CMR
INSTITUTE OF TECHNOLOGY

USN

Internal Assessment Test I – February 2024

Sub:	Data Analytics using Python							Sub Code:	22MCA31
Date:	15/02/2024	Duration:	90 mins	Max Marks:	50	Sem:	III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	What is a dataframe in pandas?Explain with a program example. **OR**	6+4	CO2	L2
2	Explain the following: a) drop() b) del c) head() d) tail() e) Series()	2+2+2+2+2	CO2	L2
	PART II			
3	Discuss about the database concepts in python with a program example. **OR**	6+4	CO2	L3
4	Discuss about vectorized string methods in detail with a program example.	6+4	CO2	L3

	PART III			
5	Explain the attributes of numpy arrays with a program example. **OR**	6+4	CO2	L2
6	Write a note on combining and merging in pandas.Explain with a program.	6+4	CO2	L3
	PART IV			
7	How to read and write into a file using pandas?Explain with a program. **OR**	6+4	CO2	L3
8	What are universal functions? Explain with a program example.	6+4	CO2	L2
	PART V			
9	Implement a python program to demonstrate Data visualization with various types of Graphs using Numpy. **OR**	10	CO3	L4
10	Implement a python program to demonstrate the following using NumPy a) Array manipulation, Searching, Sorting and splitting. b)broadcasting and Plotting NumPy arrays	10	CO3	L4

1. Dataframes:

In the Python ecosystem, particularly for data analysis and manipulation, `pandas` is a widely-used library. One of the core components of `pandas` is the DataFrame, which is a two-dimensional labeled data structure with columns of potentially different types. It's similar to a spreadsheet or SQL table, or a dictionary of Series objects. DataFrames are great for handling structured data and performing various data operations like filtering, grouping, joining, and more.

Here's an example program demonstrating the creation and basic operations on a DataFrame:

```python
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami']
}

df = pd.DataFrame(data)

# Displaying the DataFrame
print("Original DataFrame:")
print(df)
print()

# Accessing specific columns
print("Accessing 'Name' column:")
print(df['Name'])
print()

# Adding a new column
df['Gender'] = ['Female', 'Male', 'Male', 'Male', 'Female']
print("After adding 'Gender' column:")
print(df)
print()

# Filtering data based on a condition
print("Filtering people younger than 35:")
print(df[df['Age'] < 35])
print()

# Grouping and aggregation
print("Average age by city:")
print(df.groupby('City')['Age'].mean())
```

```
This program creates a DataFrame from a dictionary, performs basic operations like accessing columns, adding a new column, filtering data based on a condition, and performing a group-by operation to find the average age by city.

This demonstrates some basic functionalities of a DataFrame in pandas, such as creating, accessing, manipulating, and analyzing data.

**2.Explaination of every method:**

here's an explanation of each term:

a) **drop()**: In pandas, the `drop()` method is used to remove rows or columns from a DataFrame. It doesn't modify the original DataFrame; instead, it returns a new DataFrame with the specified rows or columns removed. The `drop()` method takes the `labels` parameter, which specifies the index or column labels to drop, and the `axis` parameter, which specifies whether to drop rows (`axis=0`) or columns (`axis=1`). Additionally, you can specify `inplace=True` to perform the operation in-place and modify the original DataFrame.

Example:

```python
import pandas as pd

# Creating a DataFrame
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6]
}
df = pd.DataFrame(data)

# Drop column 'B'
df_drop_column = df.drop(columns=['B'])
print(df_drop_column)

# Drop row with index 1
df_drop_row = df.drop(index=1)
print(df_drop_row)
```

b) **del**: In Python, `del` is a keyword used to delete objects. In the context of pandas DataFrames, you can use `del` to delete columns. Unlike `drop()`, `del` modifies the original DataFrame directly.

Example:

```python
import pandas as pd

# Creating a DataFrame
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6]
}
df = pd.DataFrame(data)

# Delete column 'B'
del df['B']
```
```

```
print(df)
```

c) **head()**: The `head()` method in pandas is used to return the first n rows of a DataFrame. By default, it returns the first 5 rows, but you can specify the number of rows to return by passing an argument to `head()`.

Example:
```python
import pandas as pd

# Creating a DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['a', 'b', 'c', 'd', 'e']
}
df = pd.DataFrame(data)

# Return the first 3 rows
print(df.head(3))
```

d) **tail()**: Similar to `head()`, the `tail()` method returns the last n rows of a DataFrame. By default, it returns the last 5 rows, but you can specify the number of rows to return by passing an argument to `tail()`.

Example:
```python
import pandas as pd

# Creating a DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['a', 'b', 'c', 'd', 'e']
}
df = pd.DataFrame(data)

# Return the last 3 rows
print(df.tail(3))
```

e) **Series()**: In pandas, a Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, etc.). It is similar to a column in a DataFrame. You can create a Series using the `Series()` constructor, passing either a Python list, NumPy array, or a dictionary as input.

Example:
```python
import pandas as pd

# Creating a Series from a list
s1 = pd.Series([1, 2, 3, 4, 5])
print(s1)

# Creating a Series from a dictionary
```

```
data = {'a': 1, 'b': 2, 'c': 3}
s2 = pd.Series(data)
print(s2)
```

These are some common operations and methods used in pandas for data manipulation and analysis.

3. Database concepts:
In Python, databases can be interacted with using various libraries such as SQLite3, SQLAlchemy, or Django ORM. Each of these libraries offers different levels of abstraction and functionality for working with databases. Let's discuss database concepts using SQLite3 as an example.

SQLite is a lightweight, serverless, self-contained SQL database engine. Python's standard library includes the `sqlite3` module, which allows Python programs to interact with SQLite databases.

Here's a basic overview of database concepts in Python using SQLite3:

1. **Connecting to a Database**: To connect to an SQLite database, you use the `connect()` function from the `sqlite3` module. If the specified database does not exist, SQLite will create it.

2. **Creating a Table**: After connecting to the database, you can create tables using SQL `CREATE TABLE` statements executed via the `execute()` method of the database connection cursor.

3. **Inserting Data**: Data can be inserted into tables using SQL `INSERT` statements. Again, the `execute()` method is used to execute these statements.

4. **Querying Data**: You can retrieve data from the database using SQL `SELECT` statements. The results are returned as a sequence of rows, which can be iterated over or fetched using methods like `fetchone()` or `fetchall()`.

5. **Updating and Deleting Data**: Data can be updated or deleted using SQL `UPDATE` and `DELETE` statements, respectively.

Here's an example program demonstrating these concepts:

```python
import sqlite3

# Connect to the SQLite database (creates it if it doesn't exist)
conn = sqlite3.connect('example.db')

# Create a cursor object to execute SQL commands
cursor = conn.cursor()

# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users
                  (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')

# Insert data into the table
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Alice', 30))
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Bob', 25))
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Charlie', 35))
```

```
# Commit changes to the database
conn.commit()

# Query data from the table
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
print("All Users:")
for row in rows:
    print(row)

# Update data in the table
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (28, 'Bob'))
conn.commit()

# Delete data from the table
cursor.execute("DELETE FROM users WHERE name = ?", ('Charlie',))
conn.commit()

# Query data again to see the changes
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
print("\nUsers after update and delete:")
for row in rows:
    print(row)

# Close the cursor and the connection
cursor.close()
conn.close()
```

1. **Create Table**:
   - Creating a table in a database involves defining its structure, including column names, data types, constraints, and indexes. This is typically done using SQL `CREATE TABLE` statements executed via the cursor object.
   - Example:
   ```python
   cursor = conn.cursor()
   cursor.execute('''CREATE TABLE IF NOT EXISTS users
                     (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')
   ```

2. **Insert**:
   - The `INSERT` command is used to add new records (rows) to a table in the database. It specifies the table name and the values to be inserted into each column.
   - Example:
   ```python
   cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Alice', 30))
   ```

3. **Select**:
   - The `SELECT` command is used to retrieve data (rows) from one or more tables in the database. It allows you to specify which columns to retrieve and can include various filtering and sorting options.

- Example:
```python
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

4. **Update**:
   - The `UPDATE` command is used to modify existing records in a table. It allows you to change the values of one or more columns in one or more rows based on specified conditions.
   - Example:
   ```python
   cursor.execute("UPDATE users SET age = ? WHERE name = ?", (28, 'Bob'))
   ```

5. **Delete**:
   - The `DELETE` command is used to remove one or more records from a table. It allows you to specify conditions to identify the records to be deleted.
   - Example:
   ```python
   cursor.execute("DELETE FROM users WHERE name = ?", ('Charlie',))
   ```

These commands form the foundation of interacting with databases in Python, allowing you to create, retrieve, update, and delete data as needed. Always remember to commit your changes (`conn.commit()`) after modifying data and close the connection (`conn.close()`) when finished to release resources properly.

4. Vectorized string functions:

Vectorized string methods in pandas are powerful tools for efficiently performing string operations on entire arrays of data without needing to write explicit loops. These methods are applied element-wise to each string in a Series or DataFrame column, making string manipulation tasks easier and more efficient. Here are 10 commonly used vectorized string methods in pandas along with examples:

1. **str.lower() / str.upper()**:
   - Convert strings to lowercase or uppercase.
   ```python
   import pandas as pd

   data = {'Name': ['Alice', 'Bob', 'Charlie', 'David']}
   df = pd.DataFrame(data)

   # Convert names to lowercase
   df['Name_lower'] = df['Name'].str.lower()
   print(df)
   ```

2. **str.len()**:
   - Compute the length of each string.
   ```python
   # Compute length of names
   ```

```
df['Name_length'] = df['Name'].str.len()
print(df)
```

3. **str.replace()**:
   - Replace occurrences of a pattern with another string.
   ```python
   # Replace 'a' with 'x'
   df['Name_replace'] = df['Name'].str.replace('a', 'x')
   print(df)
   ```

4. **str.contains()**:
   - Check if a substring or regular expression pattern is present in each string.
   ```python
   # Check if 'li' is present in names
   df['Name_contains_li'] = df['Name'].str.contains('li')
   print(df)
   ```

5. **str.split()**:
   - Split each string by a delimiter into a list.
   ```python
   # Split names by whitespace
   df['Name_split'] = df['Name'].str.split()
   print(df)
   ```

6. **str.join()**:
   - Join lists of strings into a single string using a delimiter.
   ```python
   # Join split names with a comma
   df['Name_join'] = df['Name_split'].str.join(', ')
   print(df)
   ```

7. **str.startswith() / str.endswith()**:
   - Check if each string starts or ends with a specified prefix or suffix.
   ```python
   # Check if names start with 'A'
   df['Name_starts_with_A'] = df['Name'].str.startswith('A')

   # Check if names end with 'e'
   df['Name_ends_with_e'] = df['Name'].str.endswith('e')
   print(df)
   ```

8. **str.extract()**:
   - Extract substrings matching a regular expression pattern.
   ```python
   # Extract the first letter of each name
   df['First_letter'] = df['Name'].str.extract(r'(\w)')
```

```
print(df)
```

9. **str.isnumeric() / str.isalpha() / str.isalnum()**:
   - Check if each string contains only numeric characters, alphabetic characters, or alphanumeric characters.
   ```python
   # Check if names are alphanumeric
   df['Name_is_alphanumeric'] = df['Name'].str.isalnum()
   print(df)
   ```

10. **str.strip() / str.lstrip() / str.rstrip()**:
    - Remove leading and trailing whitespace characters.
    ```python
    data = {'Name': ['  Alice ', ' Bob', 'Charlie  ', 'David  ']}
    df = pd.DataFrame(data)

    # Strip whitespace from names
    df['Name_stripped'] = df['Name'].str.strip()
    print(df)
    ```

5. Attributes of numpy arrays:

Sure! NumPy arrays are the fundamental data structure used for numerical computations in Python. They provide efficient storage and manipulation of homogeneous data, such as numbers. Here are some common attributes of NumPy arrays along with a program example:

1. **shape**: Returns a tuple representing the dimensions of the array.
2. **dtype**: Returns the data type of the elements in the array.
3. **ndim**: Returns the number of dimensions (axes) of the array.
4. **size**: Returns the total number of elements in the array.
5. **itemsize**: Returns the size in bytes of each element in the array.
6. **nbytes**: Returns the total size in bytes of the array.

Let's demonstrate these attributes with an example:

```python
import numpy as np

# Create a NumPy array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Print the array
print("Array:")
print(arr)
print()

# Print attributes
print("Shape:", arr.shape)
print("Data type:", arr.dtype)
print("Number of dimensions:", arr.ndim)
print("Total number of elements:", arr.size)
```

```
print("Size of each element (in bytes):", arr.itemsize)
print("Total size of the array (in bytes):", arr.nbytes)
```

6. Combining and merging:

Combining and merging in pandas are two essential operations for data manipulation and analysis, especially when working with multiple datasets. These operations allow you to combine data from different sources based on common columns or indices.

1. **Combining**:
   Combining involves concatenating or stacking data along an axis (row or column). It is useful when you want to add new rows or columns to an existing DataFrame.

2. **Merging**:
   Merging involves combining data from different DataFrames based on common columns or indices. It is similar to SQL join operations and allows you to merge datasets horizontally.

Let's demonstrate combining and merging with a program example:

```python
import pandas as pd

# Creating two DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3],
                    'Name': ['Alice', 'Bob', 'Charlie']})

df2 = pd.DataFrame({'ID': [4, 5, 6],
                    'Name': ['David', 'Eva', 'Frank']})

# Combining DataFrames along rows (concatenating)
combined_df = pd.concat([df1, df2], ignore_index=True)
print("Combined DataFrame (concatenation):")
print(combined_df)
print()

# Creating another DataFrame for merging
df3 = pd.DataFrame({'ID': [1, 2, 3, 4],
                    'Age': [25, 30, 35, 40]})

# Merging DataFrames based on the 'ID' column
merged_df = pd.merge(combined_df, df3, on='ID')
print("Merged DataFrame:")
print(merged_df)
```

7. Read and write into a file using pandas

In pandas, you can read and write data to/from files using the `read_csv()`, `to_csv()`, `read_excel()`, and `to_excel()` functions for CSV and Excel files, respectively. These functions provide easy-to-use methods for importing and exporting data between pandas DataFrames and files.

Here's a program example demonstrating how to read data from a CSV file into a DataFrame and then write that DataFrame back to another CSV file:

```python
```

```
import pandas as pd

# Read data from CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the DataFrame
print("DataFrame from CSV:")
print(df)
print()

# Write the DataFrame to a new CSV file
df.to_csv('output.csv', index=False)

print("DataFrame written to output.csv")
```

In this example:
- We first use `pd.read_csv('data.csv')` to read data from a CSV file named 'data.csv' into a pandas DataFrame.
- We then display the DataFrame to verify that the data has been read correctly.
- Next, we use `df.to_csv('output.csv', index=False)` to write the DataFrame to a new CSV file named 'output.csv'. The `index=False` parameter is used to exclude the DataFrame index from being written to the file.

Similarly, you can use `read_excel()` and `to_excel()` functions for Excel files:

```python
# Read data from Excel file into a DataFrame
df_excel = pd.read_excel('data.xlsx')

# Display the DataFrame
print("DataFrame from Excel:")
print(df_excel)
print()

# Write the DataFrame to a new Excel file
df_excel.to_excel('output.xlsx', index=False)

print("DataFrame written to output.xlsx")
```

8. Universal functions:

Universal functions (ufuncs) in NumPy are functions that operate element-wise on arrays. They are vectorized functions, meaning they can perform element-wise operations on arrays without the need for explicit looping, which results in faster computation compared to traditional looping over array elements. Ufuncs can perform arithmetic operations, trigonometric functions, exponential functions, comparison operations, etc., on NumPy arrays.

Here's a program example demonstrating the usage of ufuncs:

```python
import numpy as np

# Creating NumPy arrays
arr1 = np.array([1, 2, 3, 4, 5])
```

```
arr2 = np.array([6, 7, 8, 9, 10])

# Arithmetic operations using ufuncs
print("Addition:")
print(np.add(arr1, arr2))
print()

print("Subtraction:")
print(np.subtract(arr1, arr2))
print()

print("Multiplication:")
print(np.multiply(arr1, arr2))
print()

print("Division:")
print(np.divide(arr1, arr2))
print()

# Trigonometric functions using ufuncs
theta = np.linspace(0, np.pi, 5)
print("Sine:")
print(np.sin(theta))
print()

print("Cosine:")
print(np.cos(theta))
print()

# Exponential functions using ufuncs
print("Exponential:")
print(np.exp(arr1))
print()

# Comparison operations using ufuncs
print("Greater than:")
print(np.greater(arr1, arr2))
print()

print("Less than:")
print(np.less(arr1, arr2))
print()
```

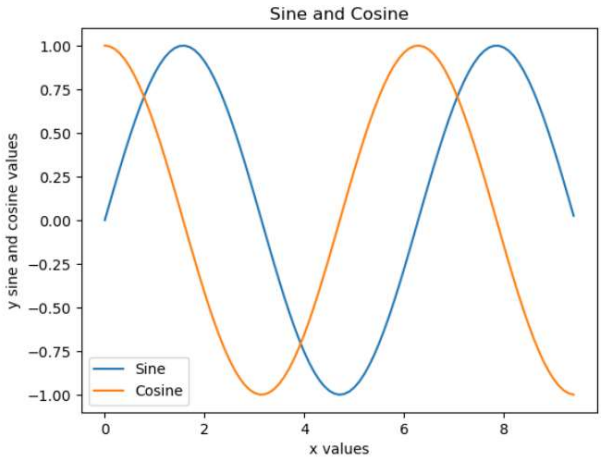
9.Implement a python program to demonstrate Data visualization with various types of Graphs using Numpy.

#plotting

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

x=np.arange(0,3*np.pi,0.1)
print("x=",x)

y_sin=np.sin(x)
y_cos=np.cos(x)

plt.plot(x,y_sin)
plt.plot(x,y_cos)
plt.xlabel('x values')
plt.ylabel('y sine and cosine values')
plt.title('Sine and Cosine')
plt.legend(['Sine','Cosine'])

plt.show()
```

Output:

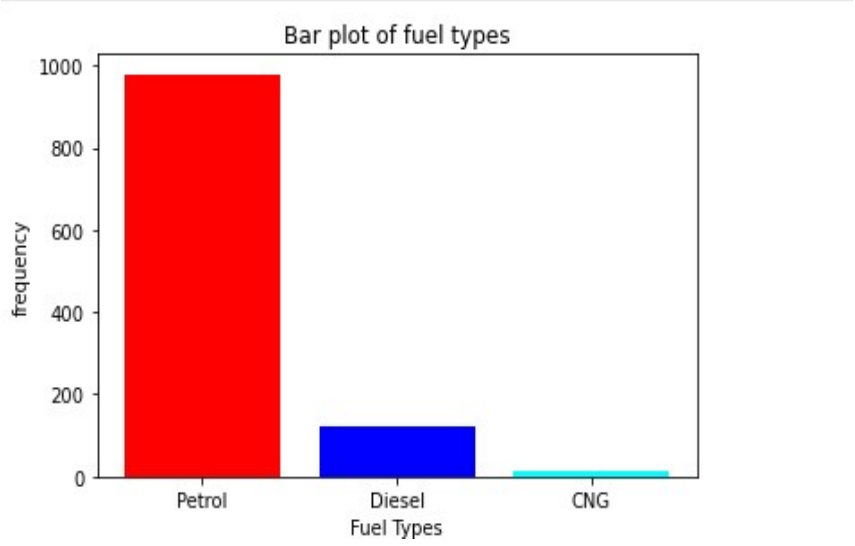
```
x= [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.  5.1 5.2 5.3
 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7.  7.1
 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.  8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9
 9.  9.1 9.2 9.3 9.4]
```



Bar Graph

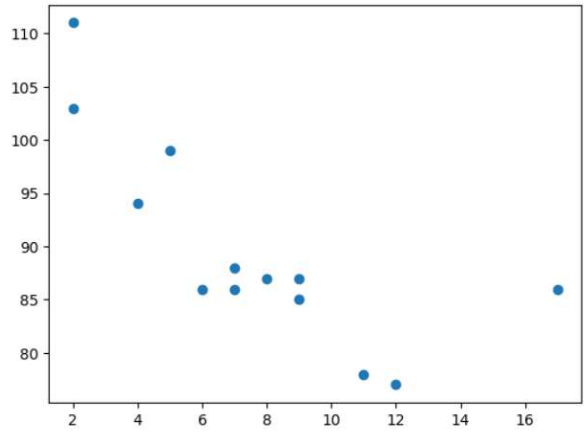
##Bar Plot(for categorical variables)

```
import numpy as np
import matplotlib.pyplot as plt
counts=[979,120,12]
fuelType=('Petrol','Diesel','CNG')
index=np.arange(len(fuelType))

plt.bar(index,counts,color=['red','blue','cyan'])
plt.title('Bar plot of fuel types')
plt.xlabel('Fuel Types')
plt.ylabel('frequency')
plt.xticks(index,fuelType,rotation=0)
Plt.show()
```



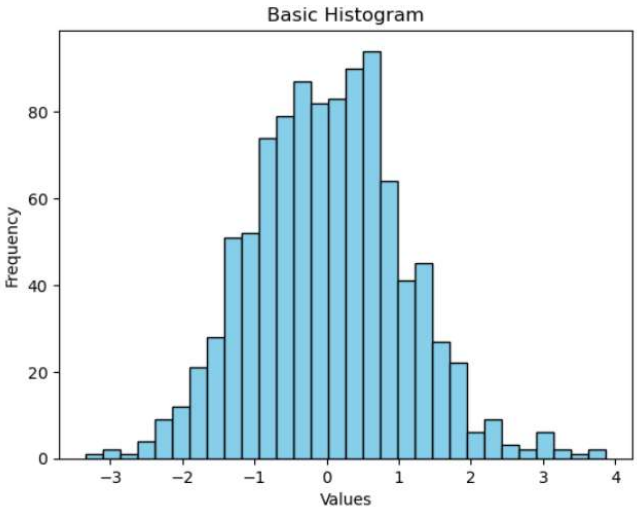
Scatter plot

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()
```



Histogram

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

# Generate random data for the histogram
data = np.random.randn(1000)
#print(data)
# Plotting a basic histogram
plt.hist(data, bins=30, color='skyblue', edgecolor='black')

# Adding labels and title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Basic Histogram')

# Display the plot
plt.show()
```



10. Implement a python program to demonstrate the following using NumPy

a)Array manipulation, Searching, Sorting and splitting.

b)broadcasting and Plotting NumPy arrays

Array manipulation, Searching,Sorting and splitting

```
#Array manipulation
#importing numpy package

import numpy as np
#Concatinatin of arrays
a=np.array([[1,2],[3,4]])
b=np.array([[5,6]])
print("concate with axis=0:",np.concatenate((a,b),axis=0))
print("concate with axis=1:",np.concatenate((a,b.T),axis=1))

print(np.concatenate((a,b),axis=None))

a=np.array([[12,4,5],[23,45,66],[45,34,23]])
b=np.array([[1,40,50],[2,4,6],[4,3,2]])

#Verticle stacking
print(np.vstack((a,b)))

#Horizontal stacking
print(np.hstack((a,b)))

print(a.reshape(3,3))
```

##Sorting sort :Return a sorted copy of an array.

#ndarray.sort : Method to sort an array in-place.

#argsort: Indirect sort. Returns the indices that would sort an array.

#numpy.sort_complex: Sort a complex array using the real part first, then the imaginary part.

```
concate with axis=0: [[1 2]
 [3 4]
 [5 6]]
concate with axis=1: [[1 2 5]
 [3 4 6]]
[1 2 3 4 5 6]
[[12  4  5]
 [23 45 66]
 [45 34 23]
 [ 1 40 50]
 [ 2  4  6]
 [ 4  3  2]]
[[12  4  5  1 40 50]
 [23 45 66  2  4  6]
 [45 34 23  4  3  2]]
[[12  4  5]
 [23 45 66]
 [45 34 23]]
```

```
#Sorting
#importing numpy package
import numpy as np
a=np.array([[1,4],[3,1]])
print(a)
print("Sorted array:\n",np.sort(a))
print("\n sorted flattened array:\n",np.sort(a,axis=0))

x=np.array([3,1,2])
print("\n indices that would sort an array",np.argsort(x))
print("\n Sorting complex number:",np.sort_complex([[3 + 4j, 1 - 2j,
                                        5 + 1j, 2 + 2j]]))
```

```
[[1 4]
 [3 1]]
Sorted array:
 [[1 4]
 [1 3]]

 sorted flattened array:
 [[1 1]
 [3 4]]

 indices that would sort an array [1 2 0]

 Sorting complex number: [[1.-2.j 2.+2.j 3.+4.j 5.+1.j]]
```

```
#Searching
import numpy as np
arr=np.array([1,2,3,4,5,4,4])
x=np.where(arr==4)
print(x)

arr=np.array([6,7,8,9])
x=np.searchsorted(arr,5)
print(x)

arr=np.array([1,3,5,7])
x=np.searchsorted(arr,[2,4,6])
print(x)
```

```
(array([3, 5, 6], dtype=int64),)
0
[1 2 3]
```

```
#Splitting
import numpy as np
x=np.arange(9.0)
print(x)
print(np.split(x,3))
```

```
print(np.split(x,[3,5,6,10]))

x=np.arange(9)
print(np.array_split(x,4))

a=np.array([[1,3,5,7,9,11],
        [2,4,6,8,10,12]])
print("Splitting along horizontal axis into 2 parts:\n",np.hsplit(a,2))
print("\n Splitting along vertical axis into 2 parts:\n",np.vsplit(a,2))
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8.]
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]
[array([0., 1., 2.]), array([3., 4.]), array([5.]), array([6., 7., 8.]), array([], dtype=float64)]
[array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]
Splitting along horizontal axis into 2 parts:
 [array([[1, 3, 5],
       [2, 4, 6]]), array([[ 7,  9, 11],
       [ 8, 10, 12]])]

 Splitting along vertical axis into 2 parts:
 [array([[ 1,  3,  5,  7,  9, 11]]), array([[ 2,  4,  6,  8, 10, 12]])]
```

 # 5 b) ##b) Broadcasting and Plotting NumPy arrays

 # 5b)Broadcasting and plotting numpy arrays

```
import numpy as np
x=np.arange(4)
print(x)
y=np.ones(5)
print(y)
xx=x.reshape(2,2)
print(xx)
z=np.ones((3,4))
print(z)
print(x.shape)

a=np.array([0.0,10.0,20.0,30.0])
b=np.array([1.0,2.0,3.0])
a[:,np.newaxis]+b
```

```
[0 1 2 3]
[1. 1. 1. 1. 1.]
[[0 1]
 [2 3]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
(4,)
```

```
Out[19]: array([[ 1.,  2.,  3.],
                [11., 12., 13.],
                [21., 22., 23.],
                [31., 32., 33.]])
```

#Plotting a two-dimensional function – Broadcasting is also frequently used in displaying images based on two- dimensional functions. If we want to define a function z=f(x, y).

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

x=np.arange(0,3*np.pi,0.1)
print("x=",x)

y_sin=np.sin(x)
y_cos=np.cos(x)

plt.plot(x,y_sin)
plt.plot(x,y_cos)
plt.xlabel('x values')
plt.ylabel('y sine and cosine values')
plt.title('Sine and Cosine')
plt.legend(['Sine','Cosine'])

plt.show()
```

Output :

```
x= [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.  5.1 5.2 5.3
 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7.  7.1
 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.  8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9
 9.  9.1 9.2 9.3 9.4]
```