       110
Bangalore 2022           28       160
          2023           30       155
          2024           31       170
```

Multi-criteria data retrieval

```
filtered = df[(df['Temperature'] > 30) & (df['Rainfall'] < 150)]
print("\nFiltered Data (Temp > 30 & Rainfall < 150):\n")
print(filtered)
```

```
Filtered Data (Temp > 30 & Rainfall < 150):

               Temperature  Rainfall
City      Year
Hyderabad 2023           32       140
          2024           33       110
Chennai   2023           31       130
          2024           34       125
```

OUTPUT ANALYSIS:

The experiment successfully demonstrated hierarchical indexing and multi-criteria data retrieval in Pandas.

- **Hierarchical indexing (MultiIndex)** allowed the DataFrame to represent multiple levels of data.
- **Multi-criteria retrieval** enabled precise filtering of data based on multiple conditions.

The results show that combining hierarchical indexing with multi-criteria filtering improves data accessibility and simplifies analysis of large, structured datasets.

CONCLUSION:

The experiment concludes that hierarchical indexing and multi- criteria data retrieval in Pandas are highly effective for managing and analyzing complex datasets. Hierarchical indexing enables the representation of multi-level data in a structured manner, while multi- criteria retrieval allows precise filtering and extraction of relevant information. Together, these techniques improve data organization, enhance analytical flexibility, and simplify operations on large and structured datasets, making data analysis more efficient and insightful.

EXPERIMENT – 08

AIM:

Plot different types of visualizations (e.g., line plot, scatter plot, histogram, bar plot) using matplotlib.

DESCRIPTION:

This experiment demonstrates how to visualize data using the **Matplotlib library** in Python. Different types of plots such as line plots, scatter plots, histograms, and bar plots are created to represent various data patterns and relationships.

- **The Line Plot** helps in showing trends or changes in data over time.
- **The Scatter Plot** is useful for analyzing the relationship between two continuous variables.
- **The Histogram** displays the frequency distribution of data, showing how values are spread.
- **The Bar Plot** compares different categories or groups using rectangular bars.

These visualizations are essential in data analysis as they help in understanding datasets, identifying patterns, detecting outliers, and making data-driven decisions effectively.

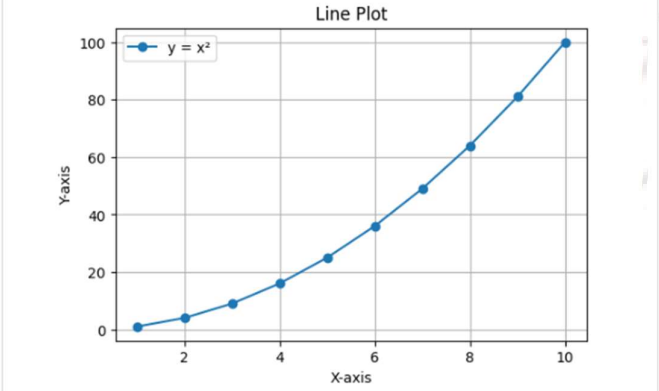
PROGRAM:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(1, 11)
y = x ** 2
z = np.random.randint(10, 100, size=10)
```

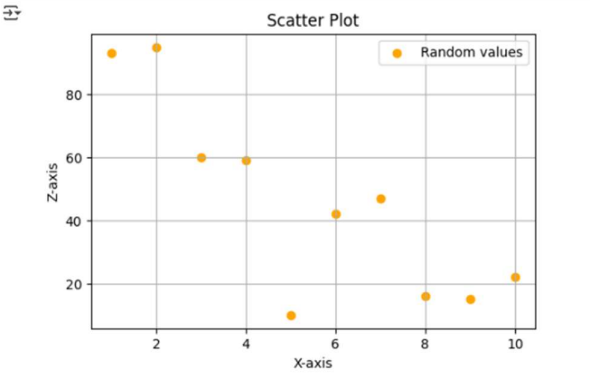
Line Plot

```
plt.figure(figsize=(6,4))
plt.plot(x, y, marker='o', linestyle='-', label='y = x²')
plt.title('Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()
plt.grid(True)
plt.show()
```



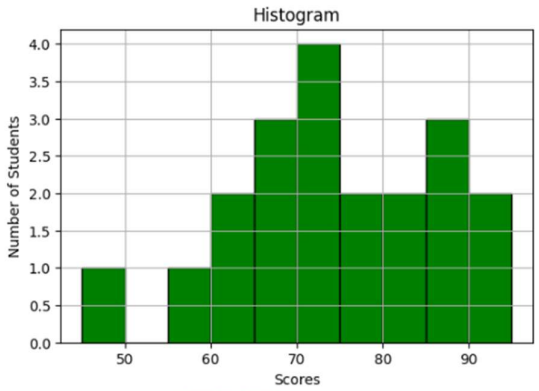
Scatter Plot

```
plt.figure(figsize=(6,4))
plt.scatter(x, z, color='orange', label='Random values')
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Z-axis')
plt.legend()
plt.grid(True)
plt.show()
```



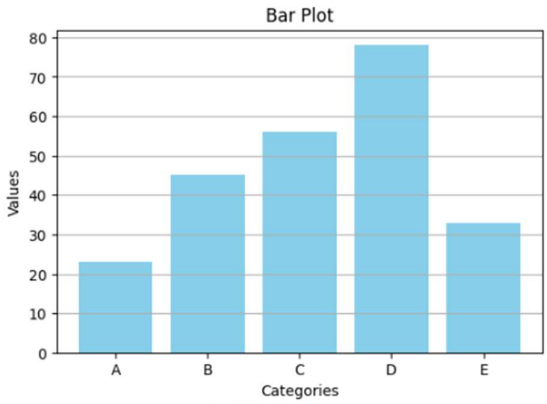
Histogram

```
# Fixed dataset (example: exam scores of 20 students)
data = [45, 55, 60, 62, 65, 67, 68, 70, 72, 73,
        74, 76, 78, 80, 82, 85, 87, 88, 90, 95]
plt.figure(figsize=(6,4))
plt.hist(data, bins=10, color='green', edgecolor='black')
plt.title('Histogram')
plt.xlabel('Scores')
plt.ylabel('Number of Students')
plt.grid(True)
plt.show()
```



Bar Plot

```
# --- Step 5: Bar Plot ---
categories = ['A', 'B', 'C', 'D', 'E']
values = [23, 45, 56, 78, 33]
plt.figure(figsize=(6,4))
plt.bar(categories, values, color='skyblue')
plt.title('Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.grid(True, axis='y')
plt.show()
```



OUTPUT ANALYSIS:

The experiment successfully generated four distinct visulizations using Matplotlib:

1. **Line Plot:** The sine and cosine curves clearly show periodic behavior. The smooth lines illustrate continuous changes in values, demonstrating trends effectively over the range of x-values.
2. **Scatter Plot:** The scatter points display individual data values without connecting lines, making it easier to observe the relationship and distribution between the x and y variables.
3. **Histogram:** The histogram shows the frequency distribution of randomly generated data. The bell-shaped curve indicates that the data follows a normal (Gaussian) distribution, with most values concentrated around the mean.
4. **Bar Plot:** The bar chart compares categorical data effectively. Each bar represents a category, and the varying heights show the differences in their corresponding values.

Overall, the output demonstrates how Matplotlib can be used to visualize and interpret data in multiple formats, helping analysts gain insights into data patterns, distributions, and relationships.

CONCLUSION:

This experiment demonstrates the versatility and effectiveness of Matplotlib in visualizing different types of data. By creating line, scatter, histogram, and bar plots, it becomes easier to interpret complex datasets and identify key trends, relationships, and distributions. Visualization simplifies data analysis by providing a clear graphical representation, making insights more intuitive and decision-making more data-driven. Thus, Matplotlib serves as an essential tool for exploratory data analysis and reporting in Python.

EXPERIMENT – 09

AIM:

Demonstrate 3D visualizations using matplotlib.

DESCRIPTION:

This experiment demonstrates how to create 3D visualizations using the Matplotlib library in Python. 3D plotting allows data to be represented in three dimensions — X, Y, and Z — providing a deeper understanding of relationships and patterns that are not easily visible in 2D plots.
Matplotlib's mpl_toolkits.mplot3d module enables the creation of various 3D plots such as 3D line plots, scatter plots, surface plots, and wireframe plots. These visualizations help in exploring complex datasets, analyzing spatial relationships, and visualizing mathematical functions or multi-dimensional data.
By using 3D visualization, analysts can interpret data more intuitively and gain insights into patterns, trends, and variations that enhance data- driven understanding and presentation.

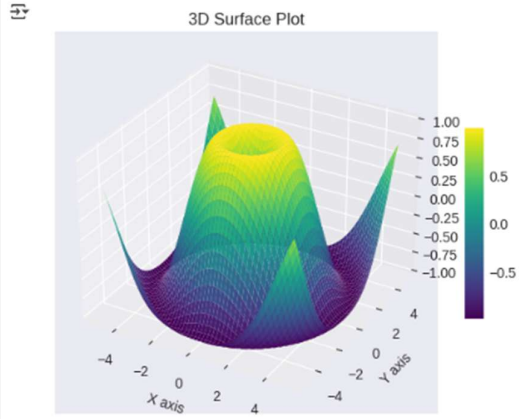
PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

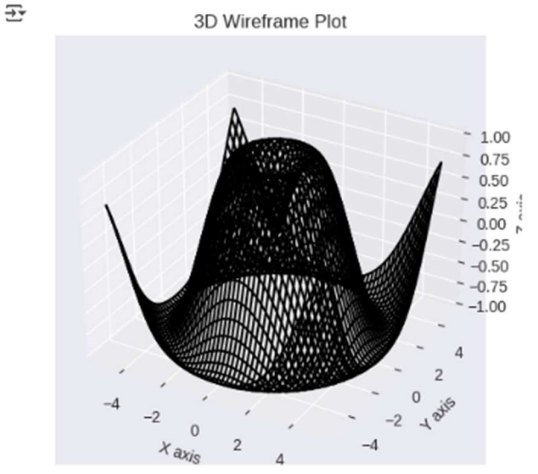
```
plt.style.use('seaborn-v0_8-darkgrid')
```

```
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
```

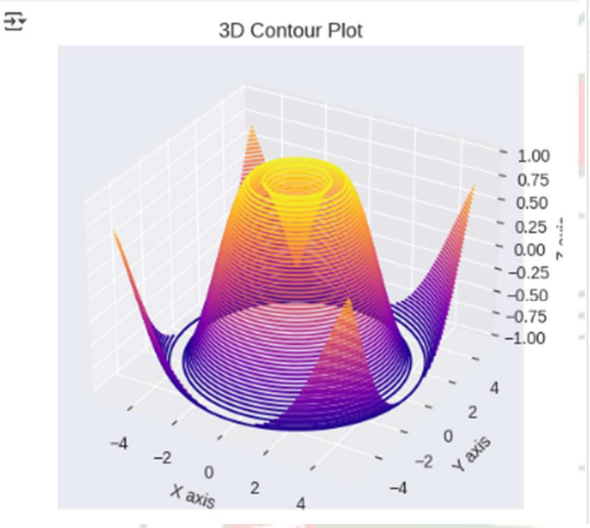
```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_title('3D Surface Plot')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=10)
plt.show()
```



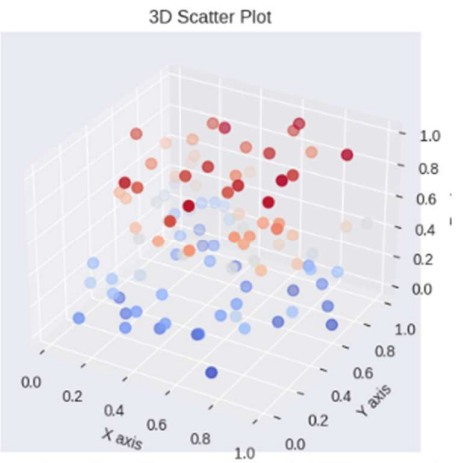

```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('3D Wireframe Plot')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```



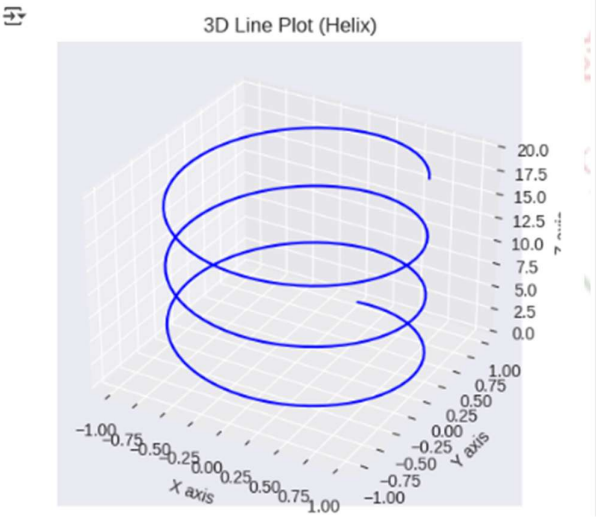
```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='plasma')
ax.set_title('3D Contour Plot')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```



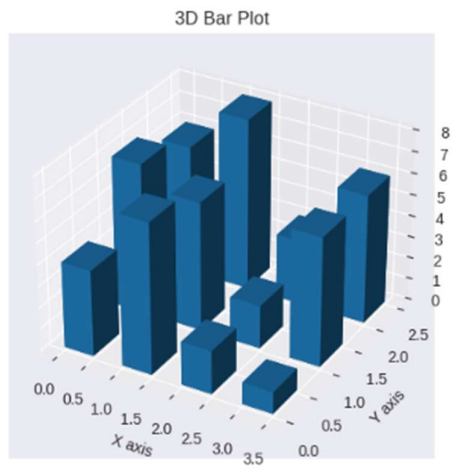

```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
x = np.random.rand(100)
y = np.random.rand(100)
z = np.random.rand(100)
ax.scatter(x, y, z, c=z, cmap='coolwarm', s=50)
ax.set_title('3D Scatter Plot')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```




```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
t = np.linspace(0, 20, 200)
x = np.sin(t)
y = np.cos(t)
z = t
ax.plot3D(x, y, z, 'blue')
ax.set_title('3D Line Plot (Helix)')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```




```
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111, projection='3d')
_x = np.arange(4)
_y = np.arange(3)
_xx, _yy = np.meshgrid(_x, _y)
x, y = _xx.ravel(), _yy.ravel()
z = np.zeros_like(x)
dx = dy = 0.5
dz = np.random.randint(1, 10, size=len(z))
ax.bar3d(x, y, z, dx, dy, dz, shade=True, cmap='cool')
ax.set_title('3D Bar Plot')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```



OUTPUT ANALYSIS:

The experiment successfully generated 3D visualizations using Matplotlib.

- **3D Line Plots** clearly showed how values change across three dimensions, providing insight into trends and relationships.
- **3D Scatter Plots** displayed individual data points in space, helping to identify clusters, patterns, or outliers.
- **3D Surface and Wireframe Plots** illustrated continuous surfaces, making it easier to visualize functions or spatial data.
- **3D Contour** Plots effectively represented the projection of surfaces on planes, highlighting levels and gradients in the data.
- **3D Bar Plots** provided a clear comparison of categorical data in three-dimensional space, making it easier to interpret grouped or stacked values.

Overall, 3D visualizations—including contour and bar plots—enhanced the understanding of multi-dimensional datasets, allowing for better interpretation of complex relationships that are not apparent in 2D plots.

CONCLUSION:

The experiment concludes that 3D visualizations using Matplotlib are an effective way to explore and interpret multi-dimensional data. By using 3D line, scatter, surface, and wireframe plots, complex relationships and patterns become more visible and intuitive. These visualizations enhance data analysis, improve insight into spatial or functional trends, and provide a powerful tool for presenting multi-dimensional information clearly.

EXPERIMENT – 10

AIM:

Create various Seaborn visualizations such as pair plots, contour plots, violin plots, and box plots to represent different aspects of your data.

DESCRIPTION:

This experiment demonstrates how to create various Seaborn visualizations to explore and represent different aspects of a dataset. Seaborn is a Python library built on Matplotlib that provides a high-level interface for creating informative and attractive statistical graphics.

- **Pair Plots:** Display pairwise relationships between numerical variables, helping to identify correlations and trends.
- **Contour Plots:** Represent three-dimensional data in two dimensions using contour lines, useful for visualizing density or function surfaces.
- **Violin Plots:** Show the distribution of a numerical variable across categories, combining a box plot and a kernel density plot for detailed insight.
- **Swarm Plots:** Display individual data points along a categorical axis, avoiding overlap, and highlighting the distribution and density of observations.
- **Box Plots:** Summarize data distribution through quartiles, highlighting median, spread, and potential outliers.

These visualizations help in understanding the underlying patterns, distributions, and relationships in the dataset, making data analysis more comprehensive and insightful.

PROGRAM:

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

df = sns.load_dataset("iris")
df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

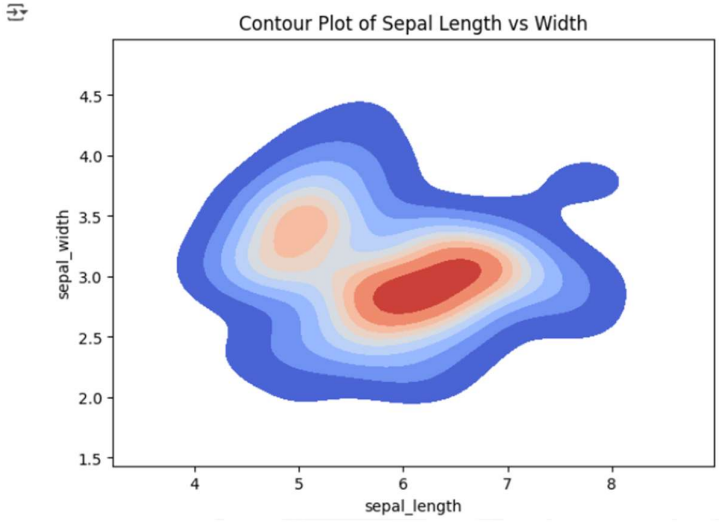
Next steps: Generate code with df | New interactive sheet

1) Pair Plot

```
sns.pairplot(df, hue="species", palette="husl")
plt.suptitle("Pair Plot - Iris Dataset", y=1.02)
plt.show()
```

2) Contour Plot

```
plt.figure(figsize=(7, 5))
sns.kdeplot(
    data=df,
    x="sepal_length",
    y="sepal_width",
    fill=True,
    cmap="coolwarm",
    levels=10,
    thresh=0.05
)
plt.title("Contour Plot of Sepal Length vs Width")
plt.show()
```



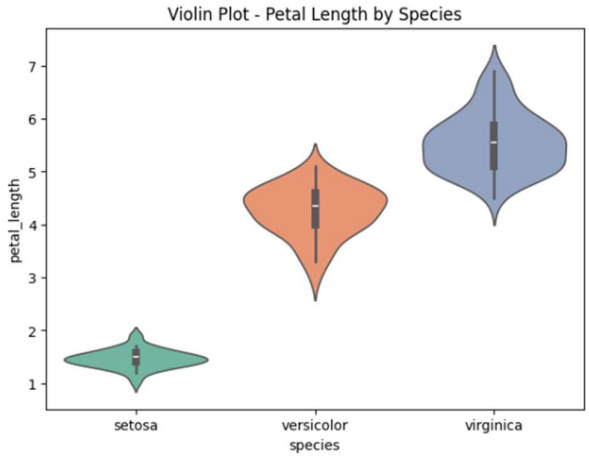
3) Violin Plot

```
plt.figure(figsize=(7, 5))
sns.violinplot(data=df, x="species", y="petal_length", palette="Set2")
plt.title("Violin Plot - Petal Length by Species")
plt.show()
```

```
/tmp/ipython-input-2964962272.py:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. As

  sns.violinplot(data=df, x="species", y="petal_length", palette="Set2")
```



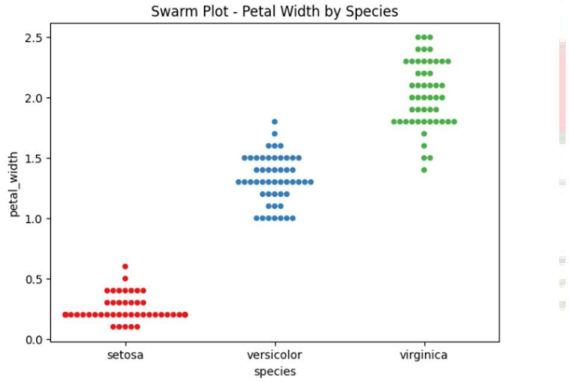
4) Swarm Plot

```
plt.figure(figsize=(7, 5))
sns.swarmplot(data=df, x="species", y="petal_width", palette="Set1")
plt.title("Swarm Plot - Petal Width by Species")
plt.show()
```

```
/tmp/ipython-input-1197807438.py:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0.

  sns.swarmplot(data=df, x="species", y="petal_width", palette="Set1")
/usr/local/lib/python3.12/dist-packages/seaborn/categorical.py:3399: UserWarning: 20.0%
  warnings.warn(msg, UserWarning)
```



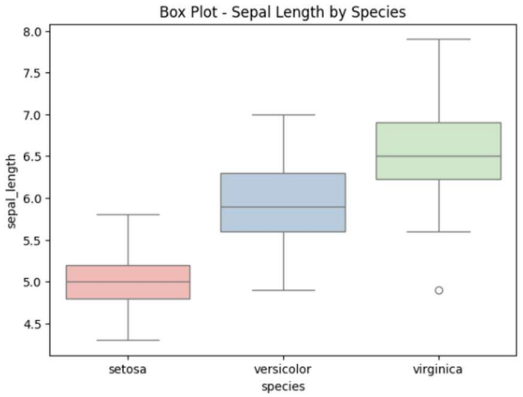
5) Box Plot

```
plt.figure(figsize=(7, 5))
sns.boxplot(data=df, x="species", y="sepal_length", palette="Pastel1")
plt.title("Box Plot - Sepal Length by Species")
plt.show()
```

```
/tmp/ipython-input-506564161.py:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Ass

  sns.boxplot(data=df, x="species", y="sepal_length", palette="Pastel1")
```



OUTPUT ANALYSIS:

The experiment successfully generated Seaborn visualizations:

- **Pair Plots:** Showed relationships between numerical variables.
- **Contour Plots:** Represented data density in 2D.
- **Violin Plots:** Displayed distribution and density across categories.
- **Swarm Plots:** Highlighted individual data points within categories.
- **Box Plots:** Summarized medians, quartiles, and outliers.

These visualizations provided clear insights into patterns, relationships, and distributions in the dataset.

CONCLUSION:

The experiment concludes that Seaborn visualizations are effective for exploring and analyzing datasets. Pair plots, contour plots, violin plots, swarm plots, and box plots help reveal relationships, distributions, and patterns in the data. These visualizations enhance understanding, support better insights, and make data analysis more intuitive and informative.