1.Python Basics

1. Write a program to print your name.

A:

Python program to print your name:

Print(“Your Name”)

2. Write a program for a Single line comment and multi-line comments

A:

```
# This is a single-line comment

"""
This is a multi-line comment.
It can span multiple lines.
Used for documentation and explanations.
"""

print("Comments in Python!")  # This prints a message
```

3. Define variables for different Data Types int, Boolean, char, float, double and print on the Console.

A:

```
# Defining variables of different data types
integer_var = 10         # Integer
boolean_var = True       # Boolean
char_var = 'A'           # Character (Python does not have a char type, so we use a string of length 1)
float_var = 10.5         # Float
double_var = 20.123456789  # Double (Python uses float for double-precision)
```

```
# Printing the variables
print("Integer:", integer_var)
print("Boolean:", boolean_var)
print("Character:", char_var)
print("Float:", float_var)
print("Double:", double_var)
```

4. Define the local and Global variables with the same name and print both variables and understand the scope of the variables.

A:

```
# Global variable
num = 10

def my_function():
    # Local variable (same name as the global variable)
    num = 20
    print("Inside the function, Local num:", num)  # Prints the local variable

# Calling the function
my_function()

# Printing the global variable
print("Outside the function, Global num:", num)
```

2.Operators

1. Write a function for arithmetic operators(+,-,*,/)

A:

Python function demonstrating arithmetic operators (+, -, *, /):

```
def arithmetic_operations(a, b):
    print("Addition:", a + b)       # Addition (+)
    print("Subtraction:", a - b)    # Subtraction (-)
    print("Multiplication:", a * b) # Multiplication (*)
    print("Division:", a / b)       # Division (/)

# Example usage
arithmetic_operations(10, 5)
```

2. Write a method for increment and decrement operators(++, --)

A:

Python function demonstrating increment and decrement operations:

python

```
def increment_decrement():
    num = 10  # Initial value

    # Increment (Equivalent to num++)
    num += 1
    print("After Increment:", num)

    # Decrement (Equivalent to num--)
    num -= 1
    print("After Decrement:", num)
```

```
# Call the function
increment_decrement()
```

3. Write a program to find the two numbers equal or not.

A:

Python program to check whether two numbers are equal or not:

python

```
def check_equality(num1, num2):
    if num1 == num2:
        print("The numbers are equal.")
    else:
        print("The numbers are not equal.")

# Example usage
check_equality(10, 10)  # Equal case
check_equality(10, 5)   # Not equal case
```

4. Program for relational operators (<,<==, >, >==)

A:

Python program demonstrating **relational operators** (<, <=, >, >=):

python

```
def relational_operators(a, b):
    print(f"{a} < {b}  :", a < b)   # Less than
    print(f"{a} <= {b} :", a <= b)  # Less than or equal to
    print(f"{a} > {b}  :", a > b)   # Greater than
```

```
    print(f"{a} >= {b} :", a >= b)  # Greater than or equal to


# Example usage
relational_operators(10, 20)
relational_operators(15, 15)
relational_operators(30, 10)
```

5. Print the smaller and larger number

A:

Python program to find and print the **smaller** and **larger** number between two given numbers:

```
def find_smaller_larger(a, b):
    smaller = min(a, b)  # Find the smaller number
    larger = max(a, b)   # Find the larger number


    print("Smaller number:", smaller)
    print("Larger number:", larger)


# Example usage
find_smaller_larger(10, 20)
find_smaller_larger(50, 15)
find_smaller_larger(30, 30)  # If both numbers are equal
```

3.Loops

1. Write a program to print “Bright IT Career” ten times using for loop

A:

Python program to print **"Bright IT Career"** ten times using a for loop:

```
# Program to print "Bright IT Career" 10 times using a for loop
for i in range(10):
    print("Bright IT Career")
```

2. Write a java program to print 1 to 20 numbers using the while loop

A:

Java program to print numbers from **1 to 20** using a while loop:

```
java
public class NumberPrinter {
    public static void main(String[] args) {
        int num = 1;  // Initialize the starting number

        while (num <= 20) {  // Loop until num reaches 20
            System.out.println(num);
            num++;  // Increment num by 1
        }
    }
}
```

Explanation:

- int num = 1; → Initializes the number at **1**.
- while (num <= 20) → Runs the loop **until num reaches 20**.

- System.out.println(num); → Prints the current number.
- num++; → Increments num by 1 in each iteration.

This will output:

1

2

3

...

20

3. Program to equal operator and not equal operators

A:

Python program demonstrating the **equal (==)** and **not equal (!=)** operators:

python

```
def compare_numbers(a, b):
    if a == b:
        print(f"{a} and {b} are equal.")
    else:
        print(f"{a} and {b} are not equal.")

# Example usage
compare_numbers(10, 10)  # Equal case
compare_numbers(15, 20)  # Not equal case
```

4. Write a program to print the odd and even numbers.

A:

Python program to print **odd and even numbers** within a given range:

python

```
def print_odd_even(start, end):
    print("Even Numbers:")
    for num in range(start, end + 1):
        if num % 2 == 0:
            print(num, end=" ")

    print("\nOdd Numbers:")
    for num in range(start, end + 1):
        if num % 2 != 0:
            print(num, end=" ")

# Example usage: Print odd and even numbers from 1 to 20
print_odd_even(1, 20)
```

Explanation:

- The function print_odd_even(start, end) takes a **range of numbers**.
- The **first loop** prints **even numbers** (num % 2 == 0).
- The **second loop** prints **odd numbers** (num % 2 != 0).
- end=" " is used to print numbers in a single line.

Example Output:

Even Numbers:

2 4 6 8 10 12 14 16 18 20

Odd Numbers:

1 3 5 7 9 11 13 15 17 19

5. Write a program to print largest number among three numbers

A:

Python program to find and print the **largest** number among three numbers:

python

```
def find_largest(a, b, c):
    largest = max(a, b, c)  # Using max() function to find the largest number
    print("The largest number is:", largest)


# Example usage
find_largest(10, 25, 15)  # Output: 25
find_largest(50, 30, 40)  # Output: 50
find_largest(5, 5, 5)     # Output: 5 (All equal case)
```

Explanation:

- The max(a, b, c) function returns the largest of the three numbers.
- The function prints the largest number.

Example Output:

```
The largest number is: 25
The largest number is: 50
The largest number is: 5
```

6. Write a program to print even number between 10 and 20 using while

A:

Python program to print even numbers between **10 and 20** using a while loop:

```
num = 10  # Start from 10

while num <= 20:  # Loop until 20
    if num % 2 == 0:  # Check if even
        print(num)
    num += 1  # Increment by 1
```

Explanation:

- Start with num = 10.
- Use a while loop to run until num <= 20.
- Check if the number is **even** using num % 2 == 0.
- If true, print the number.
- Increment num by 1 in each iteration.

Output:

10

12

14

16

18

20

7. Write a program to print 1 to 10 using the do-while loop statement.

A:

Python Program to Print Numbers from 1 to 10 Using a Simulated Do-While Loop:

```
num = 1  # Initialize the number


while True:  # Simulating a do-while loop
    print(num)
    num += 1
    if num > 10:  # Exit condition
        break
```

Explanation:

1. **while True:** creates an **infinite loop** (simulating a do-while loop).
2. The number is printed **before** checking the condition (ensuring at least one execution).

3. num is incremented by 1.

4. If num > 10, the loop **exits using break**.

Expected Output:

1

2

3

4

5

6

7

8

9

10

8. Write a program to find Armstrong number or not

A:

Python program to check whether a given number is an **Armstrong number** or not:

python

```
def is_armstrong(number):
    sum_of_digits = 0
    temp = number
    num_digits = len(str(number))  # Count the number of digits

    while temp > 0:
        digit = temp % 10  # Extract last digit
        sum_of_digits += digit ** num_digits  # Add the power of the digit
        temp //= 10  # Remove last digit
```

```
    if sum_of_digits == number:
        print(f"{number} is an Armstrong number.")
    else:
        print(f"{number} is not an Armstrong number.")

# Example usage
is_armstrong(153)  # Armstrong number (3-digit)
is_armstrong(9474) # Armstrong number (4-digit)
is_armstrong(123)
```

9. Write a program to find the prime or not.

A:

Python program to check whether a given number is prime or not:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Input from user
num = int(input("Enter a number: "))

# Check and display result
if is_prime(num):
    print(f"{num} is a prime number.")
```

```
else:
    print(f"{num} is not a prime number.")
```

Explanation:

- If the number is **less than or equal to 1**, it is not prime.
- We iterate from 2 to sqrt(n), checking divisibility.
- If any divisor is found, the number is **not prime**; otherwise, it is **prime**.

10. Write a program to palindrome or not.

A:

Python program to check whether a given string or number is a palindrome:

```
def is_palindrome(s):
    return s == s[::-1]  # Reverse the string and compare

# Input from user
text = input("Enter a string or number: ")

# Check and display result
if is_palindrome(text):
    print(f"{text} is a palindrome.")
else:
    print(f"{text} is not a palindrome.")
```

Explanation:

- A **palindrome** is a word, phrase, or number that reads the same **forward and backward**.
- We reverse the input using s[::-1] and check if it matches the original.
- **Example Runs:**
- `vbnet`

Input: madam

Output: madam is a palindrome.

Input: 12321

Output: 12321 is a palindrome.

Input: hello

Output: hello is not a palindrome.

11.Program to check whether a number is EVEN or ODD using switch

A:

Python program to check whether a number is even or odd using a switch-like approach:

python

```
def check_even_odd(n):
    switch = {
        0: "Even",
        1: "Odd"
    }
    return switch[n % 2]  # Using remainder as key

# Input from user
num = int(input("Enter a number: "))

# Check and display result
print(f"{num} is {check_even_odd(num)}.")
```

Explanation:

- The dictionary {0: "Even", 1: "Odd"} acts as a switch case.
- n % 2 gives **0 for even** and **1 for odd**.
- The result is retrieved from the dictionary and displayed.

Example Runs:

Input: 10

Output: 10 is Even.

Input: 7

Output: 7 is Odd.

12. Print gender (Male/Female) program according to given M/F using switch

A:

Python does not have a built-in switch statement like C or Java, but we can use a dictionary-based switch-case approach to achieve the same functionality**.**

Python Program to Print Gender Based on 'M' or 'F':

python

```
def get_gender(code):
    switch = {
        'M': "Male",
        'F': "Female"
    }
    return switch.get(code.upper(), "Invalid Input")  # Handle uppercase & invalid input


# Input from user
gender_code = input("Enter 'M' for Male or 'F' for Female: ")
```

```
# Check and display result
print(f"Gender: {get_gender(gender_code)}")
```

Explanation:

- We use a **dictionary (switch)** to map:
  - 'M' → "Male"
  - 'F' → "Female"
- get(code.upper(), "Invalid Input"):
  - Converts input to uppercase to handle both 'm' and 'M'.
  - Returns "Invalid Input" if the user enters something else.

Example Runs:

Input: M

Output: Gender: Male

Input: f

Output: Gender: Female

Input: X

Output: Gender: Invalid Input

4.Arrays

1. Write a function to add integer values of an array

A:

Python function to add the integer values of an array:

python

```
def sum_of_array(arr):
    return sum(arr)

# Example usage
numbers = [1, 2, 3, 4, 5]
print(sum_of_array(numbers))  # Output: 15
```

2. Write a function to calculate the average value of an array of integers

A:

Python function to calculate the average value of an array of integers:

```
def average(arr):
    if not arr:  # Check if the array is empty
        return 0
    return sum(arr) / len(arr)

# Example usage:
numbers = [10, 20, 30, 40, 50]
print(average(numbers))  # Output: 30.0
```

This function:

- Checks if the array is empty to avoid division by zero.
- Uses sum(arr) to get the total sum of elements.
- Divides by len(arr) to get the average.

3. Write a program to find the index of an array element

A:

Python program to find the index of an element in an array:

python

```
def find_index(arr, target):
    try:
        return arr.index(target)  # Returns the index of the target element
    except ValueError:
        return -1  # Returns -1 if the element is not found


# Example usage:
numbers = [10, 20, 30, 40, 50]
target = 30
print(find_index(numbers, target))  # Output: 2
```

Explanation:

- The index() method is used to find the first occurrence of the target element.
- If the element is not found, a ValueError occurs, which we catch and return -1.

5. Write a function to test if array contains a specific value

A:

Python function to remove a specific element from an array:

```
def remove_element(arr, target):
    return [x for x in arr if x != target]  # Creates a new list without the target element


# Example usage:
numbers = [10, 20, 30, 40, 50, 30]
```

```
print(remove_element(numbers, 30))  # Output: [10, 20, 40, 50]
```

Explanation:

- Uses list comprehension to filter out occurrences of target.
- Returns a new list without modifying the original array.
- If target is not found, the original array remains unchanged.

6. Write a function to copy an array to another array

A:

Python function to copy an array to another array:

```
def copy_array(arr):
    return arr[:]  # Creates a shallow copy using slicing


# Example usage:
original = [10, 20, 30, 40, 50]
copied = copy_array(original)


print(copied)  # Output: [10, 20, 30, 40, 50]
```

Alternative Methods:

1. **Using list()**

python

CopyEdit

```
copied = list(original)
```

2. **Using .copy()**

python

CopyEdit

```
copied = original.copy()
```

3. **Using copy.deepcopy()** (for nested lists)

python

```
import copy
copied = copy.deepcopy(original)
```

7. Write a function to insert an element at a specific position in the array

A:

Python function to insert an element at a specific position in an array:

```
python
def insert_element(arr, element, position):
    arr.insert(position, element)  # Inserts element at the given position
    return arr

# Example usage:
numbers = [10, 20, 30, 40, 50]
print(insert_element(numbers, 25, 2))  # Output: [10, 20, 25, 30, 40, 50]
```

Explanation:

- The .insert(index, value) method places value at the specified index in arr.
- It shifts the existing elements to the right without overwriting them.

8. Write a function to find the minimum and maximum value of an array

A:

Python function to find the minimum and maximum values in an array:

```
python
def find_min_max(arr):
    if not arr:  # Check if the array is empty
        return None, None
    return min(arr), max(arr)  # Returns a tuple (min_value, max_value)

# Example usage:
numbers = [10, 20, 5, 40, 50]
```

```
min_val, max_val = find_min_max(numbers)
print(f"Min: {min_val}, Max: {max_val}")  # Output: Min: 5, Max: 50
```

Explanation:

- Uses min() to find the smallest value.
- Uses max() to find the largest value.
- Returns None, None if the array is empty to prevent errors.

9. Write a function to reverse an array of integer values

A:

Python function to reverse an array of integers:

```
def reverse_array(arr):
    return arr[::-1]  # Uses slicing to reverse the array

# Example usage:
numbers = [10, 20, 30, 40, 50]
print(reverse_array(numbers))  # Output: [50, 40, 30, 20, 10]
```

Alternative Methods:

1. **Using .reverse() (Modifies the original array)**

```
python
CopyEdit
numbers.reverse()
print(numbers)  # Output: [50, 40, 30, 20, 10]
```

2. **Using reversed() (Creates a new reversed list)**

```
python
CopyEdit
reversed_numbers = list(reversed(numbers))
print(reversed_numbers)  # Output: [50, 40, 30, 20, 10]
```

10. Write a function to find the duplicate values of an array

A:

Python function to find duplicate values in an array:

python

```
def find_duplicates(arr):
    seen = set()
    duplicates = set()

    for num in arr:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)

    return list(duplicates)  # Convert set to list for output

# Example usage:
numbers = [10, 20, 30, 40, 50, 30, 10, 20, 60]
print(find_duplicates(numbers))  # Output: [10, 20, 30]
```

Explanation:

- Uses a set (seen) to track elements that have been encountered.
- Uses another set (duplicates) to store repeated elements.
- Converts the result into a list for easy readability.

11. Write a program to find the common values between two arrays

A:

Python program to find the common values between two arrays:

```
def find_common_values(arr1, arr2):
```

```
    return list(set(arr1) & set(arr2))  # Finds intersection of two sets


# Example usage:
array1 = [10, 20, 30, 40, 50]
array2 = [30, 50, 70, 90, 10]
print(find_common_values(array1, array2))  # Output: [10, 50, 30]
```

Explanation:

- Converts both arrays into sets.
- Uses the & operator to find common elements (set intersection).
- Converts the result back to a list for easy readability.

13. Write a method to remove duplicate elements from an array

A:

Python function to remove duplicate elements from an array while preserving the original order:

```
def remove_duplicates(arr):
    seen = set()
    return [x for x in arr if not (x in seen or seen.add(x))]


# Example usage:
numbers = [10, 20, 30, 40, 50, 30, 10, 20, 60]
print(remove_duplicates(numbers))  # Output: [10, 20, 30, 40, 50, 60]
```

Explanation:

- Uses a set (seen) to track encountered elements.
- The list comprehension keeps only the first occurrence of each element, ensuring order is preserved.

14. Write a method to find the second largest number in an array

A:

Python function to find the second largest number in an array:

```
def second_largest(arr):
    unique_numbers = list(set(arr))  # Remove duplicates
    if len(unique_numbers) < 2:
        return None  # Return None if there is no second largest number
    unique_numbers.sort(reverse=True)  # Sort in descending order
    return unique_numbers[1]  # Return the second largest number


# Example usage:
numbers = [10, 20, 30, 40, 50, 30, 10]
print(second_largest(numbers))  # Output: 40
```

Explanation:

- The first method:
  - Removes duplicates using set().
  - Sorts in descending order and picks the second element.
- The second method:
  - Scans the array once (O(n) complexity) to find the two largest numbers efficiently.

15. Write a method to find the second largest number in an array

A:

Python function to find the second largest number in an array efficiently:

Optimized Approach (O(n) time complexity, No Sorting)

```
def second_largest(arr):
    first, second = float('-inf'), float('-inf')  # Initialize variables with negative infinity


    for num in arr:
        if num > first:  # Found a new largest number
            second, first = first, num
```

```
        elif first > num > second:  # Found a new second largest number
            second = num

    return second if second != float('-inf') else None  # Return None if no second largest number exists

# Example usage:
numbers = [10, 20, 30, 40, 50, 30, 10]
print(second_largest(numbers))  # Output: 40
```

Explanation:

1. **First method (Efficient - O(n)):**
    - Iterates through the array once.
    - Keeps track of the **largest (first)** and **second largest (second)** numbers.
    - Avoids sorting, making it more efficient.
2. **Second method (Sorting - O(n log n)):**
    - Removes duplicates.
    - Sorts the array in descending order.
    - Picks the second element.

16. Write a method to find number of even number and odd numbers in an array

A:

Python function to count the number of even and odd numbers in an array:

python

Explanation:

- Uses a generator expression to count even numbers efficiently.
- Odd count is derived by subtracting even count from the total length.
- Returns a tuple (even_count, odd_count).

17. Write a function to get the difference of largest and smallest value

A:

ython function to check if an array contains both **12** and **23**:

python

```
def contains_elements(arr, elem1=12, elem2=23):
    return elem1 in arr and elem2 in arr  # Check if both elements exist


# Example usage:
numbers = [10, 12, 15, 23, 30, 45]
print(contains_elements(numbers))  # Output: True


numbers2 = [10, 15, 30, 45]
print(contains_elements(numbers2))  # Output: False
```

Explanation:

- Uses the in operator to check for both elements.
- Returns True if both are present, otherwise False.
- Default values are **12** and **23**, but you can modify them.

18. Write a program to remove the duplicate elements and return the new array

A:

Python function to remove duplicate elements from an array and return a new array:

python

```
def remove_duplicates(arr):
    return list(dict.fromkeys(arr))  # Preserves order while removing duplicates
```

Explanation:

- The first method uses dict.fromkeys() to remove duplicates while keeping the original order.
- The second method uses a set() to track seen elements and remove duplicates effi

Example usage:

numbers = [10, 20, 30, 40, 50, 30, 10, 20, 60]

print(remove_duplicates(numbers)) # Output: [10, 20, 30, 40, 50, 60]

5.Static

1. Define a static variable and access that through a class

A:

a **static variable** (also known as a **class variable**) is shared across all instances of a class. You can define it inside a class and access it using the class name or an instance.

Example: Defining and Accessing a Static Variable

```
class Example:
    static_var = 100  # Static (class) variable

    def __init__(self, value):
        self.instance_var = value  # Instance variable

# Accessing static variable using class name
print(Example.static_var)  # Output: 100

# Creating instances and accessing static variable
obj1 = Example(10)
obj2 = Example(20)

print(obj1.static_var)  # Output: 100 (shared among all instances)
print(obj2.static_var)  # Output: 100

# Modifying static variable via class
Example.static_var = 200

print(obj1.static_var)  # Output: 200 (affects all instances)
print(obj2.static_var)  # Output: 200
```

```
print(Example.static_var)  # Output: 200
```

Explanation:

- static_var is a **class variable** (shared across all instances).
- It is accessed using **ClassName.static_var** or through instances.
- Modifying static_var via the **class** changes it for all instances.

2. Define a static variable and access that through a instance

A:

```
class MyClass:
    # Static variable
    static_var = "I am a static variable"

    def __init__(self, name):
        self.name = name  # Instance variable

# Accessing the static variable through an instance
obj1 = MyClass("Object 1")
obj2 = MyClass("Object 2")

print(obj1.static_var)  # Accessing static variable through an instance
print(obj2.static_var)  # Accessing static variable through another instance

# Accessing the static variable through the class name (preferred way)
print(MyClass.static_var)
```

Output:

```
I am a static variable
I am a static variable
I am a static variable
```

Key Points:

- **Static variables are shared** across all instances of a class.
- **They are accessed using both the class name (MyClass.static_var)** and an instance (obj1.static_var), though using the class name is the preferred approach.
- **Modifying a static variable via the class affects all instances**, but modifying it via an instance creates an instance-specific attribute.

3. Define a static variable and change within the instance

A:

Example: Changing a Static Variable from an Instance

python

```
class MyClass:
    # Static variable (shared among all instances)
    static_var = "I am a static variable"

    def __init__(self, name):
        self.name = name  # Instance variable

    def change_static_var(self, new_value):
        MyClass.static_var = new_value  # Modifying the static variable through the class

# Creating instances
obj1 = MyClass("Object 1")
obj2 = MyClass("Object 2")

# Accessing static variable before modification
print("Before modification:")
print(obj1.static_var)  # I am a static variable
```

```
print(obj2.static_var)  # I am a static variable

# Modifying static variable through obj1
obj1.change_static_var("Static variable changed!")

# Accessing static variable after modification
print("\nAfter modification:")
print(obj1.static_var)  # Static variable changed!
print(obj2.static_var)  # Static variable changed!
print(MyClass.static_var)  # Static variable changed!
```

Output:

```
Before modification:
I am a static variable
I am a static variable

After modification:
Static variable changed!
Static variable changed!
Static variable changed!
```

Key Takeaways:

1. **Static variables are shared** among all instances.
2. **Modifying the static variable via the class (MyClass.static_var = new_value)** updates it for all instances.
3. **Using an instance method (change_static_var) to modify it works because it references MyClass.static_var**.
4. Define a static variable and change within the class

A:

static variable (class variable) within the **class itself**, we should always reference it using the class name. This ensures that the change applies to all instances.

Example: Changing a Static Variable within the Class

```
class MyClass:
    # Static variable (shared among all instances)
    static_var = "I am a static variable"

    @classmethod
    def change_static_var(cls, new_value):
        cls.static_var = new_value  # Modifying the static variable using the class method

# Creating instances
obj1 = MyClass()
obj2 = MyClass()

# Accessing static variable before modification
print("Before modification:")
print(obj1.static_var)  # I am a static variable
print(obj2.static_var)  # I am a static variable
print(MyClass.static_var)  # I am a static variable

# Modifying the static variable using the class method
MyClass.change_static_var("Static variable changed!")

# Accessing static variable after modification
print("\nAfter modification:")
```

```
print(obj1.static_var)  # Static variable changed!
print(obj2.static_var)  # Static variable changed!
print(MyClass.static_var)  # Static variable changed!
```

Output:

```
Before modification:
I am a static variable
I am a static variable
I am a static variable

After modification:
Static variable changed!
Static variable changed!
Static variable changed!
```

6.Strings

1. Different ways creating a string

A:

here are multiple ways to create a string. Here are some of the most common methods:

but two things are giving

1. Singl Quotes.
2. Double Quotes.

1. Using Single or Double Quotes

The simplest way to create a string is by using single (' ') or double (" ") quotes.

python

```
str1 = 'Hello, World!'   # Single quotes
str2 = "Hello, World!"   # Double quotes

print(str1)  # Hello, World!
print(str2)  # Hello, World!
```

2. Using Triple Quotes (''' ''' or """ """)

Triple quotes allow multi-line strings.

python

```
str3 = '''This is
a multi-line
string.'''

str4 = """This is also
a multi-line
string."""
```

print(str3)

print(str4)

Output:

pgsql

This is

a multi-line

string.

This is also

a multi-line

string.

2. Concatenating two strings using + operator

A:

Concatenating two strings using the + operator in Python is simple and efficient. The + operator joins two or more strings into one.

Example 1: Basic String Concatenation

python

```
str1 = "Hello"
str2 = "World"

result = str1 + " " + str2  # Adding a space between words
print(result)  # Output: Hello World
```

3. Finding the length of the string

A:

Finding the Length of a String in Python

To find the length of a string, use the built-in len() function. This function counts the number of characters, including spaces, special characters, and numbers.

Example 1: Basic Usage

python

```
text = "Hello, World!"
length = len(text)

print(length)  # Output: 13
```

4. Extract a string using Substring

A:

Extracting a Substring in Python

In Python, we can extract a **substring** using **string slicing** ([start:end]).

1. Basic Substring Extraction (Slicing)

python

```
text = "Hello, World!"

substring = text[0:5]  # Extracts characters from index 0 to 4
print(substring)  # Output: Hello
```

2. Extracting a Substring Without the End Index

python

```
text = "Programming"

substring = text[3:]  # Extract from index 3 to the end
print(substring)  # Output: gramming
```

5. Searching in strings using index()

A:

Searching in Strings Using index() in Python

The index() method in Python helps find the **position (index) of a substring** inside a string. If the substring is not found, it raises a **ValueError**.

1. Basic Usage of index()

```
text = "Welcome to Python programming"


position = text.index("Python")  # Finds the starting index of "Python"
print(position)  # Output:
```

2. Handling ValueError When Substring Not Found

```
text = "Hello, World!"


try:
    position = text.index("Python")  # "Python" is not in the string
    print(position)
except ValueError:
    print("Substring not found!")
```

3. Using index() with Start and End Parameters

```
text = "Python is fun. Python is powerful."


position = text.index("Python", 10)  # Start searching from index 10
print(position)  # Output: 16
```

4. Difference Between index() and find()

- index() raises an **error** if the substring is not found.
- find() **returns -1** instead of an error.

python

CopyEdit

```
text = "Artificial Intelligence"
```

```
print(text.find("AI"))  # Output: -1 (Not found)
print(text.index("AI"))  # Raises ValueError
```

5. Finding Multiple Occurrences Using index()

Since index() only finds the first occurrence, we can use a **loop** to find all positions.

```
text = "Python is great, and Python is easy!"

index = 0
while True:
    try:
        index = text.index("Python", index)
        print(f"Found at index: {index}")
        index += 1  # Move to the next character
    except ValueError:
        break
```

Output:

```
Found at index: 0
Found at index: 22
```

Key Takeaways:

✓ index() **returns the position** of the substring.
✓ **Raises ValueError** if the substring is not found.
✓ **Use find() instead** if you want to avoid errors.
✓ **Loop with index()** to find multiple occurrences.

6. Matching a String Against a Regular Expression With matches ()

A:

In Python, you can use the re.fullmatch() function from the re (regular expressions) module to check if a string completely matches a given pattern. It works similarly to Java's matches() method.

Syntax:

import re

result = re.fullmatch(pattern, string)

- Returns None if no match is found.
- Returns a match object if the entire string matches the pattern.

Example:

```
import re
string = "hello123"
pattern = r"[a-z]+\d+"  # Matches lowercase letters followed by digits

match = re.fullmatch(pattern, string)

if match:
   print("String matches the pattern!")
else:
   print("No match found.")
```

Output:

String matches the pattern!

7. Comparing strings

A:

Comparing Strings in Python

In Python, you can compare strings using various methods:

1. Using == and != Operators (Equality & Inequality Check)

These operators compare two strings for exact equality.

```
str1 = "hello"
str2 = "hello"
str3 = "world"

print(str1 == str2)  # True (Exact match)
print(str1 != str3)  # True (Different strings)
```

2. Using Relational Operators (<, >, <=, >=)

Strings are compared lexicographically (based on ASCII values).

```
print("apple" < "banana")  # True ('a' comes before 'b')
print("apple" > "Apple")   # True (lowercase 'a' has a higher ASCII value than uppercase 'A')
```

3. Using str.casefold() for Case-Insensitive Comparison

casefold() is more aggressive than lower() for case-insensitive comparisons.

```
str1 = "Hello"
str2 = "hello"

print(str1.casefold() == str2.casefold())  # True (ignores case differences)
```

4. Using startswith() and endswith()

These check if a string starts or ends with a specific substring.

```
text = "Python programming"
print(text.startswith("Python"))  # True
print(text.endswith("ing"))       # True
```

5. Using in Operator (Substring Check)

Checks if one string is a substring of another.

```
print("pro" in "programming")  # True
print("xyz" in "hello")        # False
```

6. Using locale.strcoll() for Locale-Sensitive Comparison

If you need locale-aware comparison, use locale:

```
import locale
locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')

print(locale.strcoll("apple", "banana"))  # Negative value (apple < banana)
```

7. Using difflib for Similarity Comparison

To find similarity between strings:

```
from difflib import SequenceMatcher

str1 = "hello"
str2 = "hallo"

similarity = SequenceMatcher(None, str1, str2).ratio()
print(similarity)  # Output: 0.8 (80% similarity)
```

8. startsWith(), endsWith() and compareTo()

A:

startswith(), endswith(), and compareTo() Equivalent in Python

In Python, the functionality of Java's startsWith(), endsWith(), and compareTo() can be achieved using built-in methods and operators.

1. startswith() – Check if a String Starts with a Substring

The startswith() method checks whether a string begins with a specific prefix.

Syntax:

string.startswith(prefix, start, end)

- prefix: The substring to check.
- start *(optional)*: The index where the check starts.
- end *(optional)*: The index where the check ends.

Example:

```
text = "Hello, world!"


print(text.startswith("Hello"))     # True
print(text.startswith("world", 7))  # True (Checks from index 7)
print(text.startswith("h"))         # False (Case-sensitive)
```

2. endswith() – Check if a String Ends with a Substring

The endswith() method checks whether a string ends with a specific suffix.

Syntax:

string.endswith(suffix, start, end)

- suffix: The substring to check.
- start *(optional)*: The index where the check starts.
- end *(optional)*: The index where the check ends.

Example:

```
text = "Python is fun!"


print(text.endswith("fun!"))   # True
print(text.endswith("Python")) # False
print(text.endswith("is", 0, 10)) # True (Checks between index 0-10)
```

3. compareTo() Equivalent in Python – Using Comparison Operators

Java's compareTo() compares two strings lexicographically:

- Returns 0 if both strings are equal.
- Returns a **negative value** if the first string is smaller.
- Returns a **positive value** if the first string is greater.

In Python, we achieve this using **comparison operators (<, >, ==)**.

Example:

```
str1 = "apple"
str2 = "banana"

print(str1 < str2)   # True ('apple' comes before 'banana')
print(str1 > str2)   # False
print(str1 == str2)  # False
```

Simulating compareTo() in Python

If you want to simulate Java's compareTo() behavior, you can use:

```
def compare_strings(s1, s2):
    if s1 == s2:
        return 0
    elif s1 < s2:
        return -1
    else:
        return 1

print(compare_strings("apple", "banana"))  # -1
print(compare_strings("grape", "apple"))   # 1
print(compare_strings("mango", "mango"))   # 0
```

9. Trimming strings with strip()

A:

Trimming Strings with strip() in Python

In Python, the strip() method is used to **remove leading and trailing whitespace** (spaces, newlines, tabs) from a string. It is equivalent to Java's trim() method.

1. strip() – Remove Leading and Trailing Whitespace

Syntax:

string.strip()

- Removes spaces, tabs (\t), and newlines (\n) from **both** ends of the string.

Example:

```
text = "   Hello, World!   "
trimmed_text = text.strip()

print(f"Original: '{text}'")
print(f"Trimmed: '{trimmed_text}'")
```

Output:

```
Original: '   Hello, World!   '
Trimmed: 'Hello, World!'
```

2. lstrip() – Remove Leading Whitespace (Left Trim)

Removes whitespace **only from the beginning** (left side) of the string.

```
text = "   Python Programming"
trimmed_text = text.lstrip()

print(f"'{trimmed_text}'")  # 'Python Programming'
```

3. rstrip() – Remove Trailing Whitespace (Right Trim)

Removes whitespace **only from the end** (right side) of the string.

```
text = "Python Programming   "
trimmed_text = text.rstrip()

print(f"'{trimmed_text}'")  # 'Python Programming'
```

4. Removing Specific Characters Using strip()

You can pass a string of characters to remove **specific leading and trailing characters**, not just whitespace.

Example:

```
text = "###Welcome to Python###"
trimmed_text = text.strip("#")

print(trimmed_text)  # 'Welcome to Python'
```

Similarly:

```
text = "---Python is fun---"
print(text.lstrip("-"))  # 'Python is fun---'
print(text.rstrip("-"))  # '---Python is fun'
```

5. Removing Multiple Characters at Once

You can specify **multiple characters** to remove.

```
text = "...,Hello!!..."
trimmed_text = text.strip(".,!")
print(trimmed_text)  # 'Hello'
```

10.Replacing characters in strings with replace()

A:

Replacing Characters in Strings with replace() in Python

In Python, the replace() method is used to **replace occurrences of a substring with another substring** in a string.

---

1. Basic Usage of replace()

Syntax:

string.replace(old, new, count)

- old: The substring to be replaced.
- new: The substring to replace it with.
- count *(optional)*: Number of occurrences to replace. If omitted, all occurrences are replaced.

Example:

```
text = "Hello, world!"
new_text = text.replace("world", "Python")

print(new_text)  # Output: "Hello, Python!"
```

2. Replacing Multiple Occurrences

If the count parameter is **not provided**, all occurrences are replaced.

```
text = "apple banana apple"
new_text = text.replace("apple", "orange")

print(new_text)  # Output: "orange banana orange"
```

3. Replacing Only a Limited Number of Occurrences

You can specify count to limit the number of replacements.

```
text = "apple apple apple"
new_text = text.replace("apple", "orange", 2)

print(new_text)  # Output: "orange orange apple"
```

4. Removing Characters (Replacing with an Empty String)

You can remove characters by replacing them with an empty string "".

```
text = "Hello!!!"
new_text = text.replace("!", "")

print(new_text)  # Output: "Hello"
```

5. Replacing Whitespace

```
text = "Hello   Python  World"
new_text = text.replace("  ", " ")  # Replace double spaces with a single space

print(new_text)  # Output: "Hello Python World"
```

6. Case Sensitivity in replace()

The replace() method is **case-sensitive**.

```
text = "Hello hello HELLO"
new_text = text.replace("hello", "hi")

print(new_text)  # Output: "Hello hi HELLO" (only lowercase "hello" is replaced)
```

7. Replacing Multiple Different Characters (Using re.sub())

If you need to replace multiple different substrings at once, use **regular expressions** with re.sub().

```
import re


text = "Hello 123, welcome!"
new_text = re.sub(r"\d", "*", text)  # Replace all digits with '*'


print(new_text)  # Output: "Hello ***, welcome!"
```

11.Splitting strings with split()

A:

Splitting Strings with split() in Python

In Python, the split() method is used to **split a string into a list of substrings** based on a specified delimiter.

1. Basic Usage of split()

Syntax:

```
string.split(separator, maxsplit)
```

- **separator** *(optional)*: The delimiter to split the string by (default is any whitespace).
- **maxsplit** *(optional)*: Maximum number of splits to perform. Default is -1 (unlimited splits).

Example: Splitting by Whitespace (Default Behavior)

```
text = "Hello world Python"
words = text.split()


print(words)  # Output: ['Hello', 'world', 'Python']
```

💡 *If no separator is provided, split() splits by any whitespace (spaces, tabs, newlines).*

2. Splitting Using a Custom Delimiter

You can specify a **character** or **substring** as the separator.

```
text = "apple,banana,grape,orange"
fruits = text.split(",")

print(fruits)  # Output: ['apple', 'banana', 'grape', 'orange']
```

3. Limiting the Number of Splits (maxsplit Parameter)

You can limit the number of times the string is split.

```
text = "one,two,three,four,five"
result = text.split(",", 2)  # Only splits twice

print(result)  # Output: ['one', 'two', 'three,four,five']
```

4. Splitting on Multiple Spaces and Newlines

Since split() by default splits on any whitespace, it removes extra spaces and newlines automatically.

```
text = "Hello   Python\nWorld"
words = text.split()

print(words)  # Output: ['Hello', 'Python', 'World']
```

5. Splitting by a Substring

You can split using a multi-character substring.

```
text = "apple--banana--grape--orange"
fruits = text.split("--")
```

```
print(fruits)  # Output: ['apple', 'banana', 'grape', 'orange']
```

6. Splitting Lines from a Multiline String (splitlines())

If you want to split a string **by newlines**, use splitlines().

```
text = """Hello
Python
World"""
lines = text.splitlines()

print(lines)  # Output: ['Hello', 'Python', 'World']
```

12.Converting integer objects to Strings

A:

Converting Integer Objects to Strings in Python

In Python, you can convert an integer to a string using several methods, such as str(), f-strings, and format().

1. Using str() (Most Common Method)

The str() function is the simplest way to convert an integer to a string.

```
num = 42
string_num = str(num)

print(string_num)  # Output: '42'
print(type(string_num))  # Output: <class 'str'>
```

2. Using f-strings (Python 3.6+)

F-strings provide a concise way to format numbers into strings.

```
num = 100
```

```
string_num = f"{num}"

print(string_num)  # Output: '100'
```

3. Using format() Method

The format() method can also convert integers to strings.

```
num = 256
string_num = "{}".format(num)

print(string_num)  # Output: '256'
```

4. Using repr() (For Exact Representation)

The repr() function returns a **string representation** of an object.

```
num = 99
string_num = repr(num)

print(string_num)  # Output: '99'
```

13.Converting to uppercase and lowercase

A:

Converting to Uppercase and Lowercase in Python

Python provides built-in string methods to convert text to **uppercase** and **lowercase**.

1. Converting to Uppercase (upper())

The upper() method converts all letters in a string to **uppercase**.

Example:

```
text = "hello world"
```

```
uppercase_text = text.upper()

print(uppercase_text)  # Output: 'HELLO WORLD'
```

2. Converting to Lowercase (lower())

The lower() method converts all letters in a string to **lowercase**.

Example:

```
text = "HELLO WORLD"
lowercase_text = text.lower()

print(lowercase_text)  # Output: 'hello world'
```

3. Capitalizing the First Letter (capitalize())

The capitalize() method converts the **first letter** to uppercase and makes the rest lowercase.

```
text = "hello WORLD"
capitalized_text = text.capitalize()

print(capitalized_text)  # Output: 'Hello world'
```

4. Title Case (title())

The title() method capitalizes the first letter of **each word**.

```
text = "python is awesome"
title_text = text.title()

print(title_text)  # Output: 'Python Is Awesome'
```

5. Swapping Case (swapcase())

The swapcase() method **swaps uppercase to lowercase and vice versa**.

```
text = "Hello World"
swapped_text = text.swapcase()

print(swapped_text)  # Output: 'hELLO wORLD'
```

7. Inheritance

A, B and C are classes A is a super class. B is a sub class of A. C is a sub class of B.

Create three methods in each class, 2 methods are specific to each class and third method (override method) should be in all three Classes A, B and C

Here's a Python program demonstrating **inheritance and method overriding** with classes **A (superclass)**, **B (subclass of A)**, and **C (subclass of B)**.

Each class contains:

- **Two unique methods** specific to that class.
- **One overridden method** (show()) that exists in all three classes.

Python Code:

```
class A:
    def method_a1(self):
        print("Method A1: Specific to Class A")

    def method_a2(self):
        print("Method A2: Specific to Class A")
```

```
    def show(self):  # Overridden method
        print("Show Method: Class A")

class B(A):  # B is a subclass of A
    def method_b1(self):
        print("Method B1: Specific to Class B")

    def method_b2(self):
        print("Method B2: Specific to Class B")

    def show(self):  # Overriding show() method from A
        print("Show Method: Class B")

class C(B):  # C is a subclass of B
    def method_c1(self):
        print("Method C1: Specific to Class C")

    def method_c2(self):
        print("Method C2: Specific to Class C")

    def show(self):  # Overriding show() method from B
        print("Show Method: Class C")

# Creating objects of each class
a = A()
b = B()
c = C()

print("Calling methods from Class A:")
```

```
a.method_a1()
a.method_a2()
a.show()  # Calls Class A's show()

print("\nCalling methods from Class B:")
b.method_b1()
b.method_b2()
b.show()  # Calls Class B's show()

print("\nCalling methods from Class C:")
c.method_c1()
c.method_c2()
c.show()  # Calls Class C's show()
```

Output:

```
Calling methods from Class A:
Method A1: Specific to Class A
Method A2: Specific to Class A
Show Method: Class A

Calling methods from Class B:
Method B1: Specific to Class B
Method B2: Specific to Class B
Show Method: Class B

Calling methods from Class C:
Method C1: Specific to Class C
Method C2: Specific to Class C
Show Method: Class C
```

Explanation:

1. **Class A**
   - Has two unique methods: method_a1() and method_a2().
   - Defines show() (which will be overridden in subclasses).
2. **Class B (Inherits from A)**
   - Has two new unique methods: method_b1() and method_b2().
   - Overrides show() from Class A.
3. **Class C (Inherits from B)**
   - Has two new unique methods: method_c1() and method_c2().
   - Overrides show() from Class B.

Create a class with main method. Create an object for each class A, B and C in main method and call every method of each class using its own object/instance.

Python program with a **main class** that creates objects for **A, B, and C**, then calls all their methods.

Python Code:

```
class A:
    def method_a1(self):
        print("Method A1: Specific to Class A")

    def method_a2(self):
        print("Method A2: Specific to Class A")
```

```
    def show(self):  # Overridden method
        print("Show Method: Class A")

class B(A):  # B is a subclass of A
    def method_b1(self):
        print("Method B1: Specific to Class B")

    def method_b2(self):
        print("Method B2: Specific to Class B")

    def show(self):  # Overriding show() method from A
        print("Show Method: Class B")

class C(B):  # C is a subclass of B
    def method_c1(self):
        print("Method C1: Specific to Class C")

    def method_c2(self):
        print("Method C2: Specific to Class C")

    def show(self):  # Overriding show() method from B
```

```
        print("Show Method: Class C")


# Main class with main method
class Main:
    @staticmethod
    def main():
        # Creating objects for each class
        obj_a = A()
        obj_b = B()
        obj_c = C()

        print("Calling methods from Class A:")
        obj_a.method_a1()
        obj_a.method_a2()
        obj_a.show()  # Calls Class A's show()

        print("\nCalling methods from Class B:")
        obj_b.method_b1()
        obj_b.method_b2()
        obj_b.show()  # Calls Class B's show()

        print("\nCalling methods from Class C:")
```

```
        obj_c.method_c1()
        obj_c.method_c2()
        obj_c.show()  # Calls Class C's show()

# Calling the main method
if __name__ == "__main__":
    Main.main()
```

Output:

```
Calling methods from Class A:
Method A1: Specific to Class A
Method A2: Specific to Class A
Show Method: Class A

Calling methods from Class B:
Method B1: Specific to Class B
Method B2: Specific to Class B
Show Method: Class B

Calling methods from Class C:
Method C1: Specific to Class C
Method C2: Specific to Class C
```

Show Method: Class C

Explanation:

1. **Classes A, B, and C**
    - Each has **two unique methods** and an **overridden show() method**.
2. **Main Class (Main)**
    - Contains a **static main() method**, which:
        - Creates instances of A, B, and C.
        - Calls all methods of each class using their respective objects.
3. **Executing Main.main()**
    - Uses if __name__ == "__main__": to ensure main() runs when the script is executed.

Call an overridden method with super class reference to B and C class's objects Runtime Polymorphism with Data Members/Instance variables, Repeat the above process only for data members

Calling an Overridden Method Using Superclass Reference (Runtime Polymorphism) & Data Members Demonstration

Concepts Used:

1. **Method Overriding & Runtime Polymorphism:**
    - A **superclass reference** can hold a **subclass object** and call overridden methods.

- The method executed is determined at **runtime** (Dynamic Method Dispatch).

2. **Runtime Polymorphism with Instance Variables:**
   - Unlike methods, **instance variables do not follow polymorphism**.
   - The **superclass reference accesses only superclass variables** (not overridden ones).

Python Code:

```
class A:
    var = "Variable from Class A"

    def show(self):  # Overridden method
        print("Show Method: Class A")

class B(A):
    var = "Variable from Class B"

    def show(self):  # Overriding method
        print("Show Method: Class B")

class C(B):
    var = "Variable from Class C"
```

```
    def show(self):  # Overriding method
        print("Show Method: Class C")

# Main Execution
if __name__ == "__main__":
    # Superclass reference pointing to subclass objects (Method Overriding)
    obj_a = A()
    obj_b = B()
    obj_c = C()

    print("Method Overriding Demonstration:")
    ref: A  # Superclass reference type

    ref = obj_b  # A reference to B's object
    ref.show()  # Calls B's show()

    ref = obj_c  # A reference to C's object
    ref.show()  # Calls C's show()

    # Runtime Polymorphism with Instance Variables
    print("\nInstance Variable Access with Superclass Reference:")
```

```
ref = obj_b
print(ref.var)  # Accesses Class A's var, NOT B's

ref = obj_c
print(ref.var)  # Accesses Class A's var, NOT C's
```

Expected Output:

Method Overriding Demonstration:

Show Method: Class B

Show Method: Class C

Instance Variable Access with Superclass Reference:

Variable from Class A

Variable from Class A

Explanation:

1. **Method Overriding (Runtime Polymorphism)**
    - When we assign ref = obj_b (or obj_c), the **method from B or C is executed** due to **dynamic dispatch**.
    - **Even though ref is of type A, the overridden method from B or C is called**.
2. **Instance Variables (NO Runtime Polymorphism)**

- ref.var always accesses the **variable from Class A**.

Instance variables do not participate in runtime

- **polymorphism**, unlike methods.

8.Access Modifiers

Private Fields and Methods in Python (Encapsulation)

In Python, **private members** are not truly private but **indicated using a double underscore (__)**. This makes them **name-mangled**, meaning they can't be accessed directly outside the class. However, they can still be accessed through **special techniques** (not recommended for general use).

Python Code:

```
class A:
    def __init__(self):
        self.__private_var = "I am Private!"  # Private field

    def __private_method(self):  # Private method
        print("This is a Private Method in Class A")

    def access_private_method(self):  # Public method to access private method
        self.__private_method()

    def get_private_var(self):  # Public method to access private variable
```

```
        return self.__private_var



class B(A):  # Subclass of A
    def try_access_private(self):
        # Trying to access private variable and method from subclass
        try:
            print(self.__private_var)  # Will raise AttributeError
        except AttributeError:
            print("Cannot access private variable directly!")

        try:
            self.__private_method()  # Will raise AttributeError
        except AttributeError:
            print("Cannot call private method directly!")



# Main Execution
if __name__ == "__main__":
    obj_a = A()

    # Accessing private fields and methods inside the main method
```

```
print("Accessing Private Members from Class A:")
print(obj_a.get_private_var())  # Accessing private variable through method
obj_a.access_private_method()   # Accessing private method through a public method

# Creating object of subclass
obj_b = B()
print("\nTrying to Access Private Members from Subclass B:")
obj_b.try_access_private()  # Attempting to access private members in subclass
```

Expected Output:

```
Accessing Private Members from Class A:
I am Private!
This is a Private Method in Class A

Trying to Access Private Members from Subclass B:
Cannot access private variable directly!
Cannot call private method directly!
```

Explanation:

1. **Private Fields (__private_var)**:

- Declared using __private_var.
- **Cannot be accessed directly** outside the class.
- Accessed using the **getter method** get_private_var().

2. **Private Method (__private_method())**:
   - Declared using __private_method().
   - **Cannot be called directly** outside the class.
   - Accessed using a **public method** access_private_method().
3. **Subclass B Cannot Access Private Members**:
   - **Direct access** to __private_var and __private_method() **fails** in B because of name mangling.
   - Trying to access them results in an AttributeError.

1.Create a class with PROTECTED fields and methods. Access these fields and methods from any other class in the same package. Also, Access the PROTECTED fields and methods from child class located in a different package Access the PROTECTED fields and methods from any class in different package.

A:

Protected Fields and Methods in Python (Encapsulation & Inheritance)

In Python, **protected members** are defined using a **single underscore (_)** before the variable or method name. This is a **convention**, not an enforced restriction. Protected members can be accessed **within the class and its subclasses**, but are still **accessible from other classes** if needed.

📌 **Scenario Overview:**

1. **Create a class (A)** with **protected fields and methods**.
2. **Access them from another class (B) in the same module**.
3. **Access them from a child class (C) in a different module/package**.
4. **Access them from a non-related class (D) in another package**.

🔹 **Step 1: Define Class A (Base Class with Protected Members)**

📌 **File: base_module.py**

```
class A:
    def __init__(self):
        self._protected_var = "I am a protected variable"

    def _protected_method(self):
        print("This is a protected method in Class A")

# Another class in the same module
class B:
    def access_protected(self):
        obj = A()
        print("Accessing protected variable from another class in the same module:", obj._protected_var)
        obj._protected_method()
```

🔹 **Step 2: Access from Child Class in Another Module (Different Package)**

📌 **File: child_package.py**

```
from base_module import A


class C(A):  # Subclass of A in a different package
    def access_protected(self):
        print("Accessing protected variable from subclass in another package:", self._protected_var)
        self._protected_method()
```

🔹 **Step 3: Access from Non-Related Class in Another Module (Different Package)**

📌 **File: unrelated_package.py**

```
from base_module import A


class D:
    def access_protected(self):
        obj = A()
        try:
            print("Trying to access protected variable from a non-related class in another package:", obj._protected_var)
            obj._protected_method()
```

```
        except AttributeError:
            print("Cannot access protected members from an unrelated class in another package!")
```

🔹 **Step 4: Main Execution to Test Accessibility**

📌 **File: main.py**

```
from base_module import B
from child_package import C
from unrelated_package import D

if __name__ == "__main__":
    print("🔹 Accessing Protected Members from Same Package:")
    obj_b = B()
    obj_b.access_protected()

    print("\n🔹 Accessing Protected Members from Child Class in Another Package:")
    obj_c = C()
    obj_c.access_protected()

    print("\n🔹 Accessing Protected Members from Unrelated Class in Another Package:")
```

```
obj_d = D()

obj_d.access_protected()
```

- **Expected Output:**

- Accessing Protected Members from Same Package:

Accessing protected variable from another class in the same module: I am a protected variable

This is a protected method in Class A

- Accessing Protected Members from Child Class in Another Package:

Accessing protected variable from subclass in another package: I am a protected variable

This is a protected method in Class A

- Accessing Protected Members from Unrelated Class in Another Package:

Trying to access protected variable from a non-related class in another package: I am a protected variable

This is a protected method in Class A

- **Key Takeaways:**

Access Scenario	Can Access Protected Members?	Why?
Same Package (Class B in base_module.py)	✅ Yes	Same module can access protected members
Subclass in Different Package (Class C in child_package.py)	✅ Yes	Inherited classes can access protected members
Unrelated Class in Different Package (Class D in unrelated_package.py)	✅ Yes	Python does **not** strictly enforce protected access

3.Create a class with PUBLIC fields and methods. Access the public methods and fields from any class in the same package or different package.

A:

In **Python**, all class attributes and methods are **public** by default, meaning they can be accessed from any class within the same module or a different module.

Steps:

1. **Create a class (Person) with public fields and methods** in module1.py.
2. **Access the public fields and methods** from another class (SamePackageAccess) in the **same module**.

3. **Create another class (DifferentPackageAccess)** in module2.py and access the public fields and methods.

1. Define the Person Class (Inside module1.py)

```
# File: module1.py
class Person:
    # Public fields
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Public method
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Accessing the class in the same module
class SamePackageAccess:
    def access_person(self):
        p = Person("John", 25)

        # Accessing public fields
        print("Name:", p.name)
```

```
        print("Age:", p.age)

        # Calling the public method
        p.display_info()

# Run test within the same module
if __name__ == "__main__":
    obj = SamePackageAccess()
    obj.access_person()
```

✅ **Since everything is public, fields and methods can be accessed directly** within the same module.

2. Access the Person Class from Another Module (module2.py)

```
# File: module2.py
# Importing the Person class from module1
from module1 import Person

class DifferentPackageAccess:
    def access_person(self):
        p = Person("Alice", 30)

        # Accessing public fields
```

```
        print("Name:", p.name)
        print("Age:", p.age)

        # Calling the public method
        p.display_info()

# Run test in different module
if __name__ == "__main__":
    obj = DifferentPackageAccess()
    obj.access_person()
```

✅ **Since Person is a public class, it can be accessed from another module using an import statement.**

Running the Code

1. **Run module1.py**

```
Name: John
Age: 25
Name: John, Age: 25
```

2. **Run module2.py**

```
Name: Alice
Age: 30
Name: Alice, Age: 30
```

9.Abstract Class

1. Create an abstract class with abstract and non-abstract methods.

A:

In Python, **abstract classes** are created using the ABC (Abstract Base Class) module from the abc package. An **abstract class** can have both **abstract methods** (methods without implementation) and **non-abstract methods** (methods with implementation).

Steps to Create an Abstract Class in Python:

1. Use ABC as the parent class.
2. Define **abstract methods** using the @abstractmethod decorator.
3. Define **non-abstract methods** (regular methods with implementation).
4. Create a **concrete subclass** that implements the abstract methods.

Example: Abstract Class with Abstract & Non-Abstract Methods

```
from abc import ABC, abstractmethod


# Abstract class
class Animal(ABC):


    # Abstract method (must be overridden by subclasses)
    @abstractmethod
```

```
    def make_sound(self):
        pass

    # Non-abstract method (has implementation)
    def sleep(self):
        print("Sleeping...")

# Concrete class that implements the abstract method
class Dog(Animal):
    def make_sound(self):
        print("Bark! Bark!")

# Concrete class that implements the abstract method
class Cat(Animal):
    def make_sound(self):
        print("Meow! Meow!")

# Creating objects of concrete classes
dog = Dog()
cat = Cat()

# Calling abstract method (implemented in subclass)
```

```
dog.make_sound()  # Output: Bark! Bark!
cat.make_sound()  # Output: Meow! Meow!

# Calling non-abstract method (inherited from abstract class)
dog.sleep()  # Output: Sleeping...
cat.sleep()  # Output: Sleeping...
```

Explanation:

✓ Abstract Class (Animal)

- Contains an **abstract method** make_sound(), which has no implementation.
- Contains a **non-abstract method** sleep(), which has a default implementation.

✓ Concrete Subclasses (Dog & Cat)

- Both **override** make_sound() and provide an implementation.
- They **inherit** sleep() from Animal and can call it without overriding.

✓ Objects Cannot Be Created from an Abstract Class

- Trying to instantiate Animal() will cause an error:

```
TypeError: Can't instantiate abstract class Animal with abstract methods make_sound
```

3. Create a sub class for an abstract class. Create an object in the child class for the abstract class and access the non-abstract methods

A:

Creating a Subclass for an Abstract Class in Python

In Python, an **abstract class** can have **abstract methods** (which must be implemented in subclasses) and **non-abstract methods** (which can be used directly).
We will:

1. **Create an abstract class (Vehicle)** with both **abstract and non-abstract methods**.
2. **Create a subclass (Car)** that implements the abstract method.
3. **Create an object of the subclass** and access both **abstract and non-abstract methods**.

Implementation:

```
from abc import ABC, abstractmethod


# Step 1: Create an Abstract Class
class Vehicle(ABC):


    # Abstract method (must be implemented by subclasses)
    @abstractmethod
    def start_engine(self):
```

```
        pass

    # Non-abstract method (inherited by subclasses)
    def stop_engine(self):
        print("Engine stopped.")

# Step 2: Create a Subclass that Implements the Abstract Method
class Car(Vehicle):

    def start_engine(self):
        print("Car engine started.")

# Step 3: Create an Object of the Subclass
car = Car()

# Access the abstract method (implemented in subclass)
car.start_engine()  # Output: Car engine started.

# Access the non-abstract method (inherited from abstract class)
car.stop_engine()   # Output: Engine stopped.
```

Explanation:

✓ Abstract Class (Vehicle)

- Has an **abstract method** start_engine(), which must be implemented by subclasses.
- Has a **non-abstract method** stop_engine(), which can be used directly.

✓ Subclass (Car)

- Implements the start_engine() method.
- Inherits stop_engine() from Vehicle without overriding it.

✓ Creating an Object of Car

- Calls **start_engine()** (abstract method, now implemented in Car).
- Calls **stop_engine()** (non-abstract method, inherited from Vehicle).

Output:

Car engine started.

Engine stopped.

4. **Create an instance for the child class in child class and call abstract methods**

A:

Creating an Instance of the Child Class in the Child Class and Calling Abstract Methods

We will:

1. **Create an abstract class (Shape)** with an **abstract method** area().

2. **Create a child class (Circle)** that **implements** the area() method.
3. **Inside the child class (Circle)**, we will **create an instance** of itself and call the abstract method.

Implementation:

```
from abc import ABC, abstractmethod


# Step 1: Create an Abstract Class
class Shape(ABC):


    @abstractmethod
    def area(self):
        pass  # Abstract method - no implementation


# Step 2: Create a Child Class that Implements the Abstract Method
class Circle(Shape):


    def __init__(self, radius):
        self.radius = radius


    # Implementing the abstract method
    def area(self):
```

```
        return 3.14 * self.radius * self.radius

    # Step 3: Creating an instance of the child class inside itself
    def calculate(self):
        circle_obj = Circle(5)  # Creating an instance of Circle
        print("Circle Area:", circle_obj.area())  # Calling the abstract method

# Step 4: Create an Object of Child Class and Call the Method
c = Circle(7)
c.calculate()
```

Explanation:

✓ Abstract Class (Shape)

- Defines the **abstract method** area(), which must be implemented by subclasses.

✓ Child Class (Circle)

- Implements the area() method.
- Has an __init__ method to initialize the radius.
- Defines calculate(), which **creates an instance of Circle inside itself** and calls area().

✓ Calling calculate()

- Creates a Circle(5) instance inside calculate().

- Calls the area() method on the new instance.

Output:

Circle Area: 78.5

4. **Create an instance for the child class in child class and call non-abstract methods.**

A:

Creating an Instance of the Child Class Inside the Child Class and Calling Non-Abstract Methods

We will:

1. **Create an abstract class (Animal)** with a **non-abstract method** sleep().
2. **Create a child class (Dog)** that implements an abstract method make_sound().
3. **Inside the child class (Dog)**, create an instance of itself and call the non-abstract method sleep().

Implementation:

```
from abc import ABC, abstractmethod


# Step 1: Create an Abstract Class

class Animal(ABC):


    @abstractmethod
```

```
    def make_sound(self):
        pass  # Abstract method

    # Non-abstract method
    def sleep(self):
        print("Animal is sleeping...")

# Step 2: Create a Child Class that Implements the Abstract Method
class Dog(Animal):

    def make_sound(self):
        print("Bark! Bark!")

    # Step 3: Creating an instance of the child class inside itself
    def action(self):
        dog_instance = Dog()  # Creating an instance of Dog inside itself
        dog_instance.sleep()  # Calling non-abstract method

# Step 4: Create an Object of the Child Class and Call the Method
d = Dog()
d.action()
```

Explanation:

✓ Abstract Class (Animal)

- **Defines an abstract method make_sound(), which must be implemented in subclasses.**
- **Defines a non-abstract method sleep(), which can be used directly by subclasses.**

✓ Child Class (Dog)

- **Implements the make_sound() method.**
- **Defines action(), which creates an instance of Dog inside itself and calls sleep().**

✓ Calling action()

- **Creates a Dog() instance inside action().**
- **Calls the inherited non-abstract method sleep().**

Output:

Animal is sleeping...

10.Packages

1. Create a program to create two class.

1.1. Create a constructor and a method for each class

1.2. Create a __init__.py for adding all packages

1.3. Import the respective packages

1.4. Call each class by creating an object to it

1.5. Create a program by all the above

A:

Python Program with Two Classes, Constructors, Methods, and Package Structure

We will:

1. **Create two classes (Car and Bike)**, each with a **constructor and a method**.
2. **Use __init__.py** to make the directory a package.
3. **Import the classes** from their respective modules.
4. **Call each class by creating an object** and executing their methods.

Project Structure

vehicle_project/

│── vehicles/

│ │── __init__.py

│ │── car.py

```
│   │── bike.py
│── main.py
```

Step 1: Create car.py (First Class)

```
# File: vehicles/car.py
class Car:
    def __init__(self, brand):
        self.brand = brand

    def drive(self):
        print(f"The {self.brand} car is driving.")
```

Step 2: Create bike.py (Second Class)

```
# File: vehicles/bike.py
class Bike:
    def __init__(self, brand):
        self.brand = brand

    def ride(self):
        print(f"The {self.brand} bike is riding.")
```

Step 3: Create __init__.py (Initialize the Package)

```
# File: vehicles/__init__.py

# This makes 'vehicles' a package

from .car import Car

from .bike import Bike
```

✅ __init__.py allows us to import Car and Bike directly from the vehicles package.

Step 4: Create main.py (Import & Call Classes)

```
# File: main.py

from vehicles import Car, Bike  # Importing classes from the package


# Creating objects and calling methods

car = Car("Toyota")

car.drive()  # Output: The Toyota car is driving.


bike = Bike("Yamaha")

bike.ride()  # Output: The Yamaha bike is riding.
```

Step 5: Run the Program

Run main.py:

```
The Toyota car is driving.

The Yamaha bike is riding.
```

11.FILES

1. Write a program to read text file

A:

Python Program to Read a Text File

You can read a text file in Python using the open() function. Here's a simple program that reads and prints the contents of a text file.

Step 1: Create a Text File (sample.txt)

Before running the program, create a text file named sample.txt and add some content:

Hello, this is a sample text file.

Python makes file handling easy!

Step 2: Python Program to Read the File

```
# Open the file in read mode
with open("sample.txt", "r") as file:
    content = file.read()  # Read entire file content
    print("File Content:\n", content)
```

Alternative: Read Line by Line

python

```
# Open the file and read line by line
with open("sample.txt", "r") as file:
    print("File Content:")
    for line in file:
        print(line.strip())  # Strip removes extra newline characters
```

Output

File Content:

Hello, this is a sample text file.

Python makes file handling easy!

2.Write a program to write text to .txt file using InputStream

A:

In Python, we don't have InputStream like in Java, but we can use open() with "w" (write) or "a" (append) mode to write text to a .txt file. Below is a program that takes user input and writes it to a text file.

Python Program to Write to a .txt File Using User Input

```
# Open a file in write mode ("w" overwrites, "a" appends)
with open("output.txt", "w") as file:
    # Take user input
    text = input("Enter text to write into the file: ")
```

```
    # Write input to the file
    file.write(text + "\n")

    print("Text written to output.txt successfully!")
```

Alternative: Writing Multiple Lines with a Mode

If you want to **append** multiple lines using a loop:

```
# Open the file in append mode
with open("output.txt", "a") as file:
    while True:
        text = input("Enter text (or type 'exit' to stop): ")
        if text.lower() == "exit":
            break
        file.write(text + "\n")

    print("Text appended to output.txt successfully!")
```

Explanation:

✓ **open("output.txt", "w")** → Opens the file in write mode (overwrites previous content).

✓ **open("output.txt", "a")** → Opens the file in append mode (adds new

content without deleting old content).

✓ **.write(text + "\n")** → Writes user input into the file and moves to the next line.

✓ **Using with open(...)** → Ensures the file closes automatically.

Would you like to extend this program to handle errors (e.g., file permission issues)?

3.Write a program to read a file stream

A:

Python Program to Read a File Stream

In Python, we use the open() function to read a file stream. Below are different ways to read a file:

Method 1: Read Entire File Using read()

```
# Open the file in read mode
with open("sample.txt", "r") as file:
    content = file.read()  # Read full file content
    print("File Content:\n", content)
```

Method 2: Read File Line by Line Using readline()

```
# Open the file and read one line at a time
with open("sample.txt", "r") as file:
    print("Reading file line by line:")
    line = file.readline()
```

```
    while line:
        print(line.strip())  # Strip removes extra newline characters
        line = file.readline()
```

Method 3: Read File Using readlines() (List of Lines)

```
# Read all lines into a list
with open("sample.txt", "r") as file:
    lines = file.readlines()  # Returns a list of lines
    print("File Content as List:")
    for line in lines:
        print(line.strip())
```

Method 4: Read File as a Stream Using iter()

```
# Open file and read using an iterator
with open("sample.txt", "r") as file:
    print("Reading using an iterator:")
    for line in iter(file.readline, ''):  # Reads until EOF
        print(line.strip())
```

Output (Example if sample.txt Contains This Text)

```
Hello, this is a sample file.
Python makes file handling easy!
```

Output for All Methods:

File Content:

Hello, this is a sample file.

Python makes file handling easy!

4.Write a program to read a file stream supports random access

A:

Python Program for Random Access File Reading

Random access means you can **seek** to any position in the file and read data from there. Python provides this functionality using the .seek() and .tell() methods.

Example: Reading a File Stream with Random Access

```
# Open file in read mode
with open("sample.txt", "r") as file:
    # Read first 10 characters
    print("Reading first 10 characters:")
    print(file.read(10))

    # Get the current position
    position = file.tell()
    print("\nCurrent position in file:", position)
```

```
# Move to a specific position (e.g., 5th character)
file.seek(5)
print("\nReading from position 5:")
print(file.read(10))  # Read next 10 characters

# Move to the end and read last 10 characters
file.seek(-10, 2)  # Move 10 bytes before the end (2 means from end)
print("\nReading last 10 characters:")
print(file.read())
```

Explanation of Key Methods

✓ **file.read(n)** → Reads n characters from the current position.

✓ **file.tell()** → Returns the current file pointer position.

✓ **file.seek(offset, whence)**

- offset: Number of bytes to move.
- whence=0 → Move from the **beginning** (default).
- whence=1 → Move from the **current position**.
- whence=2 → Move from the **end** (use negative values).

Example: If sample.txt Contains This Text

Hello, this is a test file for random access reading.

Expected Output

Reading first 10 characters:

Hello, thi

Current position in file: 10

Reading from position 5:

, this is

Reading last 10 characters:

reading.

5.Write a program to read a file a just to a particular index using seek()

A:

Python Program to Read a File from a Specific Index Using seek()

We will use the .seek() method to move the file pointer to a specific position (index) before reading the content.

Program: Read File from a Specific Index

```
# Open the file in read mode
with open("sample.txt", "r") as file:
    # Move to index 10
```

```
    file.seek(10)

    # Read the next 20 characters from index 10
    content = file.read(20)

    print("Reading from index 10:")
    print(content)
```

Example: If sample.txt Contains This Text

Hello, this is a test file for seek() function.

Expected Output

Reading from index 10:

is is a test file fo

How seek() Works?

✓ **file.seek(10)** → Moves the file pointer to index 10.

✓ **file.read(20)** → Reads 20 characters starting from index 10.

Alternative: Read from the End of the File

```
with open("sample.txt", "r") as file:
    file.seek(-10, 2)  # Move 10 bytes before the end
```

```
    content = file.read()


    print("Reading last 10 characters:")
    print(content)
```

6. Write a program to check whether a file is having read access and write access permissions

A:

Python Program to Check File Read and Write Permissions

Python provides the os.access() method to check file permissions. We can use the following flags:

✓ os.R_OK → Check **read** permission

✓ os.W_OK → Check **write** permission

Program: Check File Read and Write Access

```
import os


# File name to check
file_path = "sample.txt"


# Check if file has read and write permissions
if os.path.exists(file_path):
    has_read_access = os.access(file_path, os.R_OK)
```

```
    has_write_access = os.access(file_path, os.W_OK)


    print(f"Read Access: {'Yes' if has_read_access else 'No'}")
    print(f"Write Access: {'Yes' if has_write_access else 'No'}")
else:
    print("File does not exist.")
```

Example Output

Read Access: Yes

Write Access: No

12.Constructors

1. Write a class with a default constructor, one argument constructor and two argument constructors. Instantiate the class to call all the constructors of that class from a main class.

A:

Python Program with Multiple Constructors Using __init__

Python does not support method overloading directly, but we can achieve it using **default values** or *args (variable-length arguments).

Solution: Using Default Values

```
class MyClass:
    # Constructor Overloading using Default Arguments
```

```
    def __init__(self, arg1=None, arg2=None):
        if arg1 is None and arg2 is None:
            print("Default Constructor Called")
        elif arg2 is None:
            print(f"One-Argument Constructor Called: arg1 = {arg1}")
        else:
            print(f"Two-Argument Constructor Called: arg1 = {arg1}, arg2 = {arg2}")

# Main class to instantiate and call constructors
if __name__ == "__main__":
    obj1 = MyClass()           # Calls default constructor
    obj2 = MyClass(10)         # Calls one-argument constructor
    obj3 = MyClass(10, "Hello") # Calls two-argument constructor
```

Output

```
Default Constructor Called
One-Argument Constructor Called: arg1 = 10
Two-Argument Constructor Called: arg1 = 10, arg2 = Hello
```

Alternative Approach: Using @classmethod to Simulate Constructor Overloading

2. Call the constructors(both default and argument constructors) of super class from a child class.

A:

Python Program: Calling Superclass Constructors from a Child Class

In Python, we use the super() function to call the constructor of a **superclass (parent class)** from a **subclass (child class)**.

Solution: Calling Both Default and Argument Constructors

```
class Parent:
    def __init__(self, arg1=None):
        if arg1 is None:
            print("Parent Default Constructor Called")
        else:
            print(f"Parent Argument Constructor Called: arg1 = {arg1}")


class Child(Parent):
    def __init__(self, arg1=None, arg2=None):
        if arg1 is None and arg2 is None:
            print("Child Default Constructor Called")
            super().__init__()  # Call Parent's Default Constructor
        else:
```

```
        print(f"Child Argument Constructor Called: arg1 = {arg1}, arg2 = {arg2}")
        super().__init__(arg1)  # Call Parent's Argument Constructor

# Main Program
if __name__ == "__main__":
    print("Creating Child Object with Default Constructor:")
    obj1 = Child()  # Calls both Child and Parent default constructors

    print("\nCreating Child Object with Argument Constructor:")
    obj2 = Child(10, "Hello")  # Calls both Child and Parent argument constructors
```

Output

```
Creating Child Object with Default Constructor:
Child Default Constructor Called
Parent Default Constructor Called

Creating Child Object with Argument Constructor:
Child Argument Constructor Called: arg1 = 10, arg2 = Hello
Parent Argument Constructor Called: arg1 = 10
```

Explanation:

✓ **super().__init__()** → Calls **parent class's default constructor**.

✓ **super().__init__(arg1)** → Calls **parent class's argument constructor**.

✓ **Ensures proper constructor chaining in inheritance.**

3. Apply private, public, protected and default access modifiers to the constructor.

A:

Python Program: Applying Access Modifiers to Constructors

Python does not have explicit **private, protected, and public** access modifiers like Java or C++. However, it follows naming conventions:

- **Public (__init__)** → Accessible everywhere.
- **Protected (_init)** → Should not be accessed outside the class (but still possible).
- **Private (__init)** → Name mangled to prevent direct access outside the class.

Solution: Using Naming Conventions for Access Modifiers

```
class Example:
    # Public Constructor
    def __init__(self):
        print("Public Constructor Called")
```

```
    # Protected Constructor
    def _protected_init(self):
        print("Protected Constructor Called")

    # Private Constructor
    def __private_init(self):
        print("Private Constructor Called")

    # Method to call private constructor inside the class
    def access_private_constructor(self):
        self.__private_init()

# Main Program
if __name__ == "__main__":
    obj = Example()  # Calls public constructor

    obj._protected_init()  # Accessible but should be treated as protected

    # obj.__private_init()  # This will cause an AttributeError (private method)

    obj.access_private_constructor()  # Accessing private constructor using a method
```

Output

Public Constructor Called

Protected Constructor Called

Private Constructor Called

4. Write a program which illustrates the concept of attributes of a constructor.

A:

Python Program: Demonstrating Attributes of a Constructor

A **constructor (__init__)** initializes instance attributes when an object is created. These attributes store object-specific data.

Example: Constructor Attributes

```
class Student:
    # Constructor with attributes
    def __init__(self, name, age, course):
        self.name = name          # Instance attribute
        self.age = age            # Instance attribute
        self.course = course      # Instance attribute
        print("Constructor Called: Object Created")

    # Method to display student details
```

```
    def display_details(self):
        print(f"Name: {self.name}, Age: {self.age}, Course: {self.course}")

# Creating objects with constructor attributes
student1 = Student("Alice", 22, "Python")
student2 = Student("Bob", 24, "Java")

# Accessing attributes
print("\nStudent 1 Details:")
student1.display_details()
print("\nStudent 2 Details:")
student2.display_details()
```

Output

```
Constructor Called: Object Created
Constructor Called: Object Created

Student 1 Details:
Name: Alice, Age: 22, Course: Python

Student 2 Details:
Name: Bob, Age: 24, Course: Java
```

13.Method Overloading

1. Write two methods with the same name but different number of parameters of same type and call the methods.

A:

In Python, method overloading (having multiple methods with the same name but different parameters) isn't directly supported like in Java or C++. However, you can achieve similar behavior using **default arguments** or *args.

Example using Method Overloading via Default Arguments:

```
class Example:
    def display(self, a, b=None):
        if b is not None:
            print(f"Two parameters: {a}, {b}")
        else:
            print(f"One parameter: {a}")


# Creating an instance
obj = Example()


# Calling the overloaded methods
obj.display(10)       # Calls the method with one parameter
obj.display(10, 20)   # Calls the method with two parameters
```

Example using *args:

```
class Example:
    def display(self, *args):
        print(f"Called with {len(args)} arguments: {args}")

# Creating an instance
obj = Example()

# Calling the overloaded methods
obj.display(10)       # One argument
obj.display(10, 20)   # Two arguments
obj.display(10, 20, 30)  # Three arguments
```

2. Write two methods with the same name but different number of parameters of different data type and call the methods.

A:

In Python, method overloading (having multiple methods with the same name but different parameters) is not directly supported like in Java or C++. However, you can achieve a similar effect using **default arguments** or *args and **kwargs.

Here's an example of defining two methods with the same name but handling different parameter types:

Method Overloading using Default Arguments

```
class Demo:
```

```
    def display(self, a=None, b=None):
        if a is not None and b is not None:
            print(f"Two parameters: {a}, {b}")
        elif a is not None:
            print(f"One parameter: {a}")
        else:
            print("No parameters")

obj = Demo()
obj.display()          # No parameters
obj.display(10)        # One parameter
obj.display(10, "Hello")  # Two parameters
```

Method Overloading using *args

```
class Demo:
    def display(self, *args):
        print("Parameters:", args)

obj = Demo()
obj.display(5)         # One parameter
obj.display(10, 20)    # Two parameters
obj.display("Hello", 3.14, True)  # Multiple parameters
```

3. Write two methods with the same name and same number of parameters of same type.

A:

In Python, you **cannot** define multiple methods with the same name in a class because the last defined method will override the previous ones. However, you can achieve method overloading behavior using **function overloading tricks** like:

1. **Using @singledispatch from functools** (for different parameter types)
2. **Using Conditional Logic inside the method**

Method Overriding Example (Incorrect Way in Python)

If you try this:

```
class Demo:
    def show(self, a):
        print(f"Method 1: {a}")

    def show(self, a):  # This will override the previous method
        print(f"Method 2: {a}")

obj = Demo()
obj.show(10)  # Output: "Method 2: 10" (First method is overridden)
```

💡 **Python does not support method overloading like Java. The second method replaces the first one.**

Correct Approach: Using a Single Method with Conditional Logic

```
class Demo:
    def show(self, a):
        if isinstance(a, int):
            print(f"Integer method: {a}")
        elif isinstance(a, str):
            print(f"String method: {a}")
        else:
            print(f"Other type: {a}")

obj = Demo()
obj.show(10)      # Calls integer method
obj.show("Hello") # Calls string method
```

Alternative Approach: Using functools.singledispatch (Overloading by Type)

If you specifically want different behavior based on data type, use @singledispatch:

```
from functools import singledispatch

@singledispatch
```

```
def show(a):
    print(f"General method: {a}")

@show.register
def _(a: int):
    print(f"Integer method: {a}")

@show.register
def _(a: str):
    print(f"String method: {a}")

show(10)       # Calls integer version
show("Hello")  # Calls string version
show(3.14)     # Calls general method
```

14.Exceptions

1. Write a program to generate Arithmetic Exception without exception handling.

A:

You can generate an **ArithmeticException** in Python by performing an invalid arithmetic operation, such as **dividing a number by zero**. Here's a simple program that causes an error without handling it:

```
# This will cause a ZeroDivisionError (Arithmetic Exception in Python)
result = 10 / 0


print("This line will not be executed due to the exception.")
```

Expected Output (Error Message)

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    result = 10 / 0
ZeroDivisionError: division by zero
```

2. Handle the Arithmetic exception using try-catch block

A:

In Python, exceptions are handled using the **try-except** block (similar to try-catch in Java). Here's how you can handle an **Arithmetic Exception (ZeroDivisionError):**

Example: Handling Arithmetic Exception

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except ZeroDivisionError as e:
    print("Arithmetic Exception occurred:", e)

print("Program continues execution after handling the exception.")
```

Expected Output

Arithmetic Exception occurred: division by zero

Program continues execution after handling the exception.

3. Write a method which throws exception, Call that method in main class without try block.

A:

In Python, you can create a method that raises an exception using the raise keyword. If you call this method **without handling it** using a try-except block, the program will terminate with an error.

Example: Method That Throws an Exception

```
# Defining a method that raises an exception
def throw_exception():
    raise ValueError("This is a custom exception!")

# Calling the method without handling the exception
```

```
throw_exception()


print("This line will never execute due to the unhandled exception.")
```

Expected Output (Program Crashes)

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    throw_exception()
  File "test.py", line 3, in throw_exception
    raise ValueError("This is a custom exception!")
ValueError: This is a custom exception!
```

4. Write a program with multiple catch blocks

A:

In Python, we use **multiple except blocks** to handle different types of exceptions separately. Here's a program demonstrating multiple exception handlers:

Example: Multiple except Blocks

```
def multiple_exceptions(a, b):
    try:
        result = a / b  # May cause ZeroDivisionError
        numbers = [1, 2, 3]
```

```
        print(numbers[5])  # May cause IndexError
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    except IndexError:
        print("Error: List index out of range.")
    except Exception as e:  # Catches any other unexpected exception
        print(f"Unexpected error occurred: {e}")

# Calling the function with different cases
multiple_exceptions(10, 0)  # ZeroDivisionError
multiple_exceptions(10, 2)  # IndexError
```

Expected Output

Error: Division by zero is not allowed.

Error: List index out of range.

5. Write a program to throw exception with your own message

A:

In Python, you can **throw an exception with a custom message** using the raise keyword. Here's an example:

Example: Throwing a Custom Exception

```
def check_age(age):
```

```
    if age < 18:
        raise ValueError("Access Denied: You must be at least 18 years old.")
    else:
        print("Access Granted!")

# Calling the function with an invalid age
check_age(16)  # This will raise an exception
```

Expected Output (Exception Raised)

```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    check_age(16)
  File "test.py", line 3, in check_age
    raise ValueError("Access Denied: You must be at least 18 years old.")
ValueError: Access Denied: You must be at least 18 years old.
```

Explanation

1. The function check_age() checks if the age is less than 18.
2. If **age < 18**, it **throws a ValueError with a custom message**.
3. Since we did not use a try-except block, the program **crashes when the exception is raised**.

Handling the Exception (Optional)

If you want to handle the exception instead of crashing the program:

```
try:
    check_age(16)
except ValueError as e:
    print("Caught Exception:", e)
```

Output:

Caught Exception: Access Denied: You must be at least 18 years old.

6. Write a program to create your own exception

A:

In Python, you can **create your own exception** by defining a custom exception class that inherits from the built-in Exception class.

Example: Creating a Custom Exception

```
# Define a custom exception
class AgeTooSmallError(Exception):
    def __init__(self, message="Age must be at least 18!"):
        self.message = message
        super().__init__(self.message)

# Function to check age
def check_age(age):
```

```
    if age < 18:
        raise AgeTooSmallError(f"Access Denied: {age} is too young.")
    else:
        print("Access Granted!")

# Using try-except to handle the custom exception
try:
    check_age(16)  # This will raise the custom exception
except AgeTooSmallError as e:
    print("Caught Exception:", e)
```

Expected Output

Caught Exception: Access Denied: 16 is too young.

Explanation

✓ **Custom Exception Class (AgeTooSmallError)** inherits from Exception.

✓ We override the __init__ method to set a custom error message.

✓ The check_age() function raises AgeTooSmallError if the age is less than 18.

✓ The try-except block catches the custom exception and prints the message.

7. Write a program with finally block

A:

In Python, the **finally block** is used to execute code **regardless of whether an exception occurs or not**. It is often used for cleanup tasks like closing files or releasing resources.

Example: Using finally Block

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    finally:
        print("Execution of try-except block completed.")

# Calling the function with different cases
divide_numbers(10, 2)  # Normal case
divide_numbers(10, 0)  # Exception case
```

Expected Output

```
Result: 5.0
```

Execution of try-except block completed.

Error: Cannot divide by zero!

Execution of try-except block completed.

Explanation

✓ **try block:** Attempts division.

✓ **except block:** Catches ZeroDivisionError.

✓ **finally block:** Always executes, whether an exception occurs or not.

8. Write a program to generate Arithmetic Exception

A:

You can generate an **Arithmetic Exception** in Python by performing an invalid arithmetic operation, such as **division by zero**.

Example: Generating an Arithmetic Exception

```
# This will cause a ZeroDivisionError (Arithmetic Exception)
result = 10 / 0


print("This line will not be executed due to the exception.")
```

Expected Output (Error Message)

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
```

```
result = 10 / 0
```

```
ZeroDivisionError: division by zero
```

Explanation

- In Python, **dividing by zero** raises a ZeroDivisionError, which is similar to an **ArithmeticException** in Java.
- Since we are **not handling the exception**, the program **crashes** immediately.

Handling the Exception (Optional)

If you want to **handle the exception** to prevent a crash:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("Arithmetic Exception occurred:", e)
```

Output:

```
Arithmetic Exception occurred: division by zero
```

9. Write a program to generate FileNotFoundException.

A:

In Python, the **FileNotFoundError** occurs when you try to open a file that does not exist. Here's how you can generate it:

Example: Generating FileNotFoundError

```
# Attempting to open a non-existent file
file = open("non_existent_file.txt", "r")  # File does not exist

print("This line will not execute due to the exception.")
```

Expected Output (Error Message)

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    file = open("non_existent_file.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory:
'non_existent_file.txt'
```

Explanation

- Python raises a **FileNotFoundError** when attempting to open a file that **does not exist**.
- The program **crashes** because there's **no exception handling**.

Handling the Exception (Optional)

If you want to **handle the exception** gracefully:

```
try:
    file = open("non_existent_file.txt", "r")
```

except FileNotFoundError as e:

    print("Error: File not found!", e)

Output:

Error: File not found! [Errno 2] No such file or directory: 'non_existent_file.txt'

10. Write a program to generate ClassNotFoundException

A:

In Python, there is **no direct equivalent** of Java's ClassNotFoundException because Python is **dynamically typed** and does not require explicit class loading. However, you can simulate a similar error using **ModuleNotFoundError** or **AttributeError**.

Example 1: Simulating ClassNotFoundException Using ModuleNotFoundError

Trying to import a non-existent module

import nonexistentmodule # This module does not exist

Expected Output:

Traceback (most recent call last):

  File "test.py", line 2, in <module>

    import nonexistentmodule

ModuleNotFoundError: No module named 'nonexistentmodule'

📌 **Explanation:**

- This is similar to ClassNotFoundException in Java when a module (package) is missing.

Example 2: Simulating ClassNotFoundException Using AttributeError

```
import math


# Trying to access a non-existent class in the 'math' module
obj = math.NonExistentClass()
```

Expected Output:

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    obj = math.NonExistentClass()
AttributeError: module 'math' has no attribute 'NonExistentClass'
```

📌 **Explanation:**

- This is similar to ClassNotFoundException when trying to load a class that does not exist in a module.

Example 3: Handling the Exception Properly

If you want to **handle the error gracefully**, use a try-except block:

```
try:
    import nonexistentmodule  # This will raise ModuleNotFoundError
except ModuleNotFoundError as e:
```

```
    print("Error: The requested module was not found!", e)


try:
    obj = math.NonExistentClass()  # This will raise AttributeError
except AttributeError as e:
    print("Error: The requested class was not found in the module!", e)
```

Output:

Error: The requested module was not found! No module named 'nonexistentmodule'

Error: The requested class was not found in the module! module 'math' has no attribute 'NonExistentClass'

10.Write a program to generate IOException

A:

In Python, **IOException** does not exist as a separate exception like in Java. Instead, Python handles **input-output (I/O) errors** using the built-in OSError and its subclass IOError (which is now merged into OSError in Python 3).

Example: Generating an I/O Exception (FileNotFoundError)

```
# Trying to open a non-existent file (causes an I/O error)
file = open("non_existent_file.txt", "r")
```

Expected Output:

Traceback (most recent call last):

File "test.py", line 2, in <module>

file = open("non_existent_file.txt", "r")

FileNotFoundError: [Errno 2] No such file or directory: 'non_existent_file.txt'

📌 **Explanation:**

- This is a common I/O error when trying to read a **file that does not exist**.

Example 2: Handling I/O Exception (OSError)

```
try:
    file = open("non_existent_file.txt", "r")  # File does not exist
except FileNotFoundError as e:
    print("I/O Exception occurred:", e)
```

Output:

I/O Exception occurred: [Errno 2] No such file or directory: 'non_existent_file.txt'

Example 3: Generating an I/O Exception When Writing to a Read-Only File

```
try:
    with open("/root/protected_file.txt", "w") as file:  # Writing to a protected location
        file.write("This will fail.")
except PermissionError as e:
```

```
    print("I/O Exception: Permission denied!", e)
```

Expected Output:

I/O Exception: Permission denied! [Errno 13] Permission denied: '/root/protected_file.txt'

📌 **Explanation:**

- This occurs when **trying to write to a protected location** without proper permissions.

12. Write a program to generate NoSuchFieldException

A:

In Python, there is **no direct equivalent** of Java's NoSuchFieldException because Python is **dynamically typed** and does not enforce strict field definitions in classes.

However, trying to access a **non-existent attribute (field) in a class** raises an **AttributeError**, which is similar to Java's NoSuchFieldException.

Example: Generating AttributeError (Equivalent to NoSuchFieldException)

```
class Person:
    def __init__(self, name):
        self.name = name  # Only 'name' attribute exists

# Creating an object of Person
person = Person("John")
```

```
# Trying to access a non-existent attribute (field)
print(person.age)  # 'age' field does not exist
```

Expected Output:

```
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    print(person.age)
AttributeError: 'Person' object has no attribute 'age'
```

Example: Handling the Exception Gracefully

```
class Person:
    def __init__(self, name):
        self.name = name  # Only 'name' field exists

person = Person("John")

try:
    print(person.age)  # 'age' field does not exist
except AttributeError as e:
    print("Error: The requested field does not exist!", e)
```

Output:

```
Error: The requested field does not exist! 'Person' object has no attribute 'age'
```

15.Dictionary

Create a Dictionary with at least 5 key value pairs of the Student ID and Name 1.1. Adding the values in dictionary 1.2. Updating the values in dictionary 1.3. Accessing the value in dictionary 1.4. Create a nested loop dictionary 1.5. Access the values of nested loop dictionary 1.6. Print the keys present in a particular dictionary 1.7. Delete a value from a dictionary

A:

Here's a complete Python program demonstrating all the required dictionary operations:

1. Create a Dictionary with 5 Key-Value Pairs (Student ID & Name)

```
# Creating a dictionary with Student ID as key and Name as value
students = {
    101: "Alice",
    102: "Bob",
    103: "Charlie",
    104: "David",
    105: "Emma"
}
print("Initial Dictionary:", students)
```

1.1 Adding Values to Dictionary

```
# Adding a new student entry
students[106] = "Frank"
print("After Adding a New Student:", students)
```

1.2 Updating Values in Dictionary

```
# Updating the name of Student ID 103
students[103] = "Charlie Brown"
print("After Updating Student 103:", students)
```

1.3 Accessing a Value in Dictionary

```
# Accessing a specific student name using Student ID
student_id = 102
print(f"Student with ID {student_id}:", students[student_id])
```

1.4 Create a Nested Dictionary

```
# Creating a nested dictionary with student details
student_details = {
    101: {"Name": "Alice", "Age": 20, "Grade": "A"},
    102: {"Name": "Bob", "Age": 21, "Grade": "B"},
    103: {"Name": "Charlie", "Age": 19, "Grade": "A"},
}
print("Nested Dictionary:", student_details)
```

1.5 Access Values from Nested Dictionary

```
# Accessing details of Student 102
print("Details of Student 102:", student_details[102])

# Accessing only the Grade of Student 103
print("Grade of Student 103:", student_details[103]["Grade"])
```

1.6 Print All Keys in Dictionary

```
# Printing all keys in the dictionary
print("Keys in students dictionary:", students.keys())
```

1.7 Delete a Value from Dictionary

```
# Deleting a student from dictionary
del students[104]
print("After Deleting Student 104:", students)
```

Full Output Example

Initial Dictionary: {101: 'Alice', 102: 'Bob', 103: 'Charlie', 104: 'David', 105: 'Emma'}

After Adding a New Student: {101: 'Alice', 102: 'Bob', 103: 'Charlie', 104: 'David', 105: 'Emma', 106: 'Frank'}

After Updating Student 103: {101: 'Alice', 102: 'Bob', 103: 'Charlie Brown', 104: 'David', 105: 'Emma', 106: 'Frank'}

Student with ID 102: Bob

Nested Dictionary: {101: {'Name': 'Alice', 'Age': 20, 'Grade': 'A'}, 102: {'Name': 'Bob', 'Age': 21, 'Grade': 'B'}, 103: {'Name': 'Charlie', 'Age': 19, 'Grade': 'A'}}

Details of Student 102: {'Name': 'Bob', 'Age': 21, 'Grade': 'B'}

Grade of Student 103: A

Keys in students dictionary: dict_keys([101, 102, 103, 104, 105, 106])

After Deleting Student 104: {101: 'Alice', 102: 'Bob', 103: 'Charlie Brown', 105: 'Emma', 106: 'Frank'}