Sr No	Practical Name	Date	Page No.	Sign
1.	Programs based on Data Types	11/07/2025	1	
2.	Programs based on Functions.	18/07/2025	28	
3.	Programs based on File Handling.	25/07/2025	32	
4.	Programs based on Packages.	01/08/2025	37	
5.	Programs based on Control Structures.	08/08/2025	41	
6.	Programs based on exception handling.	22/08/2025	46	
7.	Programs based on Inheritance.	29/08/2025	50	
8.	Programs based on Overloading.	12/09/2025	54	
9.	Working on Big Data libraries:NumPy,Pandas,Matplotlib etc.	19/09/2025	60	

Practical No 1

Program based on Data Types

1. Numerical Data Type

A. Integer (int)

Integers are whole numbers, positive or negative, without any decimal point.

Example:

```
# Integer examples
a = 10
b = -7

print(type(a)) # Output: <class 'int'>
 print(type(b)) # Output: <class 'int'>
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/p
ac 1.py
<class 'int'>
<class 'int'>
```

B. Float (float)

Floats represent real numbers with a decimal point.

Example:

```
# Float examples
 x = 10.5
y = -2.16

print(type(x)) # Output: <class 'float'>
 print(type(y)) # Output: <class 'float'>
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
<class 'float'>
<class 'float'>
```

C. Complex (complex)

Complex numbers have a real part and an imaginary part. They are written in the form a + bj, where a is the real part and b is the imaginary part.

Example:

```
# Complex number examples
z1 = 4 + 5j
z2 = -2 - 4j
print(type(z1)) # Output: <class 'complex'>
print(type(z2)) # Output: <class 'complex'>
# Accessing real and imaginary parts
print(z1.real)
print(z1.imag)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
<class 'complex'>
<class 'complex'>
4.0
5.0
```

Operations on Numeric Data Types

Python allows basic arithmetic operations between numeric types:

```
# Addition
result = 5 + 3.2 # int + float
 print(result) # Output: 8.2

# Subtraction
 result = 10 - 2
print(result) # Output: 8

# Multiplication
result = 4 * 2.5 # int * float
 print(result) # Output: 10.0

# Division
result = 9 / 2 # Division always returns float
 print(result) # Output: 4.5
# Integer Division
result = 9 // 2 # Integer division (floor)
 print(result) # Output: 4

# Modulus
result = 9 % 2 # Remainder
 print(result) # Output: 1

# Exponentiation
result = 2 ** 3 # 2 raised to the power 3
 print(result) # Output: 8
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
8.2
8.2
10.0
4.5
4
1
8
```

Type Conversion

Python allows conversion between numeric types using built-in functions:

- int(): Converts a float or string to an integer.
- float(): Converts an integer or string to a float.
- complex(): Converts numbers to complex numbers.

Example:

```
# Type conversion examples

a = 10.75

b = int(a) # Float to int

 print(b) # Output: 10

c = 5

d = float(c) # Int to float

 print(d) # Output: 5.0

e = 2

f = complex(e) # Int to complex

 print(f) # Output: (2+0j)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
10
5.0
(2+0j)
```

2. String Data Type

A. **Creating strings:** Defining a sequence of characters enclosed in quotes (single, double, or triple) and assigning it to a variable.

```
string1 = "Hello, World!"
string2 = "Python Programming"
 print(string1)
print(string2)
```

Output:

```
Type "help", "copyright", "credits" or "license()"
>>
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/P
gs.py
Hello, World!
Python Programming
>>
```

B. **Accessing characters in a string:** Retrieving individual characters using their index positions, where indexing starts at 0 for the first character and -1 for the last.

```
string1 = "Hello, World!"
string2 = "Python Programming" print("First
 character of string1:", string1[0])
print("Last character of string2:", string2[-1])
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
First character of string1: H
Last character of string2: g
```

C. String length: Total number of characters in a string, including spaces and punctuation, and is measured using the len() function

.

```
string1 = "Hello, World!"
string2 = "Python Programming"
print("Length of string1:", len(string1))
print("Length of string2:", len(string2))
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Length of string1: 13
Length of string2: 18
```

C . String slicing: Extracting a portion of a string by specifying a range of index positions using the syntax [start:end].

```
string1 = "Hello, World!"
string2 = "Python Programming"
substring = string1[7:12] # Extracting 'World'
print("Substring of string1:", substring)
```

Output:

```
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep
AMD64)] on win32
Type "help", "copyright", "credits" or "li
>>
= RESTART: C:/Users/ADMIN/AppData/Local/Pr
Substring of string1: World
>>
```

C. **String concatenation:**Joining two or more strings end-to-end using the + operator.

```
string1 = "Hello, World!"
string2 = "Python Programming"
 combined_string = string1 + " " + string2
 print("Combined string:", combined_string)
```

Output:

```
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Combined string: Hello, World! Python Programming
```

D. String Methods: upper(), lower(), count(), replace(), and split().

1. upper(): Converts all characters in a string to uppercase.
2. lower(): Converts all characters in a string to lowercase.
3. count(): Returns the number of occurrences of a specified substring in the string.

4.replace(old, new): Replaces all occurrences of a specified substring (old) with another substring (new).

```
string1 = "Hello, World!"
string2 = "Python Programming"
 print("Uppercase string1:", string1.upper())
 print("Lowercase string2:", string2.lower())
 print("Count of 'o' in string1:", string1.count('o'))
```

Output:

```
>
= RESTART: C:/Users/ADMIN/AppData/Local/Pro
Uppercase string1: HELLO, WORLD!
Lowercase string2: python programming
Count of 'o' in string1: 2
>
```

E. **Replacing parts of a string:** Refers to using the replace() method to substitute all occurrences of a specified substring with another substring.

```
string1 = "Hello, World!"
string2 = "Python Programming"
new_string = string1.replace("World", "Python")
 print("After replacement:", new_string)
```

Output:

```
AMD64)] on win32
Type "help", "copyright", "credits" or "licens

= RESTART: C:/Users/ADMIN/AppData/Local/Progra
After replacement: Hello, Python!
```

F. **Splitting a string:** Involves using the split() method to divide a string into a list of substrings based on a specified delimiter (default is whitespace).

```
string1 = "Hello, World!"
string2 = "Python Programming"
 words = string2.split()
```

```
print("Words in string2:", words)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Words in string2: ['Python', 'Programming']
```

G. **Checking if a substring exists:** Refers to using the in keyword to determine whether a specified substring is present within a given string.

```
string1 = "Hello, World!"
string2 = "Python Programming"
 if "Python" in string2:
print("Substring 'Python' found in string2!")
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Substring 'Python' found in string2!
```

H. **Formatting strings:** Involves creating a new string that includes variable values using placeholders, often done with f-strings (formatted string literals) in Python, denoted by the f prefix before the string.

```
name = "Prem"
```

```
age = 30
formatted_string = f"My name is {name} and I am {age} years old."
 print("Formatted string:", formatted_string)
```

Output:

```
Type "help", "copyright", "credits" or "license()" for more infor
>>>
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312
Formatted string: My name is Prem and I am 30 years old.
>>>
```

I. **Iterating through a string:**Refers to the process of going through each character in the string one by one, typically using a loop, to perform actions or access individual characters.

```
string1 = "Hello, World!"
string2 = "Python Programming"
 print("Characters in string1:")
 for char in string1:
print(char, end=' ')
print() # New line for better output
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local
Characters in string1:
H e l l o ,   W o r l d !
>>>
```

J. **Reversing a string:**Involves creating a new string that contains the characters of the original string in reverse order, often achieved using slicing with the syntax string[::-1].

```
reversed_string = string1[::-1]
 print("Reversed string1:", reversed_string)
```

Output:

```
>>
= RESTART: C:/Users/ADMIN/AppData/Loca
Reversed string1: !dlroW ,olleH
>>
```

3. List Data type

In Python, a list is a mutable, ordered collection of items (elements) that can store elements of different data types. Lists are defined using square brackets [], and elements are separated by commas.

Key characteristics of Python lists:

1. Ordered: Items in a list maintain their order.
2. Mutable: Items can be modified (added, removed, changed).
3. Allow Duplicates: Lists can contain duplicate values.
4. Mixed Data Types: Lists can hold items of different types (integers, strings, other lists, etc.).

A. List of Integers:

```
integers_list = [6 ,7 ,8 ,9 ,10]
print(integers_list)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
[6, 7, 8, 9, 10]
```

B. List of Strings:

```
string_list = ["apple", "banana", "grape"]
print(string_list)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/p
ac 1.py
['apple', 'banana', 'grape']
```

C. List of Mixed Data Types:

```
mixed_list = [1, "apple", 3.14, True]
print(mixed_list)
```

Output:

```
>
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
[1, 'apple', 3.14, True]
>
```

D. Nested List (List inside a list):

```
nested_list = [1, [2, 3], ["apple", "banana"]]
print(nested_list)
```

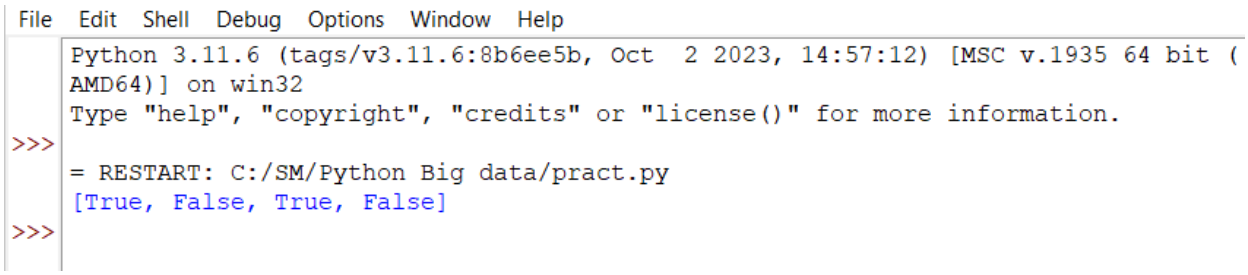
Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
[1, [2, 3], ['apple', 'banana']]
```

E. List of Boolean Values:

```
boolean_list = [True, False, True, False]
print(boolean_list)
```

Output:

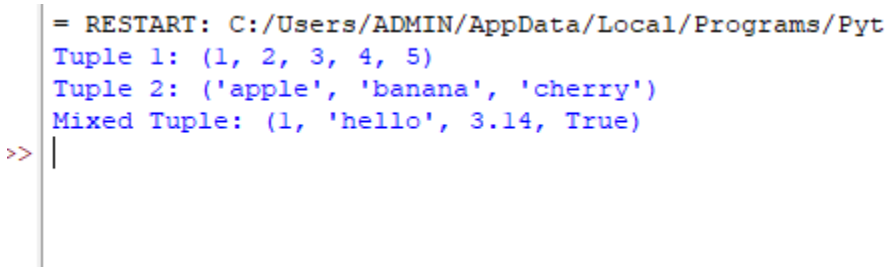


4. Tuple Data Type

A. **Creating tuples and printing the original tuples:** Demonstrates how to create and print tuples.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
mixed_tuple = (1, 'hello', 3.14, True)

# Printing the original tuples
print("Tuple 1:", tuple1)
print("Tuple 2:", tuple2) print("Mixed
Tuple:", mixed_tuple)
```

Output:



B. **Accessing elements in a tuple:** Shows how to access specific elements using indexing.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
```

```
mixed_tuple = (1, 'hello', 3.14, True)
 print("First element of tuple1:", tuple1[0])
 print("Last element of tuple2:", tuple2[-1])
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local
First element of tuple1: 1
Last element of tuple2: cherry
>
```

C. **Tuple length:** Uses len() to get the length of a tuple.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 print("Length of tuple1:", len(tuple1))
 print("Length of tuple2:", len(tuple2))
```

Output:

```
Length of tuple1: 5
Length of tuple2: 3
>>>
```

D. **Slicing a tuple:** Extracts a subset of a tuple.

```
tuple1 = (6 ,7 ,8 ,9 ,10)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
sub_tuple = tuple1[1:4] # Extracting (2, 3, 4)
 print("Sliced tuple from tuple1:", sub_tuple)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Sliced tuple from tuple1: (7, 8, 9)
```

E. **Concatenating tuples:**Combines two tuples using the + operator.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 combined_tuple = tuple1 + tuple2
 print("Combined tuple:", combined_tuple)
```

Output:

```
>
 = RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312/
 Combined tuple: (1, 2, 3, 4, 5, 'apple', 'banana', 'cherry')
>
```

F. **Repeating tuples:**Repeats a tuple using the * operator.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 repeated_tuple = tuple2 * 2
 print("Repeated tuple2:", repeated_tuple)
```

```
 = RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312/accessing.p
 Repeated tuple2: ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
>>>
```

G. Repeating tuples: Repeats a tuple using the * operator.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 repeated_tuple = tuple2 * 2
 print("Repeated tuple2:", repeated_tuple)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Repeated tuple2: ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

G. **Checking membership:**Checks if an element is in the tuple.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
```

```
if 'banana' in tuple2:
    print("'banana' is in tuple2!")
```

Output:

```
>>>
= RESTART: C:/Users/ADMIN/A
'banana' is in tuple2!
>>>
```

H. **Iterating through a tuple:** Loops through each item in a tuple.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 print("Elements in mixed_tuple:") for
 item in mixed_tuple:
print(item, end=' ')
print() # New line for better output
```

Output:

```
>
= RESTART: C:/Users/ADMIN/AppData/L
Elements in mixed_tuple:
1 hello 3.14 True
>
```

I. **Tuple unpacking:** Assigns tuple values to individual variables.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
a, b, c, d = mixed_tuple # Unpacking the mixed_tuple
 print("Unpacked values:", a, b, c, d)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local/
Unpacked values: 1 hello 3.14 True
```

J. **Nested tuples:**Demonstrates how to create a tuple containing other tuples.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 nested_tuple = (tuple1, tuple2)
 print("Nested Tuple:", nested_tuple)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312
Nested Tuple: ((1, 2, 3, 4, 5), ('apple', 'banana', 'cherry'))
>
```

K. **Converting a tuple to a list:** Converts a tuple to a list.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 tuple_to_list = list(tuple1)
print("Converted tuple1 to list:", tuple_to_list)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/Pyt
Converted tuple1 to list: [1, 2, 3, 4, 5]
>>>
```

L. **Creating a tuple with a single element:**Shows how to create a tuple with a single element (note the comma).

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
single_element_tuple = (42,)
print("Single element tuple:", single_element_tuple)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppD
Single element tuple: (42,)
>>>
```

M. **Counting occurrences:** Uses count() to find occurrences and positions of elements.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 print("Count of 2 in tuple1:", tuple1.count(2))
```

Output:

```
= RESTART: C:/Users/ADMIN/Appl
Count of 2 in tuple1: 1
```

N. **Finding index of an element:** Uses index() to find occurrences and positions of elements.

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ('apple', 'banana', 'cherry')
 mixed_tuple = (1, 'hello', 3.14, True)
 print("Index of 3 in tuple1:", tuple1.index(3))
```

Output:

```
= RESTART: C:/Users/ADMIN/App
Index of 3 in tuple1: 2
```

5. Python Dictionary

In Python, a dictionary is a collection of key-value pairs, where each key is unique, and each key maps to a value. Dictionaries are mutable, meaning you can change, add, or remove key-value pairs.

A. Creating a Dictionary

You can create a dictionary by placing key-value pairs inside curly braces {}, separated by colons, or by using the dict() constructor.

Example:

```
# Creating a dictionary
 person = {
"name": "Prasanna ",
    "age": 20,
"city": "New York"
}

# Using dict() constructor
person2 = dict(name="Bob", age=30, city="Los Angeles")
```

B. Accessing Dictionary Items

You can access the value of a specific key using square brackets [] or the .get() method.

Example:

```
# Accessing the value by key
 print(person["name"])

# Using the .get() method
 print(person.get("age"))
```

Output:

```
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Prasanna
20
```

C. Modifying

Dictionary Example:

```
# Modifying an existing key
 person["age"] = 23

# Adding a new key-value pair
 person["job"] = "Engineer"

print(person)
```

Output :

```
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Prasanna
20
{'name': 'Prasanna ', 'age': 23, 'city': 'New York', 'job': 'Engineer'}
>
```

D. Deleting Items

You can remove a key-value pair using the del keyword or the .pop() method.

Example:

```
# Removing a key-value pair

del person["city"]

# Using pop()

job = person.pop("job")

print(job) # Output: Engineer

print(person)
```

Output:

```
>
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
Prasanna
20
{'name': 'Prasanna ', 'age': 23, 'city': 'New York', 'job': 'Engineer'}
Engineer
>
```

6. Python Set

In Python, a set is an unordered collection of unique elements. Sets are mutable, meaning you can add and remove items, but the items themselves must be immutable (e.g., numbers, strings, tuples).

A) Creating a Set

You can create a set by placing elements inside curly braces {} or using the set()

Example:

```
# Creating a set
fruits = {"apple", "banana", "cherry"}

# Using set() constructor (from a list)
 numbers = set([1, 2, 3, 4, 5])

# Creating an empty set
 empty_set = set()
```

B) Characteristics of Sets:

- Unordered: The elements have no index, meaning you can't access set elements by position.
- Unique elements: Duplicate elements are automatically removed.

Example:

```
# Duplicate elements are removed
fruits = {"apple", "banana", "cherry", "apple"}
 print(fruits)
```

Output:

```
Python 3.11.6 (tags/v3.11.6:8b6ee5b, Oct  2 2023, 14:57:12) [MSC v.1935 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/SM/Python Big data/pract.py
{'cherry', 'banana', 'apple'}
>>>
```

C) Adding Elements to a Set:

You can add a single element using the .add() method or multiple elements using the .update() method.

Example:

```
# Adding a single element
fruits.add("orange")
print(fruits)

# Adding multiple elements
fruits.update(["mango", "grape"])
print(fruits)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
{'banana', 'apple', 'orange', 'cherry'}
{'grape', 'banana', 'mango', 'cherry', 'orange', 'apple'}
```

D) Removing Elements from a Set:

You can remove elements using .remove(), .discard(), or .pop().

- .remove() raises an error if the element doesn't exist.
- .discard() does not raise an error if the element doesn't exist.
- .pop() removes a random element because sets are unordered.

Example:

```
# Removing an element using remove()
fruits.remove("banana")

# Removing an element using discard()
fruits.discard("cherry")


# Removing a random element using pop()
```

```
random_fruit = fruits.pop()

print(random_fruit) # Randomly removed element
```

Output:

```
= RESTART: C:/Users/USER/AppData/Local/Programs/Python/Python312/PyPracticals/pr
ac 1.py
apple
```

E) Set Operations

Python sets support various operations, such as union, intersection, difference, and symmetric difference.

Union

The union of two sets is a set containing all the elements from both sets (no duplicates).

Example:

```
A = {1, 2, 3}

B = {3, 4, 5}

# Union of A and B

union_set = A.union(B) # or A | B

 print(union_set)
```

Output :

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
{1, 2, 3, 4, 5}
>
```

 # Intersection of A and B

Example:

```
intersection_set = A.intersection(B) # or A & B

print(intersection_set)
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
{1, 2, 3, 4, 5}
{3}
```

Difference

The difference between two sets is a set containing elements that are in the first set but not in the second.

Example:

```
# Difference of A and B

difference_set = A.difference(B) # or A - B

print(difference_set)
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\
ac 1.py
{1, 2, 3, 4, 5}
{1, 2}
```

 # Symmetric difference of A and B

```
sym_diff_set = A.symmetric_difference(B) # or A ^ B

print(sym_diff_set)
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
{1, 2, 3, 4, 5}
{1, 2, 4, 5}
```

F) Set Methods

Here are some commonly used set methods:

- add() — Adds a single element to the set.
- update() — Adds multiple elements to the set.
- remove() — Removes a specified element (raises an error if not found).
- discard() — Removes a specified element (does not raise an error if not found).
- pop() — Removes and returns a random element.
- clear() — Removes all elements from the set.

Examples:

```
# Set example with operations
A = {1, 2, 3}
B = {3, 4, 5}

# Union
 print(A | B)

# Intersection
 print(A & B)

# Difference
 print(A - B)

# Symmetric Difference
 print(A ^ B)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
{1, 2, 3, 4, 5}
{3}
{1, 2}
{1, 2, 4, 5}
```

Practical No 2

Program based on Functions

A. Defining and Calling Functions

To define a function in Python, you use the def keyword followed by the function name and parentheses.

Example:

```
# Define a function

def my_function():

    print("Hello from a function")

# Call the function

my_function()
```

Output:

```
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for m
>>>
== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python
Hello from a function
>>>
```

B. Parameters and Arguments

Functions can take parameters, which are variables listed inside the parentheses in the function definition. When a function is called, you can pass values known as arguments, which correspond to the parameters defined in the function.

Example:

```
def greet(name):
```

```
print(f"Hello, {name}")

# Call the function with arguments

 greet("Alice")

greet("Bob")
```

Output:

```
== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312/FUNCTION.py =
Hello, Alice
Hello, Bob
```

C. Return Values

Functions can return values using the `return` statement. This allows the function to output a result that can be stored in a variable or used in other parts of the program. For instance:

Example:

```
#Function that returns a value

 def add(a, b):

return a + b

# Store the result in a variable

 result = add(3, 4)

print(result)
```

Output:

```
== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312/FUNCTION.py =
7
```

D. Default Parameter Values and Arbitrary Arguments

Python functions can have default parameter values, which are used if no argument is passed for that parameter. Additionally, you can define functions that accept an arbitrary number of arguments using `*args` for non-keyword arguments and `**kwargs` for keyword arguments:

Example:

```
# Function with a default parameter value
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}")
# Arbitrary arguments
def print_names(*names):
    for name in names:
        print(name)
# Arbitrary keyword arguments
def print_key_values(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
# Using the functions
greet("Alice") # Uses default greeting
print_names("Alice", "Bob", "Charlie")
print_key_values(name="Alice", age=30)
```

Output:

```
== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python312/FUNCTION.py =
Hello, Alice
Alice
Bob
Charlie
name: Alice
age: 30
```

Practical No 3

Program based on FileHandling

A. **Creating a File:**Refers to the process of opening a file in write mode using the open() function, allowing you to write data to it and automatically closing the file when done.

```
import os
 def create_file(filename):
    try:
       with open(filename,'w') as f:
          f.write("Hello World!\n")
       print("File" + filename +" created successfully.")
    except error:
print("Error:could not create file" + filename)

 if      name  == "    main   ":
    filename="example.txt"
    create_file(filename)
```

Output:

```
Type "help", "copyright", "credits" or "lice
>
= RESTART: C:/Users/ADMIN/AppData/Local/Prog
py
Fileexample.txt created successfully.
>
```

Creating strings	9/25/2024 4:01 PM	Python File	2 KB
example	9/25/2024 4:21 PM	Text Document	1 KB

B. **Read a file:**Refers to opening the specified file in read mode, retrieving its contents using read(), and printing the content to the console.

```
import os
 def read_file(filename):
    try:
       with open(filename, 'r') as f:
          contents = f.read()
          print(contents)
except IOError:
print("Error: could not read file " + filename)
```

```
if      name  == "    main   ":
   filename="example.txt"
   read_file(filename)
```

Output:

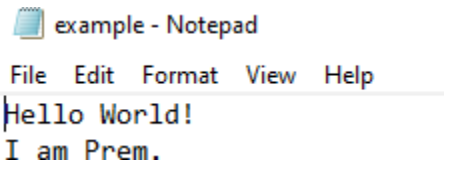
```
= RESTART: C:/Users/
py
Hello World!
```

C. **Append a file:** Adding new data to the end of an existing file without overwriting its current contents, as done by opening the file in append mode ('a') and using write() to add text.

```
import os
 def append_file(filename,text):
    try:
       with open(filename,'a') as f:
          f.write(text)
print("Text appended to file " + filename + "Successfully.")

except IOError:
print("Error: could not append to file " + filename)

 if      name  == "    main   ":
    filename="example.txt"
    append_file(filename,"I am Prem.")
```

Output:

```
py
Text appended to file example.txtSuccessfully.
```

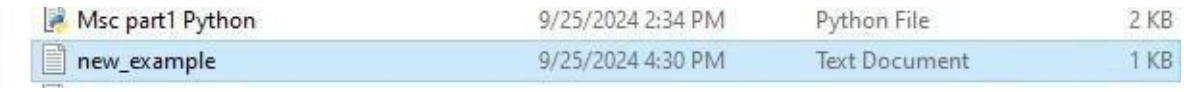


D. **Rename a file:** It means changing the file's name from filename to new_filename using the os.rename() function.

```
import os
 def rename_file(filename, new_filename):
    try:
os.rename(filename, new_filename)
        print("File " + filename + " renamed to " + new_filename + " successfully.")
    except IOError:
        print("Error: could not rename file " + filename)
 if      name  == "    main   ":
filename="example.txt" new_filename =
    "new_example.txt"
    rename_file(filename,new_filename)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Pyth
py
File example.txt renamed to new_example.txt successfully.
>
```



E. **Read Only Parts Of File:**Refers to opening the file in read mode and retrieving a specific number of characters (in this case, 5) using the read(5) function.

```
f=open("new_example.txt","r")
 print(f.read(5))
```

Output:

```
= RESTART: C:/Use
py
Hello
>>>
```

F. **Read Lines:**Refers to opening the file in read mode and retrieving one entire line of text using the readline() function.

```
f=open("new_example.txt","r")
 print(f.readline())
```

Output:

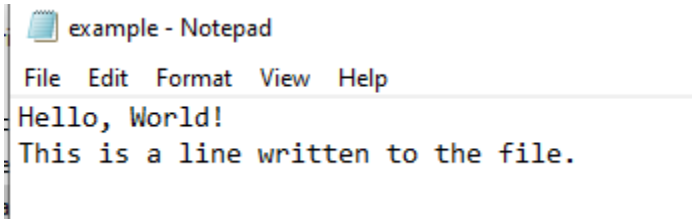
```
= RESTART: C:/Users
py
Hello World!
```

G. **Writing a File:**It means opening the file in write mode ('w'), replacing its contents with the provided data using write(), and saving the changes.

```
import os
 def write_file(filename, content):
    try:
       with open(filename, 'w') as f:
          f.write(content)
       print(f"Successfully wrote to file: {filename}")
    except IOError:
print("Error: could not write to file " + filename)

 if      name  == "    main   ":
    filename = "example.txt"
content = "Hello, World!\nThis is a line written to the file."
    write_file(filename, content)
```

Output:

```
    py
    Successfully wrote to file: example.txt
>>> |
```



H. **Closing a File:**Refers to using the close() method to terminate the file's connection, ensuring that all resources are released and any changes are saved.

```
f=open("example.txt","r")
 print(f.readline())
 f.close()
```

Output:

```
= RESTART: C:/Users/ADM
py
Hello, World!

>>
```

I. **Delete a file:I**t means using the os.remove() function to permanently remove the specified file from the filesystem.

```
import os
 def delete_file(new_filename):
    try:
os.remove(new_filename)
        print("File " + new_filename + " deleted successfully.")
    except IOError:
        print("Error: could not delete file " + new_filename)
 if       name  == "    main   ":
new_filename = "new_example.txt"
    delete_file(new_filename)
```

Output:

```
= RESTART: C:/Users/ADMIN/AppData/Local/Program
py
File new_example.txt deleted successfully.
>>
```

Practical No 4

Program based on Packages

A. **Creating and Using a Simple**

Package Step 1: Directory Structure

```
my_package/
    init .py
    math_operations.py
    string_operations.py
```

Step 2: Package Files

math_operations.py (Module inside the package):

```
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Step 3:string_operations.py (Another module inside the package):

```
def to_uppercase(s):
    return s.upper()

def to_lowercase(s):
    return s.lower()
```

Step 4:__init**__**.py (Package initializer):

```
from .math_operations import add, subtract
from .string_operations import to_uppercase, to_lowercase
```

Step 5:Using the Package

Now, you can use the my_package package in another script:

```
# main.py
import my_package

# Math operations
print(my_package.add(5, 3))        # Output: 8
```

```
print(my_package.subtract(5, 3))  # Output: 2

# String operations print(my_package.to_uppercase('hello'))
 # Output: HELLO
print(my_package.to_lowercase('WORLD')) # Output: world
```

Output:

```
8
2
HELLO
world
```

B. **Using External Packages:**Let's use an external package such as numpy (a library for numerical computations) and write a small program using it.

Step 1: Install numpy:We can install numpy packages by running "pip install numpy" on command prompt.

Step 2: Program using numpy

```
# numpy_example.py

 import numpy as np

# Create an array
arr = np.array([1, 2, 3, 4, 5])

# Perform operations
 print("Original Array:", arr)
print("Array multiplied by 2:", arr * 2)
 print("Sum of array:", np.sum(arr))
 print("Mean of array:", np.mean(arr))
```

Output:

```
Original Array: [1 2 3 4 5]
Array multiplied by 2: [ 2  4  6  8 10]
Sum of array: 15
Mean of array: 3.0
```

C. Building a Custom Package for Data Handling

We will now create a package that helps handle basic CSV operations.

Step 1: Directory Structure

```
csv_handler/
    init .py
    read_csv.py
    write_csv.py
```

Step 2: Package Files

1. **read_csv.py** (Module for reading CSV files):

```
# read_csv.py

 import csv

 def read_csv_file(Salary_Sheet):
    data = []
    with open(Salary_Sheet, mode='r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            data.append(row)
return data
```

2. **write_csv.py** (Module for writing to CSV files):

```
# write_csv.py

 import csv

def write_csv_file(Salary_Sheet, data):
    with open(Salary_Sheet, mode='w', newline='') as file:
        csv_writer = csv.writer(file)
        csv_writer.writerows(data)
```

3. **___init_____py:**Package Initializer

```
#       init      .py
from .math_operations import add, subtract
from .string_operations import to_uppercase, to_lowercase
```

Step 3:Using the package

```
import csv_handler

# Writing data to a CSV file
data_to_write = [['Name', 'Basic Pay'], ['Mukesh', 30000], ['Prem'', 25000]]
 csv_handler.write_csv_file('Salary_Sheet.csv', data_to_write)

# Reading data from a CSV file
data = csv_handler.read_csv_file('Salary_Sheet.csv')

print(data)
```

Output :

```
[['Name', 'Basic Pay'], ['Mukesh', '30000'], ['Prem', '25000']]
```

Practical No 5

Program Based On Controlled Structures

A . If ,else and elif

if Statement: Checks if the number is greater than zero and prints that it is positive. elif Statement: Checks if the number is less than zero and prints that it is negative. else Statement: Executes if neither condition is true (meaning the number is zero) and prints accordingly.

```
def check_number():
    # Get user input
number = float(input("Enter a number: "))

# Using if-elif-else to check the number if
    number > 0:
        print(f"{number} is a positive number.")
    elif number < 0:
        print(f"{number} is a negative number.")
    else:
print("The number is zero.")

if      name  == "    main   ":
    check_number()
```

Output:

```
Enter a number: 6
6.0 is a positive number.

Enter a number: -5
-5.0 is a negative number.

Enter a number: 0
The number is zero.
```

A.For Loop:

1. It Iterates over a sequence (like a list, tuple, or string) or a range of numbers.
2. The for loop iterates over a range of numbers from 1 to 10 (inclusive).
3. The range(1, 11) function generates a sequence of numbers starting from 1 up to (but not including) 11.
4. Inside the loop, it calculates the square of each number using the expression number ** 2.

```
def print_squares():
print("Squares of numbers from 1 to 10:")

# Using a for loop to iterate over a range of numbers for
    number in range(1, 11):
square = number ** 2 # Calculate the square of the number
       print(f"The square of {number} is {square}")

if      name  == "    main   ":
    print_squares()
```

Output:

```
Squares of numbers from 1 to 10:
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

A. While Loop:

1. A variable count is initialized to 1, which will be used to keep track of the current number.
2. The while count <= 5: statement starts the loop, which continues as long as count is less than or equal to 5.
3. Inside the loop, it prints the current value of count.
4. After printing, count is incremented by 1 with count += 1.

```
def count_to_five():
count = 1 # Initialize the counter

# While loop to count from 1 to 5
    while count <= 5:
print(count) # Print the current count count += 1
       # Increment the counter by 1

if      name  == "    main   ":
    count_to_five()
```

Output:

```
1
2
3
4
5
```

C.Break Statement:

1. A list numbers contains some predefined integer values.
2. The for loop iterates through the indices of the list numbers.
3. Inside the loop, an if statement checks if the current number matches the target.
4. If the target number is found, it prints the index and exits the loop immediately using the break statement.
5. An else clause is associated with the for loop, which executes if the loop completes without encountering a break (i.e., the number was not found).

```
def find_number(target):
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Using a for loop to iterate through the list for
    index in range(len(numbers)):
if numbers[index] == target:
print(f"Number {target} found at index {index}.") break #
          Exit the loop if the number is found
else:
       print(f"Number {target} not found in the list.") # This executes if the loop
 completes without a break

if       name  == "    main   ":
target_number = int(input("Enter a number to find: "))
    find_number(target_number)
```

Output:

```
Enter a number to find: 70
Number 70 found at index 6.
```

D.Continue Statement:

1. The for loop iterates over the numbers from 1 to 10 using range(1, 11).
2. Inside the loop, an if statement checks if the current number is odd using the condition number % 2 != 0.

3. If the number is odd, the continue statement is executed, which skips the remaining code in the loop and proceeds to the next iteration.
4. If the number is even, it is printed to the console.

```
def print_even_numbers():
    print("Even numbers from 1 to 10:")

# Using a for loop to iterate through numbers from 1 to 10 for
    number in range(1, 11):
if number % 2 != 0:
            continue # Skip odd numbers
        print(number) # Print even numbers

if      name  == "    main   ":
    print_even_numbers()
```

Output:

```
Even numbers from 1 to 10:
2
4
6
8
10
```

E.Pass Statement:

1. The for loop iterates through each number in the provided list.
2. An if statement checks if the current number is even using number % 2 == 0.
3. If the number is even, the pass statement is executed, which means no action is taken for even numbers.
4. If the number is odd, it is printed to the console.
5. The main() function creates a list of numbers from 1 to 10 and calls the skip_even_numbers() function.

```
def skip_even_numbers(numbers):
    for number in numbers:
if number % 2 == 0:
            pass # Do nothing for even numbers
        else:
print(f"Odd number: {number}")

def main():
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
print("Processing numbers:")
    skip_even_numbers(numbers)

if      name  == "    main   ":
    main()
```

Output:

```
Processing numbers:
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

Practical No 6
Program based on Exception Handling

In Python, exception handling is done using try, except, else, and finally blocks to manage errors during program execution.

- try block: Contains the code that might raise an exception.
- except block: Handles specific exceptions if they occur.
- else block (optional): Runs if no exceptions are raised in the try block.
- finally block (optional): Always runs, regardless of whether an exception occurred or not.
- If an error occurs, it's caught by the corresponding except block, preventing the program from crashing.

1. Basic Exception Handling:

- try block contains the code that may potentially throw exceptions.
- except ValueError handles the case where the user input is not a valid integer.
- except ZeroDivisionError handles cases where division by zero is attempted.

```
try:
# Prompt user for input and convert to integer
    number = int(input("Enter a number: "))
result = 100 / number # Attempt division
    print("The result is:", result)
except ValueError:
# Handle invalid input (non-numeric) print("Invalid
    input! Please enter a valid integer.")
except ZeroDivisionError:
# Handle division by zero print("Error!
    Cannot divide by zero.")
```

Output:

```
Enter a number: 5
The result is: 20.0
Enter a number: c
Invalid input! Please enter a valid integer.
```

```
---------- RESTART: C:/User
Enter a number: 0
Error! Cannot divide by zero.
```

2. **Catching Multiple Exceptions:**It can be done using a single except block by specifying a tuple of exceptions, like except (TypeError, ValueError):. This allows handling different exceptions with the same code block.

```
try:
num = int(input("Enter a number: ")) result =
    10 / num
except ValueError:
    print("That's not a valid number.")
 except ZeroDivisionError:
print("Cannot divide by zero.")
```

Output:

```
Enter a number: abc
That's not a valid number.
Enter a number: 0
Cannot divide by zero.
```

3. **Using else block:**The else block will run only if no exceptions were raised in the try block.

```
try:
num = int(input("Enter a number: ")) result =
    10 / num
 except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
print("Result is:", result)
```

Output:

```
Enter a number: 0
Cannot divide by zero.
```

```
Enter a number: 5
Result is: 2.0
```

4. **Using finally Block:** The finally block will execute no matter what happens in the try and except blocks.

```
try:
file = open("example.txt", "r") content
    = file.read()
 except FileNotFoundError:
    print("File not found.")
finally:
print("Execution complete.")
```

Output:

If file not found

```
File not found.
Execution complete.
```

5. **Raising Exceptions:** You can use raise to manually trigger an exception.

```
try:
age = int(input("Enter your age: ")) if
    age < 0:
        raise ValueError("Age cannot be negative.")
 except ValueError as ve:
print(ve)
```

Output:

```
Enter your age: -21
Age cannot be negative.
```

6. **Custom Exceptions:** You can define your own exceptions by subclassing the built-in Exception class.

```
class NegativeAgeError(Exception):
    pass

def check_age(age):
    if age < 0:
raise NegativeAgeError("Age cannot be negative.")

try:
age = int(input("Enter your age: "))
    check_age(age)
except NegativeAgeError as e:
    print(e)
```

Output:

```
Enter your age: -6
Age cannot be negative.
```

Practical No 7

Program based on Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reuse and allows for hierarchical classifications.

A) Example of Inheritance in Python

Let's create an example of a parent class Animal and two child classes Dog and Cat that inherit from Animal.

Example Code:

```
# Parent class
 class Animal:
    def          init      (self,
        name): self.name =
        name

def speak(self):
return f"{self.name} makes a sound."

# Child class inheriting from Animal
 class Dog(Animal):
def speak(self):
return f"{self.name} barks."

# Another child class inheriting from Animal
 class Cat(Animal):
def speak(self):
return f"{self.name} meows."

# Creating objects dog
 = Dog("Buddy") cat =
 Cat("Whiskers")

# Calling the methods
 print(dog.speak())
 print(cat.speak())
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
Buddy barks.
Whiskers meows.
>
```

Explanation:

1. Parent Class (Animal):
    - The Animal class has an __init__() method for setting the name attribute and a speak() method to describe a generic animal sound.
2. Child Classes (Dog and Cat):
    - Both Dog and Cat inherit from the Animal class.
    - They override the speak() method to provide behavior specific to each animal (barking for dogs, meowing for cats).

B) Example of Using super() in Inheritance

The super() function is used to call methods from the parent class inside the child class. Here's an example where we extend the parent class method using super().

Example:

```
# Parent class
 class Animal:
    def         init     (self, name,
       species): self.name = name
self.species = species

def speak(self):
return f"{self.name} makes a sound."

# Child class Dog with super()
 class Dog(Animal):
def     init     (self, name, species, breed):
super().        init     (name, species) # Call the parent class constructor
```

```
self.breed = breed

def speak(self):
return f"{self.name}, the {self.breed}, barks."

# Creating an object of Dog
dog = Dog("Buddy", "Dog", "Golden Retriever")

# Accessing attributes and methods
 print(dog.name)
print(dog.species)
 print(dog.breed)
 print(dog.speak())
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
Buddy
Dog
Golden Retriever
Buddy, the Golden Retriever, barks.
```

Explanation:

- The Dog class uses super().__init__() to call the constructor of the Animal class, ensuring that the name and species attributes are initialized properly.
- The speak() method is overridden in the Dog class to give a more specific message.

C) Types of Inheritance:

1. Single Inheritance – A child class inherits from one parent class.
2. Multiple Inheritance – A child class inherits from more than one parent class.
3. Multilevel Inheritance – A child class inherits from a parent class, which in turn inherits from another parent class.

Example of Multiple Inheritance:

```
# Parent class 1
 class Animal:
def speak(self):
return "Animal sound"

# Parent class 2
 class Vehicle:
def move(self):
return "Moves on the road"

# Child class inheriting from both Animal and Vehicle
 class RobotDog(Animal, Vehicle):
def speak(self):
return "RobotDog barks electronically"

# Creating an object of RobotDog
 robot_dog = RobotDog()

# Accessing methods
 print(robot_dog.speak())
 print(robot_dog.move())
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
RobotDog barks electronically
Moves on the road
```

D) Key Takeaways:

- Inheritance allows one class to inherit properties and methods from another, making code reusable and organized.
- Overriding allows child classes to modify or replace methods from the parent class.
- super() is used to call the parent class's methods or constructors, ensuring parent class attributes are initialized properly.

Practical No 8

Program based onOverloading

In Python, method overloading (as seen in other languages like Java) is not directly supported because Python does not support function signature-based overloading. However, we can achieve similar behavior by using default parameters, variable-length arguments, or checking types inside the function to handle different input types.

Here are examples of how to achieve the effect of overloading using different techniques in Python.

Example 1: Overloading with Default Arguments

You can define a function with default parameters to simulate overloading.

Code:

```
class Calculator:

def add(self, a, b=0, c=0):

"""Adds two or three numbers depending on the arguments passed.""" return a +

        b + c



# Create an object of Calculator class

 calc = Calculator()

# Calling with two arguments

 print(calc.add(5, 10))

# Calling with three arguments

 print(calc.add(5, 10, 15))
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
15
30
```

In this example, the add() method can take two or three arguments, simulating overloading by providing default values for b and c.

Example 2: Overloading with Variable-Length Arguments (*args)

You can use *args to accept a variable number of arguments.

Code:

```
class Calculator:

def add(self, *args):

"""Adds any number of numbers.""" return

        sum(args)


# Create an object of Calculator class

 calc = Calculator()


# Calling with two arguments

 print(calc.add(5, 10))

# Calling with three arguments

 print(calc.add(5, 10, 15))
```

```
# Calling with more than three arguments

print(calc.add(5, 10, 15, 20, 25))
```

Output:

```
Edit  Shell  Debug  Options  Window  Help
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
15
30
```

In this example, *args allows the method to accept any number of arguments, and it calculates the sum of all the arguments passed.

Example 3: Overloading by Checking the Type of Arguments

You can simulate method overloading by checking the types of the arguments passed inside the function.

Code:

```
class Calculator:

    def multiply(self, a, b):

"""Multiplies either two numbers or concatenates strings.""" if

        isinstance(a, int) and isinstance(b, int):

            return a * b
```

```
        elif isinstance(a, str) and isinstance(b, int):
            return a * b
else:
return "Invalid input types"
# Create an object of Calculator class
 calc = Calculator()
# Multiply two integers
 print(calc.multiply(5, 10))
# Repeat a string multiple times
 print(calc.multiply("Hello", 3))
# Invalid input types
 print(calc.multiply("Hello", "World"))
```

Output:

```
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\pr
ac 1.py
50
HelloHelloHello
Invalid input types
```

In this example, the multiply() method behaves differently depending on the types of arguments passed. If two integers are passed, it performs multiplication. If a string and an integer are passed, it repeats the string. If invalid types are passed, it returns an error message.

Example 4: Overloading with @singledispatch (Type-Based Overloading)

Python's functools module provides a @singledispatch decorator to create type-based function overloading.

Code:

from functools import singledispatch

```
@singledispatch
 def display(value):
    print("Default:", value)
 @display.register(int)
 def _(value):
    print("Integer:", value)
 @display.register(str)
 def _(value):
    print("String:", value)
 @display.register(list)
 def _(value):
print("List:", value)
# Calling the overloaded function
 display(5)
display("Hello")
 display([1, 2, 3])
 display(3.14)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\
ac 1.py
Integer: 5
String: Hello
List: [1, 2, 3]
Default: 3.14
```

In this example, @singledispatch allows us to overload the display() function based on the type of the argument. It selects the appropriate function based on the type of the passed parameter.

Summary

While Python doesn't support traditional overloading based on function signatures, you can achieve similar behavior through:

- Default arguments
- Variable-length arguments (*args)
- Type checking with isinstance()
- Type-based dispatch with @singledispatch.

Each of these approaches allows for flexibility in defining functions that handle different input types and numbers of arguments.

Practical No 9

Working on Big Data Libraries: Numpy, Pandas, Matplotlib

1. Numpy:

NumPy (Numerical Python) is a powerful Python library widely used for numerical computing, data manipulation, and scientific computations. It provides support for arrays, matrices, and high-level mathematical functions to operate on these data structures efficiently.

Key Features of NumPy

1. N-dimensional Arrays (ndarray): The core feature of NumPy is the ndarray object, which allows for fast and efficient array processing.
2. Mathematical Functions: NumPy provides a wide array of mathematical functions to operate on arrays, including trigonometric, statistical, and linear algebra functions.
3. Broadcasting: Allows operations between arrays of different shapes, making element-wise operations simple and intuitive.
4. Integration with Other Libraries: NumPy arrays are often used as the foundational data structure in other libraries such as Pandas, SciPy, and machine learning libraries like TensorFlow and PyTorch.

Installation

To install NumPy, you can use pip:

-pip install numpy

Importing Numpy

To import NumPy, you can use:

-import numpy as np

A) NumPy Arrays

NumPy provides a high-performance multidimensional array object, ndarray, and tools for working with these arrays.

Creating Arrays

You can create arrays from Python lists or tuples.

Example:

```
import numpy as np

# 1D Array
arr = np.array([1, 2, 3, 4, 5])
 print(arr)

# 2D Array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
 print(arr2d)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\
ac 1.py
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
>
= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\
ac 1.py
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
>
```

B) Array Operations

NumPy allows for element-wise operations on arrays without the need for loops. Basic

Operations

Examples:

```
import numpy as np

# Creating NumPy arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise addition
```

```
c = a + b
 print("Addition:", c)

# Element-wise multiplication
 d = a * b
 print("Multiplication:", d)

# Broadcasting: scalar and array
 e = a + 10
print("Broadcasting (adding 10):", e)

# Summing all elements
print("Sum of elements:", np.sum(a))

# Mean of elements
print("Mean of elements:", np.mean(a))
```

Output:

```
Addition: [5 7 9]
Multiplication: [ 4 10 18]
Broadcasting (adding 10): [11 12 13]
Sum of elements: 6
Mean of elements: 2.0
```

C) Indexing and Slicing

You can access and modify specific elements or slices of a NumPy array using indexing.

Example:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Accessing a single element
 print(arr[0])
```

```
# Slicing
 print(arr[1:4])

# Modifying a slice
 arr[2:4] = [10, 20]
print(arr)
```

Output:

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:\Users\USER\AppData\Local\Programs\Python\Python312\PyPracticals\
ac 1.py
1
[ 1  2 10 20  5]
```

B. Reshaping Arrays

Change the shape of arrays using reshape.

```
import numpy as np #

 Creating a 1D array

arr = np.arange(1, 13)

 print("Original array:", arr)

# Reshaping the array into a 3x4 matrix

 reshaped_arr = arr.reshape(3, 4)

 print("Reshaped array (3x4):\n", reshaped_arr)

 # Flattening the 2D array back to 1D

 flattened_arr = reshaped_arr.flatten()

 print("Flattened array:", flattened_arr)
```

Output:

```
Original array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
Reshaped array (3x4):
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Flattened array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

C. Array Arithmetic

Perform element-wise operations on arrays.

```
import numpy as np
 #Creating two arrays

arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

#Element-wise addition
 print("Addition:", arr1 + arr2)

#Element-wise subtraction
 print("Subtraction:", arr1 - arr2)

#Element-wise multiplication
 print("Multiplication:", arr1 * arr2)

#Element-wise division
 print("Division:", arr1 / arr2)
```

Output:

```
Addition: [ 6  8 10 12]
Subtraction: [-4 -4 -4 -4]
Multiplication: [ 5 12 21 32]
Division: [0.2        0.33333333 0.42857143 0.5
```

D. Broadcasting

Perform operations between arrays of different shapes.

```
import numpy as np
```

```
# Creating a 1D array and a 2D array
 arr1 = np.array([1, 2, 3])
arr2 = np.array([[4], [5], [6]])

# Broadcasting: Adding a 1D array to each row of a 2D array
 result = arr2 + arr1
print("Broadcasting result:\n", result)
```

Output:

```
Broadcasting result:
  [[5 6 7]
  [6 7 8]
  [7 8 9]]
```

E. Array Concatenation

Concatenate arrays along different axes.

```
import numpy as np

# Creating two 2D arrays arr1
 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Concatenating along rows (axis=0)
concat_rows = np.concatenate((arr1, arr2), axis=0)
 print("Concatenated along rows:\n", concat_rows)

# Concatenating along columns (axis=1) concat_cols
 = np.concatenate((arr1, arr2), axis=1)
 print("Concatenated along columns:\n", concat_cols)
```

Output:

```
Concatenated along rows:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
Concatenated along columns:
 [[1 2 5 6]
 [3 4 7 8]]
```

F. Array Transposition

Transpose arrays (swap rows with columns).

```
import numpy as np

# Creating a 2D array
arr = np.array([[1, 2], [3, 4], [5, 6]])

# Transposing the array
 transposed = np.transpose(arr)
 print("Original array:\n", arr)
print("Transposed array:\n", transposed)
```

Output:

```
Original array:
 [[1 2]
 [3 4]
 [5 6]]
Transposed array:
 [[1 3 5]
 [2 4 6]]
```

G. Stacking Arrays

Stack arrays vertically or horizontally.

```
import numpy as np

# Creating two 1D arrays
 arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Stacking arrays vertically
 vert_stack = np.vstack((arr1, arr2))
print("Vertically stacked:\n", vert_stack)

# Stacking arrays horizontally horiz_stack
 = np.hstack((arr1, arr2))
 print("Horizontally stacked:", horiz_stack)
```

Output:

```
Vertically stacked:
 [[1 2 3]
 [4 5 6]]
Horizontally stacked: [1 2 3 4 5 6]
```

H. Splitting Arrays

Split arrays into multiple subarrays.

```
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4, 5, 6])

# Splitting the array into 3 subarrays
 split_arr = np.split(arr, 3)
 print("Splitted array:", split_arr)

# Creating a 2D array
arr2D = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Splitting the 2D array along rows
 split_arr2D = np.vsplit(arr2D, 3)
```

```
print("Vertically split 2D array:\n", split_arr2D)
```

Output:

```
Splitted array: [array([1, 2]), array([3, 4]), array([5, 6])]
Vertically split 2D array:
 [array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
```

I. Array Sorting

Sort arrays along a specific axis.

```
import numpy as np

# Creating a 1D array
arr = np.array([3, 1, 2, 5, 4])

# Sorting the array
 sorted_arr = np.sort(arr)
print("Sorted array:", sorted_arr)

# Creating a 2D array
arr2D = np.array([[3, 1, 2], [5, 4, 6]])

# Sorting along rows (axis=1)
 sorted_arr2D = np.sort(arr2D, axis=1)
print("2D array sorted along rows:\n", sorted_arr2D)
```

Output:

```
Sorted array: [1 2 3 4 5]
2D array sorted along rows:
 [[1 2 3]
 [4 5 6]]
```

M.Matrix Operations

This example shows how to perform matrix operations such as dot product, transpose, and inverse.

```
import numpy as np

# Creating two matrices
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

# Matrix multiplication (dot product)
 matrix_product = np.dot(matrix_a, matrix_b)
 print("Matrix multiplication:\n", matrix_product)

# Matrix transpose
matrix_transpose = np.transpose(matrix_a)
 print("Transpose of matrix A:\n", matrix_transpose)

# Matrix inverse (only for square matrices)
 matrix_inverse = np.linalg.inv(matrix_a)
 print("Inverse of matrix A:\n", matrix_inverse)
```

Output:

```
Matrix multiplication:
  [[19 22]
  [43 50]]
Transpose of matrix A:
  [[1 3]
  [2 4]]
Inverse of matrix A:
  [[-2.   1. ]
  [ 1.5 -0.5]]
```

N.Random Array and Statistical Functions

This example demonstrates generating random arrays and performing statistical analysis on them.

```
import numpy as np

# Generate an array of random values from a normal distribution
 random_array = np.random.randn(5)
print("Random array:", random_array)

# Compute the standard deviation
 std_dev = np.std(random_array)
```

```
print("Standard Deviation:", std_dev)

# Generate a 3x3 array of random integers between 1 and 10
random_ints = np.random.randint(1, 10, (3, 3))
print("Random 3x3 integer array:\n", random_ints)

# Find the max and min values
max_value = np.max(random_ints)
min_value = np.min(random_ints)
print("Max value:", max_value)
print("Min value:", min_value)
```

Output:

```
Random array: [ 1.78024161  1.09563637 -0.16679139  0.54447887  0.01210994]
Standard Deviation: 0.7157059923549496
Random 3x3 integer array:
 [[3 6 9]
 [7 4 1]
 [1 9 4]]
Max value: 9
Min value: 1
```

O.Using Numpy to solve linear equations

This example demonstrates how to solve a system of linear equations using NumPy.

$$2x+y=5$$

$$x+3y=5$$

```
import numpy as np

# Coefficients of the equations
A = np.array([[2, 1], [1, 3]])

# Constants on the right-hand side
B = np.array([5, 7])

# Solving the system of linear equations
solution = np.linalg.solve(A, B)
print("Solution (x, y):", solution)
```

Output:

```
Solution (x, y): [1.6 1.8]
```

2. Pandas

Pandas is a powerful and widely-used Python library for data manipulation, analysis, and exploration. It is built on top of NumPy and provides data structures like DataFrame and Series that make it easier to work with structured data, particularly for tasks related to data cleaning, transformation, and analysis.

Key Features of Pandas

1. Data Structures:
   - Series: A one-dimensional labeled array, similar to a list or a column in a spreadsheet.
   - DataFrame: A two-dimensional labeled data structure, similar to a table or a spreadsheet, where each column can be of a different data type (e.g., integers, strings, floats).
2. Handling Missing Data: Pandas provides easy ways to handle missing or NaN values, such as filling missing values with defaults, interpolating, or dropping missing data.
3. Data Alignment: Pandas automatically aligns data when performing operations on different data structures. This feature makes it easy to handle datasets that may not be perfectly aligned.
4. Data Transformation: Pandas allows for various data transformation operations like filtering, sorting, reshaping, grouping, and aggregating data, which are crucial for data analysis.
5. File I/O Operations: Pandas can easily read and write data from and to various formats, such as CSV, Excel, JSON, SQL databases, and more.
6. Integration with Other Libraries: Pandas integrates well with other Python libraries like Matplotlib (for plotting) and NumPy (for numerical operations), making it an essential part of the Python data science ecosystem.

A. Creating a Pandas DataFrame

Create a DataFrame from a dictionary and a list of lists.

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['Digvijay', 'Anuran', 'Chiranjiv', 'Ashutosh'],
```

```
'Age': [21, 22, 23, 24],
        'City': ['Ahmedabad', 'Lucknow', 'Chennai', 'Mumbai']}
 df = pd.DataFrame(data)
print("DataFrame from dictionary:\n", df)

# Creating a DataFrame from a list of lists
 data = [['Digvijay', 21, 'Ahmedabad'],
['Anuran', 22, 'Lucknow'],
['Ashutosh', 24, 'Mumbai']]
df2 = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
 print("\nDataFrame from list of lists:\n", df2)
```

Output:

```
DataFrame from dictionary:
         Name  Age       City
0    Digvijay   21  Ahmedabad
1      Anuran   22    Lucknow
2   Chiranjiv   23    Chennai
3    Ashutosh   24     Mumbai

DataFrame from list of lists:
        Name  Age       City
0   Digvijay   21  Ahmedabad
1     Anuran   22    Lucknow
2   Ashutosh   24     Mumbai
```

B. Reading data from a csv file.

Read data from a CSV file into a Pandas DataFrame.

```
import pandas as pd

# Reading a CSV file
df = pd.read_csv('Salary Sheet.csv')

# Display the first few rows of the DataFrame
 print("First 5 rows of the DataFrame:\n", df.head())

# Display the column names
 print("\nColumn names:", df.columns)
```

Output:

```
First 5 rows of the DataFrame:
       Name  Basic Pay  TA(4%)  DA(6%)  Gross Salary  PF(3%)  Net Salary
0    Ankush      33000    1320    1980         36300    1089       35211
1    Yogesh      25000    1000    1500         27500     825       26675
2    Khushi      31000    1240    1860         34100    1023       33077
3    Ayushi      59000    2360    3540         64900    1947       62953
4  Anubhav       46000    1840    2760         50600    1518       49082

Column names: Index(['Name', 'Basic Pay', 'TA(4%)', 'DA(6%)', 'Gross Salary', 'PF(3%)',
       'Net Salary'],
      dtype='object')
```

C. Selecting Data(Selecting and Indexing)

Access specific rows and columns in a DataFrame.

```
import pandas as pd

# Sample DataFrame
 data = {'Name': ['Anuran', 'Prem', 'Mukesh', 'Sunny'],
        'Age': [21, 22, 23, 24],
        'City': ['Mumbai', 'Delhi', 'Bengaluru', 'Gurugram']}
 df = pd.DataFrame(data)

# Selecting a single column
 print("Age column:\n", df['Age'])

# Selecting multiple columns
print("\nName and City columns:\n", df[['Name', 'City']])

# Selecting rows by index (iloc: integer location)
 print("\nFirst two rows:\n", df.iloc[:2])

# Selecting rows by condition
print("\nRows where Age > 22:\n", df[df['Age'] > 22])
```

Output:

```
Age column:
 0    21
1    22
2    23
3    24
Name: Age, dtype: int64

Name and City columns:
      Name       City
0  Anuran     Mumbai
1    Prem      Delhi
2  Mukesh  Bengaluru
3   Sunny   Gurugram

First two rows:
      Name  Age    City
0  Anuran   21  Mumbai
1    Prem   22   Delhi

Rows where Age > 22:
      Name  Age       City
2  Mukesh   23  Bengaluru
3   Sunny   24   Gurugram
```

D. Adding and Removing Columns

Add a new column or remove an existing column in a DataFrame.

```
import pandas as pd

# Sample DataFrame
 data = {'Name': ['Prem', 'Satyendra', 'Mahesh'],
       'Age': [25, 30, 35],
       'City': ['Bhubaneshwar', 'Pune', 'Lucknow']}
 df = pd.DataFrame(data)

# Adding a new column
df['Salary'] = [50000, 60000, 70000]
print("DataFrame after adding a new column:\n", df)

# Removing a column
df = df.drop('Salary', axis=1)
print("\nDataFrame after removing the Salary column:\n", df)
```

Output:

```
DataFrame after adding a new column:
         Name  Age          City  Salary
0       Prem   25  Bhubaneshwar   50000
1  Satyendra   30          Pune   60000
2     Mahesh   35       Lucknow   70000

DataFrame after removing the Salary column:
         Name  Age          City
0       Prem   25  Bhubaneshwar
1  Satyendra   30          Pune
2     Mahesh   35       Lucknow
```

E. Sorting Data

Sort the data in a DataFrame by a specific column.

```
import pandas as pd

# Sample DataFrame
 data = {'Name': ['Prem', 'Mahesh', 'Bindu', 'Angel'],
        'Age': [40, 25, 35, 30],
        'City': ['Dubai', 'Capetown', 'Canberra', 'Wellington']}
 df = pd.DataFrame(data)

# Sorting by age in ascending order
 sorted_df = df.sort_values(by='Age')
print("DataFrame sorted by Age (ascending):\n", sorted_df)

# Sorting by name in descending order
sorted_df = df.sort_values(by='Name', ascending=False)
 print("\nDataFrame sorted by Name (descending):\n", sorted_df)
```

Output:

```
DataFrame sorted by Age (ascending):
      Name  Age        City
1  Mahesh   25    Capetown
3   Angel   30  Wellington
2   Bindu   35    Canberra
0    Prem   40       Dubai

DataFrame sorted by Name (descending):
      Name  Age        City
0    Prem   40       Dubai
1  Mahesh   25    Capetown
2   Bindu   35    Canberra
3   Angel   30  Wellington
```

F. GroupBy and Aggregation

Group the data by a specific column and apply aggregate functions like sum, mean, etc.

```
import pandas as pd

# Sample DataFrame
data = {'Department': ['HR', 'IT', 'HR', 'Finance', 'IT','Finance'],
'Employee': ['Sonal', 'Anita', 'Kiran', 'Rahul', 'Virat','Gulfisha'],
        'Salary': [50000, 60000, 55000, 65000, 62000,68000]}
df = pd.DataFrame(data)

# Grouping by department and calculating the mean salary
 grouped_df = df.groupby('Department')['Salary'].mean()
 print("Mean salary by department:\n", grouped_df)

# Grouping by department and calculating the sum of salaries
 grouped_sum = df.groupby('Department')['Salary'].sum()
 print("\nTotal salary by department:\n", grouped_sum)
```

Output:

```
Mean salary by department:
 Department
Finance    66500.0
HR         52500.0
IT         61000.0
Name: Salary, dtype: float64

Total salary by department:
 Department
Finance    133000
HR         105000
IT         122000
Name: Salary, dtype: int64
```

G. Handling Missing Data

Handle missing values by filling or dropping them.

```
import pandas as pd
 import numpy as np

# Sample DataFrame with missing values
 data = {'Name': ['Anuran', 'Balwant', 'Raju', np.nan],
       'Age': [25, np.nan, 35, 40],
```

```
        'City': ['Pune', np.nan, 'Hydreabad', 'Bengaluru']}
 df = pd.DataFrame(data)

# Dropping rows with missing values
 df_dropped = df.dropna()
print("DataFrame after dropping rows with missing values:\n", df_dropped)

# Filling missing values with a default value
df_filled = df.fillna({'Name': 'Prem', 'Age': 30, 'City': 'Noida'})
 print("\nDataFrame after filling missing values:\n", df_filled)
```

Output:

```
DataFrame after dropping rows with missing values:
      Name   Age       City
0   Anuran  25.0       Pune
2     Raju  35.0  Hydreabad

DataFrame after filling missing values:
       Name   Age       City
0    Anuran  25.0       Pune
1   Balwant  30.0      Noida
2      Raju  35.0  Hydreabad
3      Prem  40.0  Bengaluru
```

H. Merging and Joining Data-Frames

Merge or join two DataFrames.

```
import pandas as pd

# Sample DataFrames
 df1 = pd.DataFrame({'Employee': ['Sonal', 'Kiran', 'Gulfisha'],
                'Department': ['HR', 'IT', 'Finance']})
 df2 = pd.DataFrame({'Employee': ['Sonal', 'Rachana', 'Gulfisha'],
                'Salary': [50000, 60000, 55000]})

# Merging DataFrames on the 'Employee' column merged_df
 = pd.merge(df1, df2, on='Employee', how='inner')
 print("Merged DataFrame (inner join):\n", merged_df)

# Left join
left_join_df = pd.merge(df1, df2, on='Employee', how='left')
 print("\nLeft join DataFrame:\n", left_join_df)
```

Output:

```
Merged DataFrame (inner join):
    Employee Department  Salary
0      Sonal         HR   50000
1   Gulfisha    Finance   55000

Left join DataFrame:
    Employee Department   Salary
0      Sonal         HR  50000.0
1      Kiran         IT      NaN
2   Gulfisha    Finance  55000.0
```

I. Pivot Tables

Create pivot tables to summarize data.

```
import pandas as pd

# Sample DataFrame
data = {'Department': ['HR', 'HR', 'IT', 'Finance', 'IT'],
'Employee': ['Sonal', 'Rohit', 'Saqlain','Disha', 'Arvind'],
       'Salary': [50000, 55000, 60000, 65000, 62000]}
df = pd.DataFrame(data)

# Creating a pivot table showing the sum of salaries by department
pivot_table = df.pivot_table(values='Salary', index='Department', aggfunc='sum')
 print("Pivot table (sum of salaries by department):\n", pivot_table)
```

Output:

```
Pivot table (sum of salaries by department):
             Salary
Department
Finance       65000
HR           105000
IT           122000
```

J. Exporting Data to CSV

Save a DataFrame to a CSV file.

```
import pandas as pd

# Sample DataFrame
 data = {'Name': ['Anuran', 'Aarohi', 'Shivam', 'Rahul'],
        'Basic Pay':[69000, 71000, 55000, 45000],}
df = pd.DataFrame(data)

# Exporting the DataFrame to a CSV file
 df.to_csv('Salary Sheet.csv', index=False)
 print("DataFrame exported to 'Salary Sheet.csv'")
```

Output:

```
====== RESTART: C:/Users/anura/OneDrive/P:
DataFrame exported to 'Salary Sheet.csv'
```

	A	B
1	Name	Basic Pay
2	Anuran	69000
3	Aarohi	71000
4	Shivam	55000
5	Rahul	45000

3. Matplotlib

Matplotlib is a popular Python library used for creating static, animated, and interactive visualizations. It provides a flexible way to generate a wide variety of plots and charts, making it essential for data analysis, exploration, and presentation. Matplotlib is commonly used in conjunction with other libraries like NumPy and Pandas for data science tasks.

Key Features of Matplotlib:

1. Wide Variety of Plots: Matplotlib allows you to create different types of visualizations such as:
    - Line plots
    - Bar charts (vertical and horizontal)
    - Scatter plots
    - Histograms
    - Pie charts

- Box plots
- Heatmaps
- And more

2. Customizable Plots: Every aspect of a plot can be customized, including axes, colors, labels, markers, styles, and legends. You can also add titles, grid lines, and annotations.
3. Integration with Other Libraries: Matplotlib integrates seamlessly with NumPy, Pandas, and SciPy, making it easy to visualize data directly from DataFrames or arrays.
4. Subplots: You can create multiple plots in a single figure by arranging them in grids, making it easier to compare different datasets side by side.
5. Interactive Plots: While primarily used for static images, Matplotlib also supports interactive plotting when used in environments like Jupyter notebooks. Plots can also be saved in various formats like PNG, PDF, and SVG.

Components of a Matplotlib Plot:

A Matplotlib figure typically consists of the following components:

- Figure: The overall container that holds one or more plots.
- Axes: The area where data is plotted (can be more than one in a figure).
- Axis: Represents x and y axes within the plot.
- Labels: Descriptive text for axes and data points.
- Legend: A guide to distinguish different data series in the plot.
- Grid: Optional lines on the plot to enhance readability.

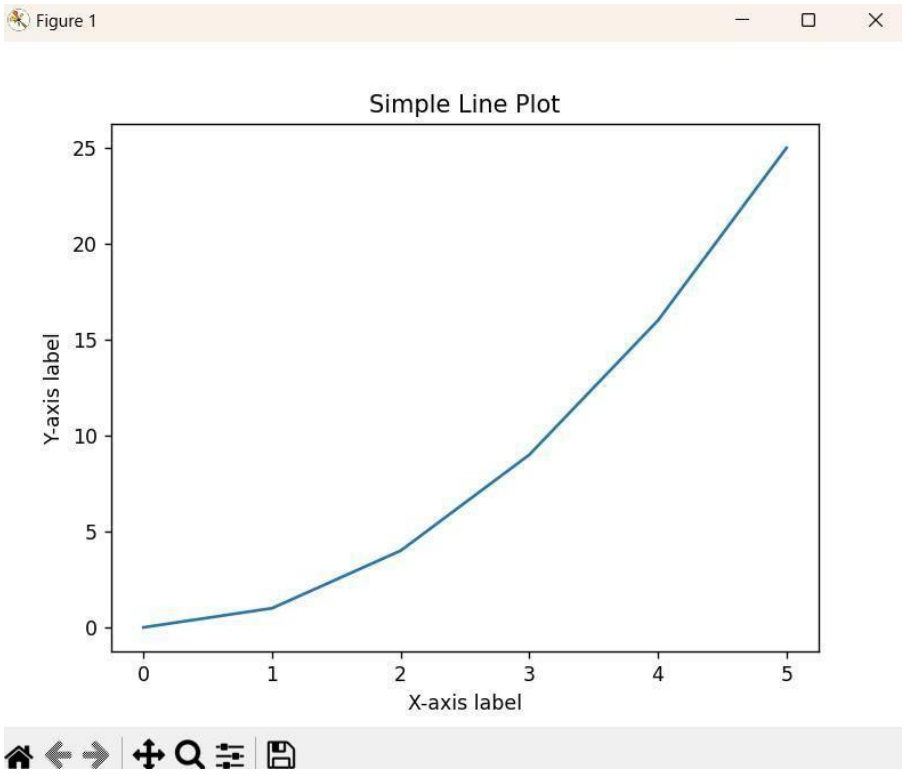
A. Basic Line Plot

This program shows how to create a simple line plot with labels and a title.

```
import matplotlib.pyplot as plt

# Data for plotting x
 = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

# Creating the plot
 plt.plot(x, y)

# Adding labels and title
 plt.xlabel('X-axis label')
 plt.ylabel('Y-axis label')
 plt.title('Simple Line Plot')
```

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

Output:



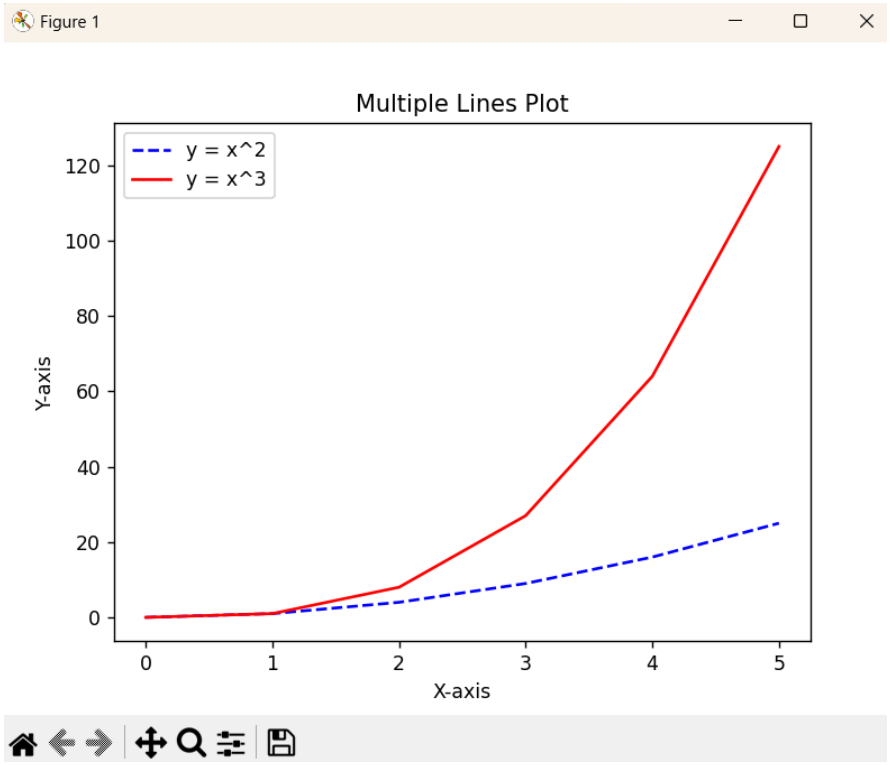
B. Multiple Lines on the Same Plot

This example plots multiple lines on the same graph with different styles.

```
import matplotlib.pyplot as plt

# Data for plotting x
 = [0, 1, 2, 3, 4, 5]
y1 = [0, 1, 4, 9, 16, 25]
y2 = [0, 1, 8, 27, 64, 125]

# Plotting multiple lines
plt.plot(x, y1, label='y = x^2', color='blue', linestyle='--')
 plt.plot(x, y2, label='y = x^3', color='red', linestyle='-')

# Adding labels and title
 plt.xlabel('X-axis')
 plt.ylabel('Y-axis')
 plt.title('Multiple Lines Plot')
```

```
plt.legend()

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

Output:



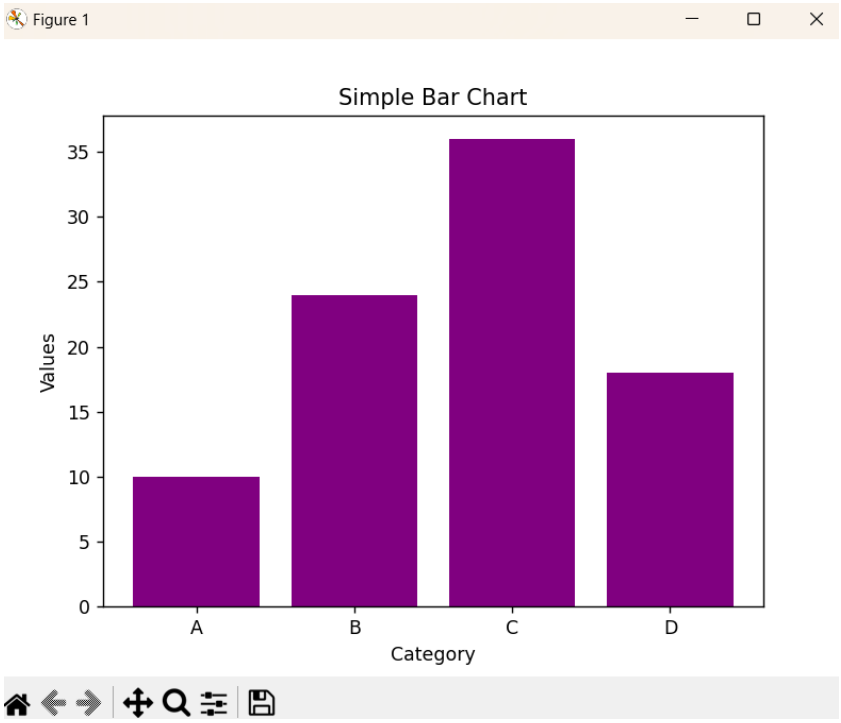
C. Bar Chart

Create a bar chart to compare different categories.

```
import matplotlib.pyplot as plt

# Data for plotting categories
 = ['A', 'B', 'C', 'D'] values =
 [10, 24, 36, 18]

# Creating the bar chart plt.bar(categories,
 values, color='purple')

# Adding labels and title
 plt.xlabel('Category')
 plt.ylabel('Values')
 plt.title('Simple Bar Chart')
```

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

Output:

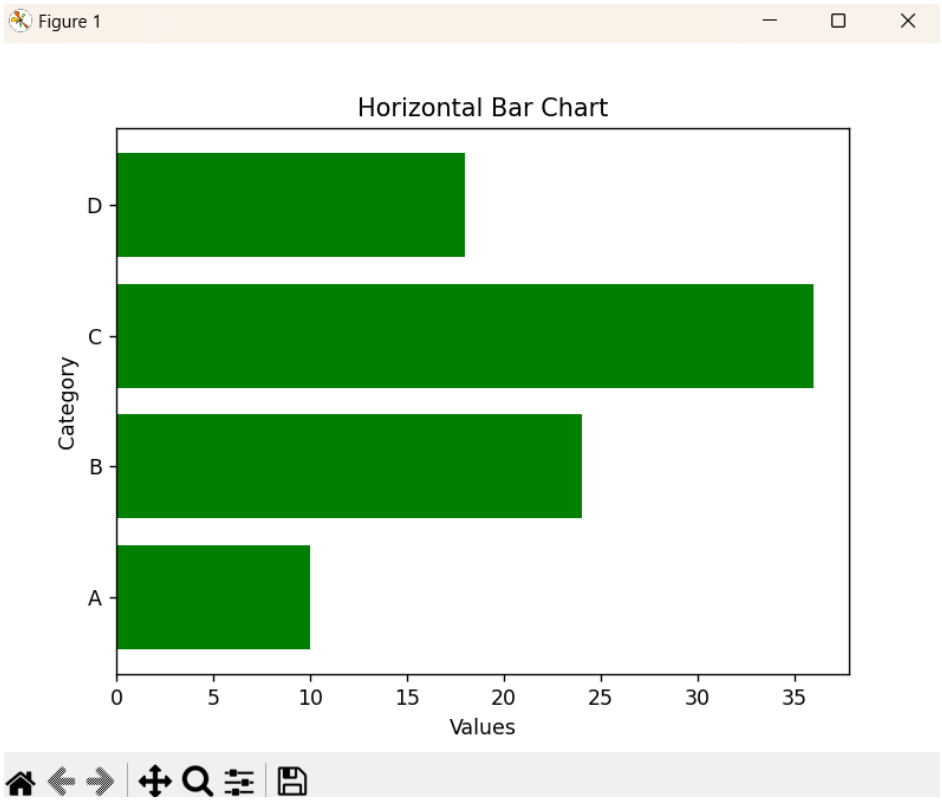


D. Horizontal Bar Chart

This example demonstrates how to create a horizontal bar chart.

```
import matplotlib.pyplot as plt

# Data for plotting categories
 = ['A', 'B', 'C', 'D'] values =
 [10, 24, 36, 18]

# Creating the horizontal bar chart
 plt.barh(categories, values, color='green')

# Adding labels and title
 plt.xlabel('Values')
 plt.ylabel('Category')
 plt.title('Horizontal Bar Chart')

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

Output:

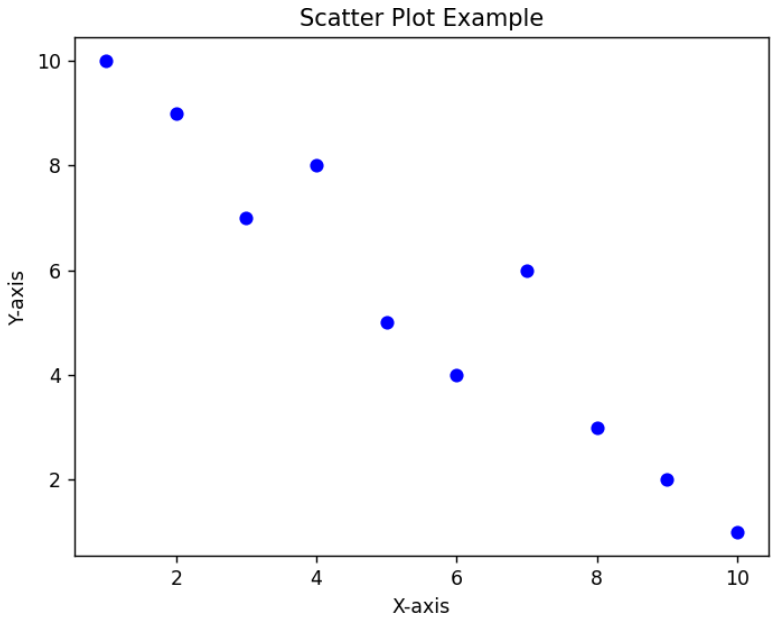


E. Scatter Plot

This program shows how to create a scatter plot to visualize relationships between two variables.

```
import matplotlib.pyplot as plt

# Data for plotting
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [10, 9, 7, 8, 5, 4, 6, 3, 2, 1]

# Creating the scatter plot
plt.scatter(x, y, color='blue', marker='o')

# Adding labels and title
 plt.xlabel('X-axis')
 plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')

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

Output:

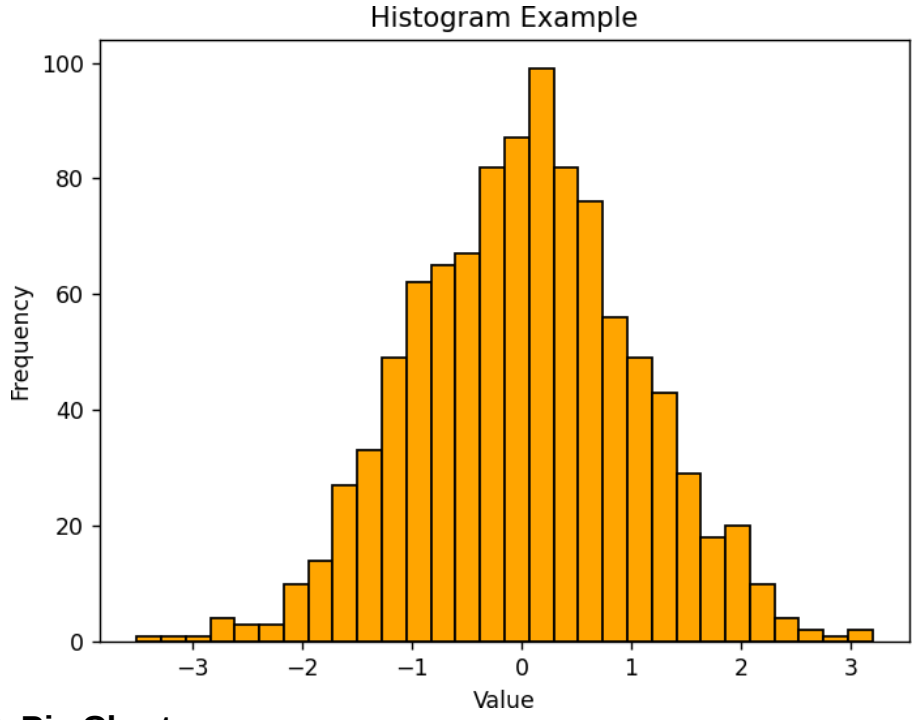


F. Histogram

Create a histogram to visualize the distribution of data.

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

# Generating random data data
 = np.random.randn(1000)

# Creating the histogram
plt.hist(data, bins=30, color='orange', edgecolor='black')

# Adding labels and title
 plt.xlabel('Value')
 plt.ylabel('Frequency')
 plt.title('Histogram Example')

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

Output:

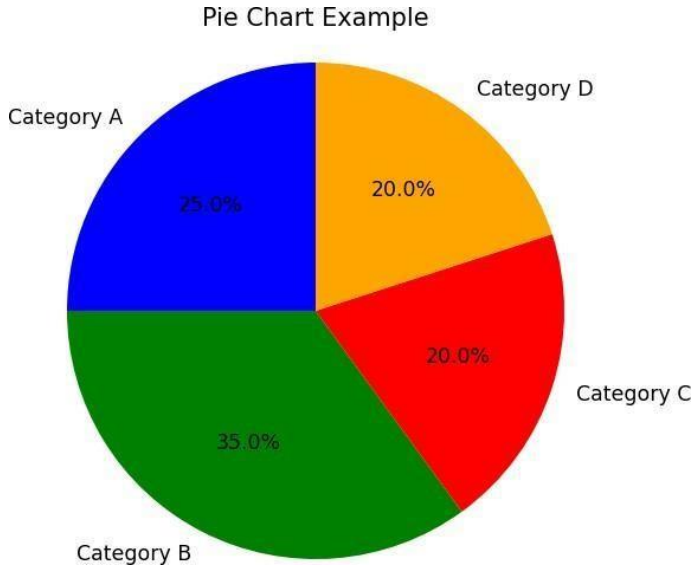


G. Pie Chart

This program demonstrates how to create a pie chart to represent the proportion of categories.

```
import matplotlib.pyplot as plt

# Data for plotting
labels = ['Category A', 'Category B', 'Category C', 'Category D']
 sizes = [25, 35, 20, 20]
colors = ['blue', 'green', 'red', 'orange']

# Creating the pie chart
plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90)

# Adding title
plt.title('Pie Chart Example')

# Equal aspect ratio ensures that pie is drawn as a circle
 plt.axis('equal')

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

Output:

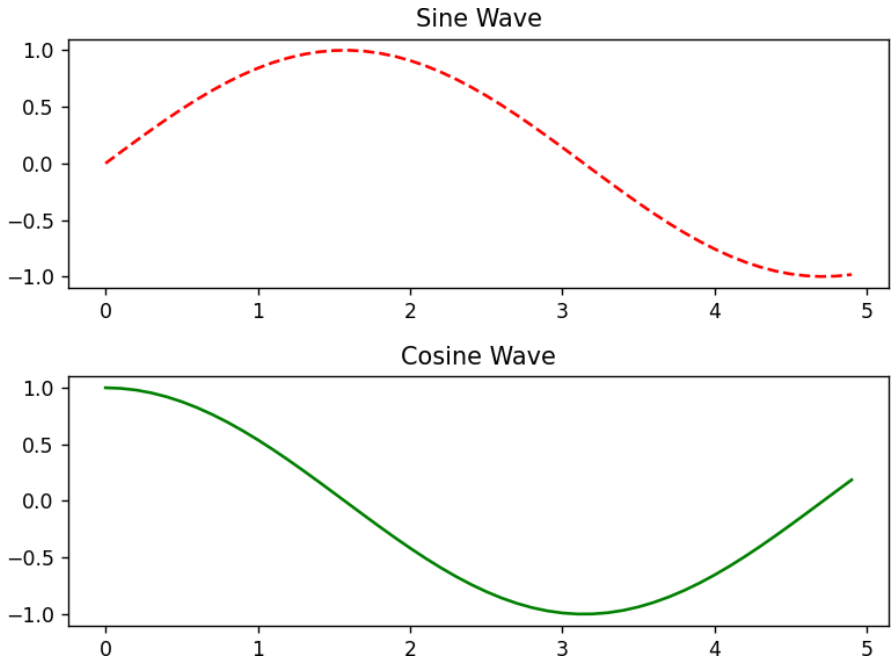


H. Subplots (Multiple Plots)

This example shows how to create multiple subplots in the same figure.

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

# Data for plotting
x = np.arange(0, 5, 0.1)
 y1 = np.sin(x)
y2 = np.cos(x)

# Creating subplots
 plt.subplot(2, 1, 1)
 plt.plot(x, y1, 'r--')
 plt.title('Sine Wave')

plt.subplot(2, 1, 2)
 plt.plot(x, y2, 'g-')
 plt.title('Cosine Wave')

# Adjust layout
 plt.tight_layout()

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

Output:

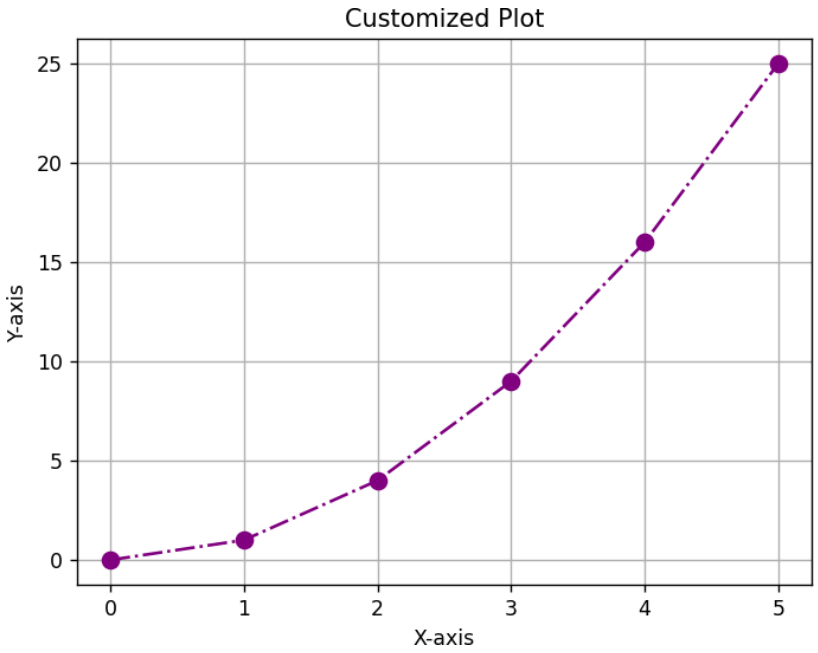


I. Customizing Plot Appearance

Customize the plot appearance, including grid lines, color, and style.

```
import matplotlib.pyplot as plt

# Data for plotting x
 = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

# Creating the plot
plt.plot(x, y, color='purple', linestyle='-.', marker='o', markersize=8)

# Adding labels and title
 plt.xlabel('X-axis')
 plt.ylabel('Y-axis')
 plt.title('Customized Plot')

# Adding grid lines
 plt.grid(True)

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

Output:

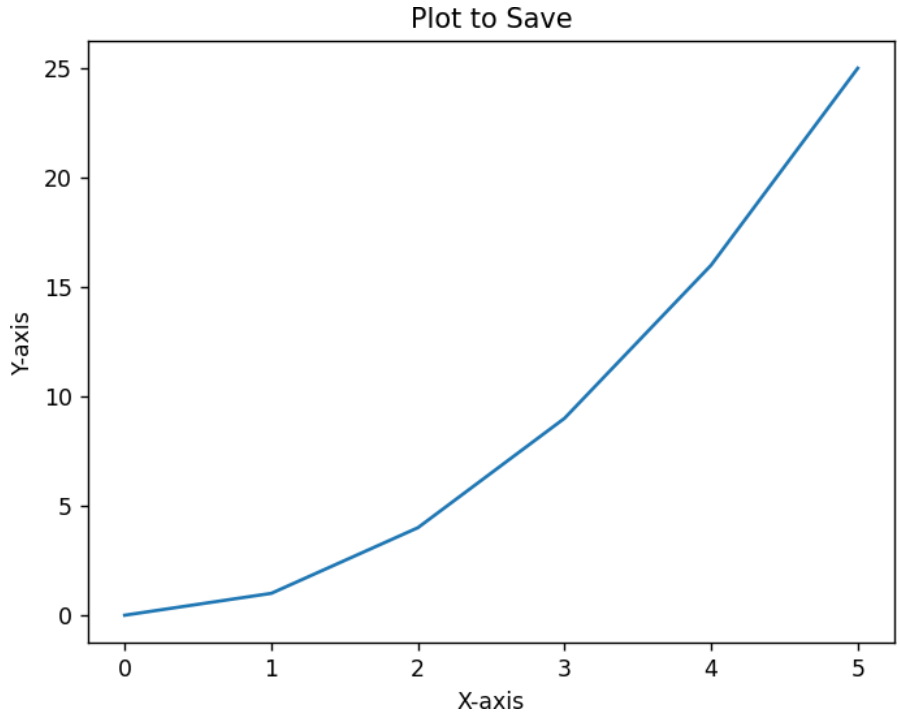


J. Saving a Plot to a File

This program demonstrates how to save a plot to a file (e.g., PNG, PDF).

```
import matplotlib.pyplot as plt

# Data for plotting x
 = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

# Creating the plot
 plt.plot(x, y)

# Adding labels and title
 plt.xlabel('X-axis')
 plt.ylabel('Y-axis')
 plt.title('Plot to Save')

# Saving the plot to a PNG file
 plt.savefig('plot_example.png')

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

Output:

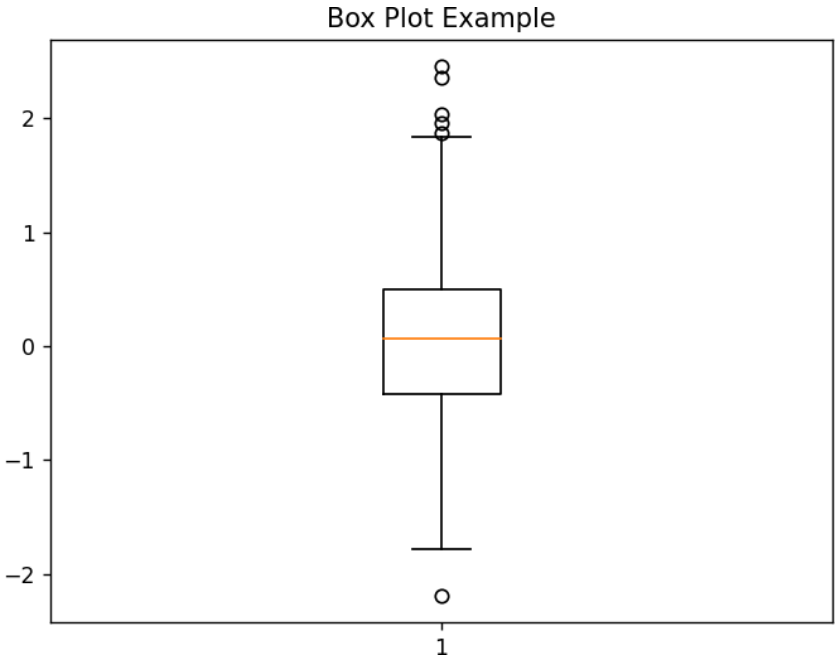


K. Box Plot

Create a box plot to display data distribution and outliers.

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

# Generating random data data
 = np.random.randn(100)

# Creating the box plot
 plt.boxplot(data)

# Adding title
plt.title('Box Plot Example')

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

Output:

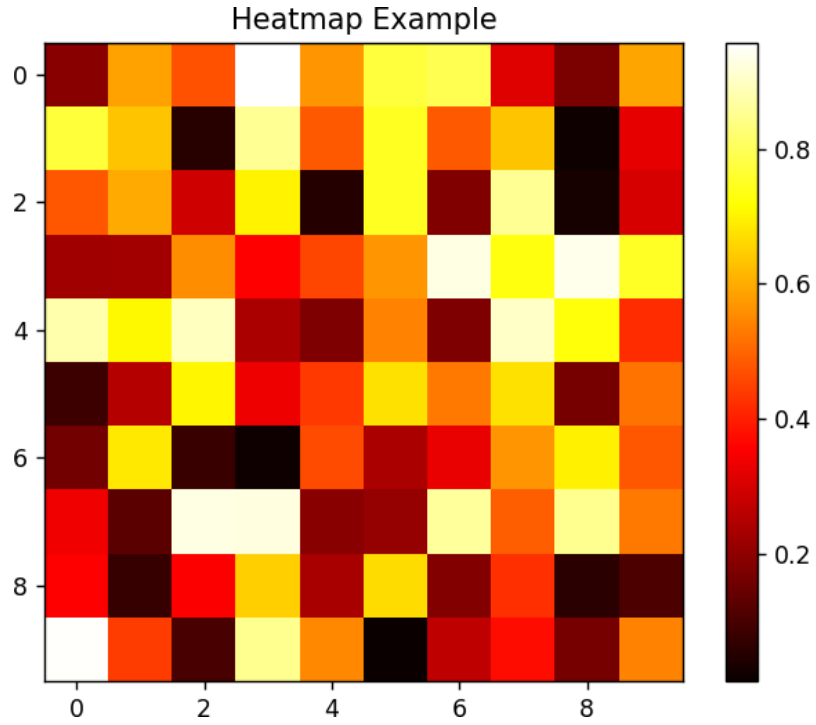


L. Heatmap

This program demonstrates how to create a heatmap to visualize matrix-like data.

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

# Generating data for the heatmap
 data = np.random.rand(10, 10)

# Creating the heatmap
plt.imshow(data, cmap='hot', interpolation='nearest')

# Adding a color bar
 plt.colorbar()

# Adding title
 plt.title('Heatmap Example')

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

Output:

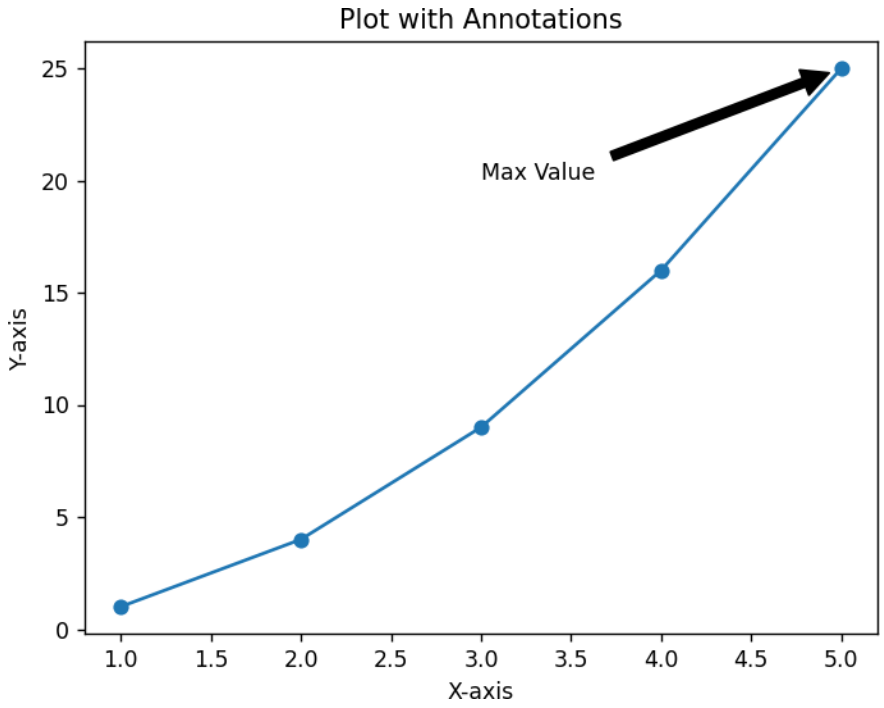


M. Annotations on a Plot

Add annotations to highlight specific points in the plot.

```
import matplotlib.pyplot as plt

# Data for plotting
 x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# Creating the plot
 plt.plot(x, y, marker='o')

# Adding annotations
 plt.annotate('Max Value', xy=(5, 25), xytext=(3, 20),
             arrowprops=dict(facecolor='black', shrink=0.05))

# Adding labels and title
 plt.xlabel('X-axis')
 plt.ylabel('Y-axis')
plt.title('Plot with Annotations')

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

Output:


Plot with Annotations
Max Value
Y-axis
X-axis
0
5
10
15
20
25
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0