S-VYASA
DEEMED - TO - BE - UNIVERSITY
SCHOOL FOR ADVANCED STUDIES

# PSUP - 2

Course code: **BTPS 204**

**Syllabus**

**Module-1 - 21 hours**
Control Structures Advanced:
Nested if, Nested if, else if, Else, for loop, while loop in arrays and strings; Exception handling; using recursive functions to solve problems that can be broken down into smaller sub-problems

**Module – 2 - 21 hours**
Loops Advanced:
advanced techniques for using loops, such as using loop counters, loop flags, and sentinel values; using advanced branching techniques such as the ternary operator and conditional expressions

**Module – 3 - 21 hours**
Modularity:
Organising code into modules, classes, and functions to improve code structure and reusability; Function parameters; Function return value; reviewing recursion and its use in function design and implementation; Libraries and APIs.

**Module-4 - 21 hours**
Introduction to Data Structures:
Overview of data structures, Arrays and linked lists, Stacks and queues, Trees and graphs - basic concepts along with practice problems.

**Module-5 - 21 hours**

Using Generative AI:

Breaking down the problem to programming patterns, Prompt engineering for code, Prompt engineering for C and Python, use of AI agents in writing functionality.

Textbooks:

● Think Like a Programmer: An Introduction to Creative Problem Solving by V. Anton Spraul, Released August 2012, published by No Starch Press

Reference Books:

● Programming in Python 3: A Complete Introduction to the Python Language; Mark Summerfield; Pearson Education; Second edition

# MODULE - 1

## Control Structures Advanced

**1.Q:What is a nested if statement in Python?**

**A:**

A nested if is an if statement placed inside another if statement.It allows multi-level decision-making.The inner if executes only if the outer if condition is True.

Key Points:

- Used when decisions depend on multiple conditions.
- Helps in hierarchical decision-making.
- Inner if cannot execute unless the outer if condition is met.

Eg:

```
age = 20

has_license = True

if age >= 18:  # Outer if condition

    if has_license:  # Nested if condition

        print("You are allowed to drive.")

    else:

        print("You need a driving license.")

else:

    print("You are underaged.")
```

**2. Q: What is a nested if statement in Python?**

**A:**

An if-else statement provides two execution paths.If the condition is True, the if block runs.If False, the else block runs.

Key Points:

- Helps in decision-making with two possible outcomes.
- Ensures that at least one block of code executes.
- Avoids unnecessary checks when one condition is met.

Eg:

```
temperature = 35

if temperature > 30:

    print("It's hot outside!")

else:

    print("The weather is cool.")
```

**3. What is for loop in python ?**

**A:**

A for loop is used to iterate over sequences like lists, strings, and ranges.

It executes the loop body a fixed number of times.

Key Points:

- Used for iterating over sequences (lists, strings, tuples, ranges).
- Helps in reducing redundancy in code.
- Works efficiently with indexing.

Eg:

```
numbers = [10, 20, 30, 40]

for num in numbers:

    print(num)
```

**4. Q: What is while loop in python?**

**A:**

A while loop executes repeatedly as long as a given condition is True.

It is commonly used when the number of iterations is unknown beforehand.

In arrays (lists), while loops help in iterating over elements dynamically.

Key Points:

- Used when the number of iterations is not predetermined.
- Requires manual index management (i needs to be incremented).
- Can be used for searching, filtering, or modifying elements in a list.

Eg:

```
numbers = [5, 10, 15, 20]

i = 0  # Initialize index


while i < len(numbers):  # Loop until index reaches the list length
    print(numbers[i])
    i += 1  # Move to the next element
```

**5. Q: Write a Python program that calculates an employee's bonus based on their performance rating. The bonus is a percentage of the employee's current salary and is determined as follows:**

- **Rating 4.5 and above → 30% of salary**
- **Rating 4.0 to 4.49 → 20% of salary**
- **Rating 3.5 to 3.99 → 10% of salary**
- **Rating 3.0 to 3.49 → 5% of salary**
- **Rating Below 3.0 → No bonus**

**Take the employee’s salary and rating as input and output the calculated bonus amount.**

**A:**

```
def calculate_bonus(salary, rating):
    if rating >= 4.5:
        bonus = salary * 0.30  # 30% bonus
    elif rating >= 4.0:
        bonus = salary * 0.20  # 20% bonus
    elif rating >= 3.5:
        bonus = salary * 0.10  # 10% bonus
    elif rating >= 3.0:
        bonus = salary * 0.05  # 5% bonus
    else:
        bonus = 0  # No bonus
    return bonus
salary = float(input("Enter your salary: "))
rating = float(input("Enter your performance rating (0-5): "))
bonus = calculate_bonus(salary, rating)
print("Your bonus amount is:", bonus)
```

**6.Q:Write a Python function to evaluate student grades based on their scores. The grading system is asfollows:**

- **90 - 100: Grade A**
- **80 - 89: Grade B**
- **70 - 79: Grade C**
- **60 - 69: Grade D**
- **Below 60: Grade F**
- **Invalid if the score is negative or above 100**

**The program should take the student's score as input and output the corresponding grade.**

**A:**

```
def calculate_grade(score):
  if 90 <= score <= 100:
     grade = "A"
  elif 80 <= score < 90:
     grade = "B"
  elif 70 <= score < 80:
     grade = "C"
  elif 60 <= score < 70:
     grade = "D"
  elif 0 <= score < 60:
     grade = "F"
  else:
     grade = "Invalid score"
  return grade
score = float(input("Enter your score: "))
grade = calculate_grade(score)
print("Your grade is: “,grade)
```

**7.Q: Design an ATM system where the user enters their current bank balance and a withdrawal amount. The system should follow these rules:**

- **The withdrawal amount must be greater than 0.**
- **The withdrawal amount must be in multiples of ₹100.**
- **The withdrawal amount must not exceed the available balance.**
- **If all conditions are met, deduct the amount and display the remaining balance.**
- **If any condition fails, display an appropriate error message.**

A:

```
def atm_withdrawal(balance, amount):
  if amount <= 0:
     return "Invalid amount entered."
  elif amount > balance:
     return "Insufficient balance."
  elif amount % 100 != 0:
     return "Amount must be in multiples of 100."
  else:
     balance -= amount
     return "Withdrawal successful. Remaining balance: “,balance
balance = float(input("Enter your account balance: "))
amount = float(input("Enter withdrawal amount: "))
print(atm_withdrawal(balance, amount))
```

**8. Q: A program that checks whether a person qualifies for a loan based on their credit score, income, and current debts.**

**Write a Python program to determine if a person is eligible for a loan. The loan approval depends on the following conditions:**

- **If the person's credit score is 700 or higher, they may qualify for a loan.**
- **If the income is greater than 50,000, they are more likely to qualify.**
- **If the current debts are less than 20,000, the loan can be approved.**
- **The program should return an appropriate message based on these conditions.**

**A:**

```
def loan_approval(credit_score, income, current_debts):
  if credit_score >= 700:
     if income > 50000:
```

```
        if current_debts < 20000:
            return "Loan Approved"
        else:
            return "Loan Denied due to high debts"
    else:
        return "Loan Denied due to low income"
  else:
    return "Loan Denied due to low credit score"
# Test the function
print(loan_approval(750, 60000, 15000))
print(loan_approval(650, 45000, 25000))
```

**9.Q: Write a Python program to determine the price of a movie ticket based on the following conditions:**

- **If the customer is under 18, the ticket price is $8, but if they have a membership, they get a $5 ticket.**
- **If the customer is between 18 and 60 years old, the ticket price is $15, but if they have a membership, they get a $10 ticket.**
- **If the customer is over 60, the ticket price is $10, but if they have a membership, they get a $7 ticket.**

**A:**

```
def ticket_price(age, membership):
  if age < 18:
    if membership == "yes":
      return "Ticket Price: $5 (Member Discount)"
    else:
      return "Ticket Price: $8"
  elif 18 <= age <= 60:
```

```
        if membership == "yes":
            return "Ticket Price: $10 (Member Discount)"
        else:
            return "Ticket Price: $15"
    else:
        if membership == "yes":
            return "Ticket Price: $7 (Member Discount)"
        else:
            return "Ticket Price: $10"
# Test the function
print(ticket_price(17, "yes"))
print(ticket_price(45, "no"))
```

**10.Q:Write a Python program that calculates an employee's overtime pay based on the number of extra hours worked.**

- **For each hour worked beyond 40 hours, the employee earns 1.5 times their hourly wage.**
- **If an employee worked 45 hours and their hourly wage is $20, the overtime pay should be (5 * 20 * 1.5) = $150.**

**Take the total hours worked and hourly wage as input and output the calculated overtime pay.**

A:

```
def calculate_overtime(hours_worked, hourly_wage):
    overtime_pay = 0
    if hours_worked > 40:
        extra_hours = hours_worked - 40
        for _ in range(extra_hours):
            overtime_pay += hourly_wage * 1.5
```

```
    return overtime_pay

hours = int(input("Enter total hours worked: "))

wage = float(input("Enter hourly wage: "))

print("Your overtime pay is: $", calculate_overtime(hours, wage))
```

**11.Q: Write a Python program that calculates the total balance in a savings account over multiple years, given an initial deposit and a fixed interest rate.**

- **Interest is compounded annually at a fixed rate (e.g., 5% per year).**
- **The program should use a for loop to calculate the new balance for each year.**
- **Take the initial deposit, interest rate, and number of years as input and output the balance for each year.**

A:

```
def calculate_balance(deposit, rate, years):

    for year in range(1, years + 1):

        deposit += deposit * (rate / 100)

        print(f"Year {year}: Balance = ${deposit:.2f}")


initial = float(input("Enter initial deposit: "))

rate = float(input("Enter annual interest rate (%): "))

years = int(input("Enter number of years: "))

calculate_balance(initial, rate, years)
```

**12.Q: Write a Python program that calculates the total parking fee based on the number of hours a vehicle is parked.**

- **The first 2 hours are free.**
- **After 2 hours, each additional hour costs $5.**
- **The program should keep asking for the number of hours parked until a valid number is entered (> 0).**
- **Take the hours parked as input and output the total parking fee.**

**A:**

```
while True:
    hours = int(input("Enter number of hours parked: "))
    if hours > 0:
        break
    print("Invalid input. Please enter a valid number of hours.")


if hours <= 2:
    fee = 0
else:
    fee = (hours - 2) * 5
print(f"Total parking fee: ${fee}”)
```

**13.Q: Write a Python program that asks the user to enter a password.**

- **The password must be at least 8 characters long.**
- **The program should keep asking for a valid password until the condition is met.**
- **Once a valid password is entered, print "Password set successfully!".**

**A:**

```
while True:
    password = input("Enter a password (at least 8 characters long): ")
    if len(password) >= 8:
        print("Password set successfully!")
        break
    else:
        print("Password too short. Try again.”)
```

**14.Q.Write a Python program to print the Fibonacci series using a while loop.**

- **The Fibonacci series starts with 0, 1 and each next number is the sum of the previous two.**
- **The user inputs how many terms to generate.**

**Example:**

**Input: 7**

**Output: 0 1 1 2 3 5 8**

A:

```
n = int(input("Enter the number of terms: "))

a, b = 0, 1

count = 0

while count < n:

    print(a, end=" ")

    a, b = b, a + b

    count += 1
```

**15.Q: Write a Python program that tracks the attendance of employees for an entire week (7 days).**

- **The program should ask the user for attendance (Present / Absent) for each day.**
- **After the week ends, it should count how many days the employee was present and display the attendance percentage**

**A:**

```
total_days = 7

present_days = 0

for day in range(1, total_days + 1):

    status = input(f"Day {day}: Were you present? (yes/no): ").lower()

    if status == "yes":

        present_days += 1
```

```
attendance_percentage = (present_days / total_days) * 100

print(f"Total Present Days: {present_days}")

print(f"Attendance Percentage: {attendance_percentage:.2f}%")
```

**16. Q: Write a Python program that calculates an employee's total commission based on weekly sales for 4 weeks.**

- **If weekly sales are above $5000, the employee earns a 10% commission for that week.**
- **If weekly sales are between $3000 and $4999, the commission is 5%.**
- **If weekly sales are below $3000, there is no commission.**

**The program should calculate the total commission earned over 4 weeks**

**A:**

```
total_commission = 0

for week in range(1, 5):
    sales = float(input(f"Enter sales for Week {week}: "))
    if sales > 5000:
        commission = sales * 0.10
    elif sales >= 3000:
        commission = sales * 0.05
    else:
        commission = 0
    total_commission += commission
print(f"Total commission earned in 4 weeks: ${total_commission:.2f}")
```

**17.Q:What is exception handling and why do we need exception handling ?**

A:

- Exception handling is a mechanism used to handle runtime errors in a program.
- It prevents the program from crashing and allows smooth execution.
- Errors are caught using try and except blocks.

Why Use Exception Handling?

- Prevents program crashes.
- Helps in debugging by identifying errors.
- Improves user experience by handling errors gracefully.
- Allows execution of alternate code when an error occurs.

**18.Q:What is the purpose of the try, except, else, and finally blocks in Python?**

**A:**

try: This block is used to write code that you want to test for errors. If an error occurs, it will be caught in the except block.
except: This block catches and handles the error if one occurs in the try block.
else: This block is optional. It runs only if no exceptions are raised in the try block. It must come after all except blocks.
finally: This block always runs, no matter what — whether an exception occurs or not. It is typically used for cleanup actions like closing files or releasing resources.

**19.Q: What happens if an exception is not handled in Python?**

**A:**

If an exception is not handled in Python, the program immediately stops running (crashes) at the point where the exception occurred. Python then automatically generates an error message, known as a traceback.

The traceback provides detailed information about:

- The type of exception that occurred (e.g., ZeroDivisionError, ValueError, IndexError, etc.).
- The exact line number in the code where the exception happened.
- The call stack, showing the sequence of function calls that led to the error.

This information is useful for debugging because it helps the programmer identify and fix the source of the problem. However, from a user's point of view, an unhandled exception causes the program to

terminate unexpectedly, leading to a poor user experience.That is why it is important to handle exceptions properly to make programs more reliable and user-friendly.

**20.Q.John is developing a calculator application where users enter two numbers to perform division. However, he notices that if the denominator is zero, the program crashes with a runtime error.How can John modify his code to gracefully handle this situation and display an appropriate error message to the user, instead of crashing**

**A:**

```
def divide_numbers():
    try:
        numerator = float(input("Enter the numerator: "))
        denominator = float(input("Enter the denominator: "))

        result = numerator / denominator
        print(f"Result: {result}")

    except ZeroDivisionError:
        print("Error: Division by zero is not allowed. Please enter a non-zero denominator.")
    except ValueError:
        print("Error: Please enter valid numeric values.")

# Run the function
divide_numbers()
```

**21.Q:Imagine you are developing a banking application where customers must deposit money into their accounts.You must ensure that the customer only enters a positive integer amount when making a deposit.If the customer enters invalid input (such as letters, symbols, or negative numbers), the system should not crash, but instead show an error message and ask again until valid input is provided.**

**A:**

```
def get_valid_deposit():
    while True:
        try:
            deposit = int(input("Enter the amount to deposit: "))
            if deposit > 0:
                return deposit
            else:
                print("Error: Deposit amount must be a positive number.")
        except ValueError:
            print("Error: Please enter a valid integer amount (e.g., 100, 500).")


# Run the function
deposit_amount = get_valid_deposit()
print(f"Deposit successful! Amount deposited: ${deposit_amount}")
```

**22.Q. What is recursive function and why do we need recursive function ? explain with an example**

A:
A recursive function is a function that calls itself in order to solve a problem.We use recursive functions when a problem can be broken down into smaller sub-problems of the same type.This makes the code simpler and easier to understand, especially for problems involving repetition or nested structures.

## Why do we need recursion?

- It helps solve complex problems like factorial, Fibonacci, tree traversal, tower of Hanoi, etc.
- It provides a clean and elegant solution for problems that are naturally recursive.
- It avoids the need for loops in some cases.

Eg:

```
def factorial(n):
    if n == 0:  # base case
        return 1
    else:
        return n * factorial(n - 1)  # recursive cas


# Test the function
print("Factorial of 5 is:", factorial(5))
```

**23. Write the pros and cons of recursive functions?**

A:

## Advantages of Recursive Functions:

1. Simplicity: Recursion can make complex problems (like factorial, Fibonacci, or tree traversal) easier to understand and solve.
2. Cleaner Code: Reduces the need for loops, leading to more readable and concise code.
3. Mathematical Mapping: Perfect for problems with mathematical or hierarchical structures (e.g., divide and conquer).

Disadvantages of Recursive Functions:

1. Memory Usage: Each recursive call uses memory on the stack, and too many calls can lead to stack overflow.

2. Performance: Recursive solutions can be less efficient due to repeated calculations (e.g., Fibonacci without optimization).

3. Limited Depth: Most languages (like Python) have a recursion depth limit, which can cause errors with deep recursion.

## MODULE - 2

## Loops Advanced

**1.Q: What is a loop counter, and how is it used in loops?**

**A:**

A loop counter is a variable used to keep track of the number of iterations in a loop. It is typically initialized before the loop starts and is incremented or decremented during each iteration. Loop counters are useful for controlling how many times a loop executes and can be used in for loops or while loops.

```
# Example of loop counter

for i in range(5):  # i will count from 0 to 4

   print(f"Iteration {i+1}")
```

**2.Q:What is a loop flag, and how is it used in loops?**

**A:**
A loop flag is a boolean variable used to control the execution of a loop. It can be set to True or False to signal whether the loop should continue or break. Flags are often used when the exit condition is complex or involves multiple checks.

```
# Example of loop flag

found = False

while not found:
```

```
    user_input = input("Enter a positive number: ")
    if user_input.isdigit() and int(user_input) > 0:
        print(f"Valid input: {user_input}")
        found = True  # Set the flag to True to exit the loop
    else:
        print("Invalid input. Try again.")
```

**3. Q:What is a sentinel value, and how does it work in loops?**

**A:**
A sentinel value is a special value used to signal the end of data input or the termination of a loop. It is typically an impossible or unusual value that wouldn't appear in normal data. Sentinel values are commonly used in while loops to keep reading input until the sentinel value is encountered.

```
# Example of sentinel value
total = 0
while True:
    user_input = input("Enter a number to add to the total (or type 'done' to finish): ")
    if user_input == 'done':  # Sentinel value
        break
    total += int(user_input)
print(f"Total sum: {total}")
```

**4. Q:A company has a system that processes customer orders. Each order has a total price, but the system should stop processing when it encounters an order with a negative value (indicating an error in the order). Use a sentinel value to terminate the loop when this happens.**

A:
```
total_amount = 0
while True:
```

```
    order_price = float(input("Enter the price of the order (negative value to stop): "))
    if order_price < 0:  # Sentinel value to stop processing
        print("Error encountered. Stopping order processing.")
        break
    total_amount += order_price
print(f"Total processed amount: {total_amount}")
```

**5.Q:You are writing a program to find the first even number in a list. You need to use a loop flag to exit the loop as soon as the first even number is found.**

**A:**

```
numbers = [11, 15, 21, 8, 19, 25]
found_even = False
for num in numbers:
    if num % 2 == 0:
        print(f"First even number found: {num}")
        found_even = True
        break
if not found_even:
    print("No even number found.")
```

**6.Q:You need to create a program that loops through a series of test scores and counts how many are above 50. Use a loop counter to track the number of scores above 50.**

**A:**

```
scores = [45, 78, 56, 39, 87, 55, 67, 32]
count_above_50 = 0
```

```
for score in scores:
    if score > 50:
        count_above_50 += 1
print(f"Number of scores above 50: {count_above_50}")
```

**7.Q: Write a program to calculate the sum of all even numbers from 1 to 100 using a loop counter.**

**A:**

```
total_sum = 0
for num in range(2, 101, 2):  # Start from 2, go up to 100, increment by 2
    total_sum += num
print(f"Sum of even numbers from 1 to 100: {total_sum}")
```

**8.Q:Write a program that keeps asking the user for numbers and adds them to a running total. The loop should terminate when the user enters "0" using a sentinel value.**

**A:**

```
total = 0
while True:
    user_input = input("Enter a number (or '0' to stop): ")
    if user_input == '0':  # Sentinel value to stop
        break
    total += int(user_input)
print(f"Total sum: {total}")
```

**9.Q: Create a program that finds the largest number in a list. Stop the loop when you encounter a number greater than 100 using a loop flag.**

**A:**

```
numbers = [45, 67, 89, 101, 32, 54]

largest = None

found = False

for num in numbers:

    if num > 100:  # Flag to stop the loop

        found = True

        break

    if largest is None or num > largest:

        largest = num

if found:

    print("A number greater than 100 was found. Stopping.")

else:

    print(f"Largest number in the list is: {largest}")
```

**10.Q:Imagine you are organizing a sports event, and you need to distribute participation medals. You decide to reward only the participants with even-numbered registration IDs, ranging from 1 to N. Your task is to calculate the total sum of all even-numbered registration IDs to determine the total cost of medals. Write a program that takes a positive integer N as input and computes the sum of all even numbers from 1 to N using a for loop.**

**A:**

```
N = int(input("Enter the value of N\n"))

total = 0

# Loop from 1 to N

for i in range(1, N + 1):

    if i % 2 == 0:

        total += i

print("Total sum of even-numbered registration IDs:", total)
```

**11.Q: You are building an online shopping cart system. When a user enters items into their cart, the system should keep a running total of the price. If the user types "done", the system should stop asking for prices. Implement this using a sentinel value**.

**A:**

```
total_price = 0

while True:

    price = input("Enter the price of an item (or type 'done' to finish): ")

    if price.lower() == 'done':

        break

    total_price += float(price)

print(f"Total price of items: ${total_price}")
```

**12.Q: A company's payroll system calculates the total salary of employees. The system should stop adding salary figures if a negative value is entered (indicating an error). Use a sentinel value to terminate the loop and avoid adding incorrect values.**

**A:**

```
total_salary = 0

while True:

    salary = float(input("Enter salary (negative value to stop): "))

    if salary < 0:

        print("Negative salary entered. Stopping payroll processing.")

        break

    total_salary += salary

print(f"Total salary: ${total_salary}")
```

**13.Q: Create a program that accepts a list of integers from the user. The program should count how many of the integers are even numbers. Use a loop counter to track the count of even numbers.**

**A:**

```
even_count = 0

numbers = input("Enter a list of numbers separated by spaces: ").split()

for num in numbers:

    if int(num) % 2 == 0:

        even_count += 1

print(f"Total even numbers: {even_count}")
```

**14.Q:A school is processing student grades. The system should continue accepting grades for students until a grade of `-1` (sentinel value) is entered. The program should then calculate and display the average grade.**

**A:**

```
grades = []

while True:

    grade = float(input("Enter grade (or -1 to stop): "))

    if grade == -1:

        break

    grades.append(grade)

if grades:

    average_grade = sum(grades) / len(grades)

    print(f"The average grade is: {average_grade}")

else:

    print("No grades entered.")
```

**15.Q: Create a program that counts the number of prime numbers in a list of integers. Use a loop counter to keep track of the number of prime numbers found.**

**A:**

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

numbers = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

prime_count = 0

for num in numbers:
    if is_prime(num):
        prime_count += 1

print(f"Number of prime numbers: {prime_count}")
```

**16.Q: Write a program that simulates a simple ATM system. The user must input their PIN correctly (repeated attempts allowed). Use a loop flag to stop asking for the PIN once it is correctly entered.**

**A:**

```
correct_pin = "1234"

pin_attempts = 0

pin_entered = False

while pin_attempts < 3 and not pin_entered:
    pin = input("Enter your PIN: ")
    if pin == correct_pin:
```

```
        print("PIN correct. Access granted.")
        pin_entered = True
    else:
        print("Incorrect PIN. Try again.")
        pin_attempts += 1
if not pin_entered:
    print("Too many incorrect attempts. Access denied.")
```

**17.Q: How can a sentinel value help improve the performance of a program by reducing unnecessary iterations? Provide an example where a sentinel value can be used to stop unnecessary processing of data.**

**A:**
A sentinel value is a special value used to indicate the end of a sequence or signal that no further processing is needed. It can be used to terminate a loop early when a certain condition is met, thereby reducing unnecessary iterations and improving the performance of a program.

- When a program processes large datasets or user inputs, it might not always need to process every element in the sequence. A sentinel value allows the program to detect when to stop processing, which can significantly reduce the number of iterations and, consequently, the computational time.
- By using a sentinel value, the program can terminate a loop as soon as it reaches the desired condition (e.g., finding a specific value or reaching an end point), thus avoiding further unnecessary checks or computations.

## Example: Using a Sentinel Value to Stop Processing

Suppose you are processing a list of customer orders and you need to find the first "canceled" order. Once a "canceled" order is found, there's no need to continue checking the rest of the orders. By using a sentinel value, the loop can terminate early and prevent unnecessary processing of the remaining orders.

**18:Q: Write a program that takes a list of numbers as input and counts how many of them are even and how many are odd. The program should use a loop counter to iterate through the numbers and a flag to check whether a number is even or odd.**

**A:**

```
N = int(input())

numbers = list(map(int, input().split()))

even_count = 0

odd_count = 0

for num in numbers:

  if num %2 ==0:

     even_count += 1

  else:

     odd_count += 1

print(even_count)

print(odd_count)
```

**19.Q: Discuss the potential drawbacks of using recursive functions to solve problems that can be solved using loops. Why might a loop-based solution be preferred in some cases?**

**A:**

Drawbacks of Using Recursive Functions:

1. High Memory Usage:
   Each recursive call uses memory on the call stack. For deep recursion, this can quickly consume large amounts of memory.

2. Risk of Stack Overflow:
   If recursion is too deep, it may exceed the maximum recursion limit and crash the program with a

stack overflow or RecursionError.

3. Slower Performance:
   Recursive calls involve function call overhead. Loops, on the other hand, are more efficient and faster for most repetitive tasks.

4. Complexity in Debugging:
   Recursive logic can be harder to trace and debug, especially for beginners. Loops tend to follow a more straightforward flow.

5. Limited Recursion Depth:
   Many programming languages, like Python, have a limit on how many recursive calls can be made. Loops do not have this limitation.

Why Loops Are Sometimes Preferred:

- Better performance and efficiency.
- Easier to read and understand.
- Less memory usage.
- No risk of stack overflow.

**20.Q: In a while loop, how would you use a loop flag to terminate the loop once a certain condition is met? Illustrate with a simple example where the program reads user input and exits when a particular keyword is entered.**

**A:**

A loop flag is a boolean variable used to control the execution of a loop. In a while loop, it can be set to True to start the loop and then changed to False when a specific condition is met, which terminates the loop.

Example: User Input with Loop Flag

In this example, the program keeps asking the user for input until the user types "exit". The loop flag running is used to control when the loop should stop.

```
running = True  # Loop flag

while running:

   user_input = input("Enter something (type 'exit' to stop): ")

    if user_input.lower() == "exit":

       print("Exiting the program.")

       running = False  # Set flag to False to terminate the loop

   else:

       print(f"You entered: {user_input}")
```

## MODULE - 3

## Modularity

**1.Q: How does modular programming improve scalability and maintainability?**

**A:** Modularity divides a program into smaller, independent modules. This allows multiple developers to work simultaneously on different features (scalability) and enables easy fixing or updating of parts without affecting the entire system (maintainability).
**Example:** Like a thali, where each dish (module) can be prepared or replaced independently.

**2.Q: Write a program to find the largest among three numbers.**
**A:**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if a >= b and a >= c:
   print("Largest number is", a)
elif b >= a and b >= c:
```

```
    print("Largest number is", b)
else:
    print("Largest number is", c)
```

**3.Q: Write a program to check whether a given year is a leap year or not.**
**A:**

```
year = int(input("Enter a year: "))
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print(year, "is a leap year.")
else:
    print(year, "is not a leap year.")
```

**4. Q: What are the pros and cons of modularity**
**A:**

Pros:

a) Scalability: Building Step by Step : Just like Lego blocks, modular systems can be expanded easily. If a software company wants to add a new feature, they can build a separate module instead of rewriting the entire application.

b) Maintainability: Easier to Fix Issues: Imagine if a car was built as a single solid piece—you'd have to replace the whole thing for a minor issue. But because cars are modular, you can just replace a faulty tire or battery. Similarly, in software, fixing a small module is much easier than debugging an entire system.

c) Reusability: Don't Reinvent the Wheel: Companies save time and effort by reusing existing modules. For example, the same chatbot module can be used across multiple websites instead of creating a new one for each.

d) Flexibility and Customization: With modularity, users can personalize their experience. Gamers, for example, can build custom PCs with specific processors, graphics cards, and RAM, tailoring the machine to their needs instead of buying a one-size-fits-all product.

Cons:

a) Increased Complexity in Integration: While individual modules are easy to develop, integrating them can be tricky. It's like assembling a puzzle—each piece must fit perfectly, and sometimes, different modules don't communicate well with each other, leading to compatibility issues.

b) Performance Overhead: Because modular systems rely on communication between different parts, they may introduce slight inefficiencies. For instance, software built with multiple independent modules may run slightly slower than a monolithic, all-in-one system.

c) Higher Initial Cost and Development Time: Designing a modular system requires careful planning and extra effort upfront. A car manufacturer that wants to make modular engines needs to invest in research and development, which can be costly.

d) Potential Security Risks: Modular systems can sometimes introduce security vulnerabilities. If one module in a software system is compromised, hackers might use it as an entry point to access the entire system. This is why companies constantly update and patch software to close security gaps

**5.Q: Define functions and why functions are required in programming**

**A:**

A function is a block of code that performs a specific task or set of tasks. Functions help organize code, making it reusable, readable, and maintainable. They can take inputs (called parameters or arguments) and return a result (a value)

Eg: def greet(name):

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

Why functions are important

Modularity: Functions allow you to break your program into smaller, manageable pieces. Each function can handle one specific task, making the code easier to understand and debug.

Code Reusability: Functions allow you to define a piece of code once and use it multiple times. You don't have to repeat yourself, which reduces redundancy and makes the code shorter.

For example, if you need to calculate the area of a circle in multiple places, you can write the logic inside a function and call it whenever needed.

**6.Q: Differentiate between call by value and call by reference with examples**

**A**:

**Call by value**

When a function is called with call by value, just a copy of the original value is passed so the original variable will not be changed

Any changes made to this value inside the function do not affect the original value outside the function.

Eg:

```
def add(x):
    x = x + 1
    print(x)
a = 5
add(a)  # Inside the function, x is changed to 6, but a remains 5
print(a)  # Output will be 5, since the original value is unchanged
```

**Call by reference**

When a function is called with call by reference, the variable itself is passed so the original variable may be altered (changed)

Changes made inside the function will affect the original value outside the function.

Eg:

```
def add(arr):
    arr[0] = arr[0] + 1
a = [5]
add(a)  # Inside the function, the first element of the list is changed to 6
print(a)  # Output will be [6], since the original value was modified
```

**7.Q: What is object oriented programming and explain the four pillars of oops**

**A:**

Object-Oriented Programming (OOP) is a way of writing computer programs using objects.

Four pillars of oops :

1. Encapsulation
2. Abstraction

3. Inheritance
4. Polymorphism

1. Encapsulation: It is the process of binding up of data into a single unit

Ex: class Person:

```
    def __init__(self, name, age):
        self.name = name        # public
        self.__age = age        # private
    def display(self):
        print("Name:", self.name)
        print("Age:", self.__age)


p = Person("Alice", 25)
p.display()
# print(p.__age)  #  Error: age is private
```

2. Abstraction is the process of hiding the internal details and showing only the essential features of an object.

Ex:

```
class Animal(ABC):
    def make_sound(self):
        pass
class Dog(Animal):
    def make_sound(self):
        print("Bark")
d = Dog()
d.make_sound()  # Output: Bark
```

3. Inheritance  is a mechanism where one class acquires the properties and behaviors (fields and methods) of another class.

Ex:

```
class Vehicle:
    def start(self):
        print("Engine started")
class Car(Vehicle):  # Inheriting Vehicle
    def drive(self):
        print("Driving...")
c = Car()
c.start()   # From parent class
c.drive()   # From child class
```

4. Polymorphism: allows one method or object to behave in different ways based on the context, typically through method overloading or overriding.

Ex:

Method Overriding (runtime polymorphism)

```
class Bird:
    def speak(self):
        print("Some bird sound")
class Parrot(Bird):
    def speak(self):
        print("Parrot says hello!")
b = Bird()
p = Parrot()
b.speak()  # Output: Some bird sound
p.speak()  # Output: Parrot says hello!
```

Method Overloading

```
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c
```

```
calc = Calculator()
print(calc.add(2, 3))      # Output: 5
print(calc.add(2, 3, 4))   # Output: 9
```

**8.Q: Write a program to check whether a given number is a prime number or not.**

**A :**

```
num = int(input("Enter a number: "))
    if num > 1:
       for i in range(2, int(num ** 0.5) + 1):
          if num % i == 0:
             print(num, "is not a prime number.")
             break
          else:
          print(num, "is a prime number.")
    else:
       print(num, "is not a prime number.")
```

**9.Q: what are access specifiers and the types of access specifiers**

**A:**

Access specifiers in Object-Oriented Programming (OOP) define how class members (attributes and methods) can be accessed. They help in enforcing encapsulation by restricting access to data and ensuring security.

Types:

1.Private (private in C++, __var in Python)

Can only be accessed within the class. Used to hide sensitive data from direct external modification.

2.Protected (protected in C++, _var in Python)

Can be accessed within the class and by subclasses.

3.Public (public in C++, no special syntax in Python)

Can be accessed from anywhere in the program.

Used for methods and attributes that need to be openly accessible.

Ex:

```
class ClassName:
    def __init__(self):
        self.Var = 10
        self._Var = 20
        self.__Var = 30
```

**10.Q: Write a program to find the sum of the squares of the first N natural numbers.**

**A:**

```
n = int(input("Enter a number: "))
sum_squares = n * (n + 1) * (2 * n + 1) // 6
print("Sum of squares of first", n, "natural numbers is", sum_squares)
```

**11.Q: Write a program to generate all prime numbers up to a given number N.**

**A:**

```
n = int(input("Enter a number: "))
print("Prime numbers up to", n, "are:")
for num in range(2, n + 1):
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            break
    else:
        print(num, end=' ')
```

**12.Q: Differentiate between recursion and iteration by explaining their working mechanisms, efficiency, and typical use cases, along with one example for each.**

**A:**

| Feature | Recursion | Iteration |
|---|---|---|
| Approach | Function calls itself | Loops (for, while) |
| Memory Use | More (stack frames) | Less (single loop block) |
| Readability | Often simpler for complex problems | Can be more efficient |
| Speed | Slower in most cases | Usually faster |
| Example | Tower of Hanoi | Fibonacci using loops |

**13.Q: Explain the differences between a constructor and a destructor**

**A:**

**Constructors**

A constructor is a special function that is automatically called when an object is created. It is primarily used to initialize an object's attributes with default or user-defined values.

Key characteristics of constructor

1.Same Name as the Class: The constructor name must match the class name (in C++) or be init (in Python).

2.No Return Type: Constructors do not have a return type in C++ (not even void).

3.Called Automatically: When an object is instantiated, the constructor is invoked without explicitly calling it.

4.Can Have Parameters: Constructors can take arguments to initialize object attributes.

5.Can Be Overloaded (C++ only): Multiple constructors with different parameters can be defined.

**Destructors**

A destructor is a special function that is automatically called when an object is destroyed. It is used to release resources such as dynamically allocated memory, closing file connections, or performing other cleanup tasks.

Key characteristics of destructors:

1.Same Name as the Class (but preceded by ~ in C++, and __del__ in Python).

2. Automatically Called when:

The object goes out of scope (e.g., at the end of a function).

The object is explicitly deleted (for dynamically allocated objects).

The program terminates**.**

**14. Q: Create a BankAccount class with private attributes account_number and balance. Implement a constructor to initialize these values and public methods to deposit, withdraw, and display balance.**

**A:**

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance  # Private attribute
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance!")
    def display_balance(self):
        print(f"Account Number: {self.__account_number}, Balance: {self.__balance}")
```

```
# Using the class
acc1 = BankAccount(12345, 5000)
acc1.deposit(2000)
acc1.withdraw(3000)
acc1.display_balance()
```

**15. Q: Explain the types of recursion with examples**

**A:**

**Types of Recursion in Programming**

**1. Direct Recursion**

When a function **calls itself directly**.

**Example:**

```
def direct_recursion(n):
  if n > 0:
    print(n)
    direct_recursion(n - 1)

direct_recursion(3)
```

**2. Indirect Recursion**

When a function calls **another function**, and that function calls the **original function**.

**Example:**

```
def function_a(n):
  if n > 0:
    print("A:", n)
    function_b(n - 1)

def function_b(n):
  if n > 0:
```

```
    print("B:", n)
    function_a(n - 1)
function_a(3)
```

**3. Tail Recursion**

The **recursive call is the last operation** in the function. Python doesn't optimize tail recursion like some other languages (e.g., Scheme), but it still helps understand the structure.

**Example:**

```
def tail_recursion(n):
  if n == 0:
    return
  print(n)
  tail_recursion(n - 1)
tail_recursion(5)
```

In Python, deep recursion (like 1000+ levels) may cause a RecursionError.

**4. Head Recursion**

The recursive call happens **before any operation** in the function.

**Example:**

```
def head_recursion(n):
  if n == 0:
    return
  head_recursion(n - 1)
  print(n)
head_recursion(5)
```

**5. Tree Recursion**

A recursive function calls **itself more than once** per call, forming a **tree-like structure** of calls.

**Example: Fibonacci**

```
def tree_recursion(n):
  if n <= 1:
```

```
    return n
  return tree_recursion(n - 1) + tree_recursion(n - 2)

print(tree_recursion(5))  # Output: 5
```

Tree recursion is elegant but **inefficient** without memoization or dynamic programming.

**6. Nested Recursion**

The recursive call's **argument** is a result of **another recursive call**.

**Example:**

```
def nested_recursion(n):
  if n > 100:
    return n - 10
  return nested_recursion(nested_recursion(n + 11))
print(nested_recursion(95))  # Output: 91
```

**16.Q: Write a Recursive Function to Calculate the Factorial of a Number**

**A:**

```
def factorial(n):
  if n == 0 or n == 1:  # Base case
    return 1
  else:
    return n * factorial(n - 1)  # Recursive case

# Example
print("Factorial of 5:", factorial(5))  # Output: 120
```

**17. Q: Write a Python function to compute the nth Fibonacci number using recursion.**

**A:**

```
def fibonacci(n):
  if n <= 1:  # Base cases: F(0)=0, F(1)=1
     return n
  else:
     return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

# Example: print first 10 Fibonacci numbers
print("Fibonacci sequence:")
for i in range(10):
  print(fibonacci(i), end=" ")  # Output: 0 1 1 2 3 5 8 13 21 34
```

**18.Q: Write a Python function using recursion to calculate the sum of the first n natural numbers.**

**A:**

```
def sum_natural(n):
  if n == 0:  # Base case
     return 0
  else:
     return n + sum_natural(n - 1)  # Recursive case

# Example
print("Sum of first 10 natural numbers:", sum_natural(10))  # Output: 55
```

**19.Q: Write a program to find the maximum of two numbers.**

**A:**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
```

```
if a > b:
   print("Maximum is", a)
else:
   print("Maximum is", b)
```

**20.Q: Write a Python function using recursion to reverse a given string.**

**A:**

```
def reverse_string(s):
  if len(s) == 0:
     return s
  else:
     return s[-1] + reverse_string(s[:-1])  # Last char + reverse rest

# Example
print("Reversed string:", reverse_string("python"))  # Output: nohtyp
```

**21. Q: Differentiate between local and global scope with examples.**

**A:**

variable is said to have global scope if it is declared outside of all functions.
Such variables can be accessed from anywhere in the code, including inside functions.

```
global_var = 10  # This is a global variable
def test():
   print(global_var)  # You can access global_var here
test()
```

- global_var is defined outside the function.
- It is accessible both inside and outside the function.

A variable declared inside a function is said to have a local scope.
It can only be used within that function and not outside.

```
def test():
    local_var = 5     # This is a local variable
    print(local_var)  # This works
test()
print(local_var)  #  Error: local_var is not accessible outside the function
```

- local_var is defined inside the function.
- It exists only while the function is running.
- Once the function finishes, local_var is deleted from memory.

**22.Q: Explain the concept of instantiation in object-oriented programming. Why is instantiation considered important? Discuss at least two benefits it provides in software development.**

**A:**
Instantiation is the process of creating an object from a class. It allows us to use the attributes and methods defined within the class. Each object created from a class has its own copy of the attributes and can interact with methods independently.
Why is Instantiation Important?
Encapsulation: Each object holds its own data, ensuring separation of concerns.
Code Reusability: The same class can be used to create multiple objects with different values.
Modularity: Objects interact with each other without affecting the core class definition.
Data Integrity: Each object maintains its own state, reducing unintended data modification

**23. Q: What is a scope and why scope is important?**

**A**:
Scope defines the region of a program where a variable is accessible. Variables can have different scopes, such as:
Local Scope means that the variable is accessible only within a specific block (e.g., inside a function).
Global Scope means that the variable is accessible throughout the program.
Why scope is important :
1.Scope is crucial because it prevents confusion by avoiding naming conflicts between variables used in different parts of a program. This means you can use the same variable name in different functions or blocks of code without them interfering with each other.
2. Scope also improves code readability by ensuring that variables are relevant only to the specific
sections of code where they are used. This makes it easier to understand the purpose of each variable and how it's being used

**24.Q: Explain recursion with example and what are its two main components?**

**A:**
**Recursion** is a programming technique where a function calls itself to solve a smaller instance of a problem.
Eg:
def countdown(n):
if n == 0: // base case
print("Blast off!")
else:
print(n)
countdown(n - 1). //recursive case
countdown(5)
Every recursive function must have:

- **base case** – the condition to stop recursion.
- **recursive case** – where the function calls itself.

**25.Q: What are Conditional Expressions?**
**A:**

Conditional expressions are short forms of if-else statements that let you assign a value or perform an action based on a condition, in a single line.

They're also known as ternary operators (because they take 3 parts: condition, true result, false result).
Syntax:
value_if_true if condition else value_if_false
Example 1: Basic usage

```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)  # Output: Adult
```

If age >= 18, then status = "Adult", else status = "Minor".

Example 2: Inside a function

```
def max_num(a, b):
    return a if a > b else b
print(max_num(5, 10))  # Output: 10
```

Returns the greater of the two numbers.

**26. Q: What are Ternary Operators?**

**A:**

A ternary operator is a shortcut for an if-else statement, written in a single line. It is called ternary because it takes three operands:

1. A condition
2. A result if the condition is true
3. A result if the condition is false

**Syntax in Python:**

result = value_if_true if condition else value_if_false

**Example:**

```
x = 10
y = 20
max_value = x if x > y else y
print(max_value)  # Output: 20
```

# MODULE - 4

## Introduction to Data Structures

**1.Q: What is a data structure? Why are data structures important?**
**A:**

A data structure is a way of organizing, storing, and managing data so that it can be accessed and modified efficiently.It helps you group and arrange data logically to perform operations like insertion, deletion, searching, and sorting.

Example: Arrays, Linked Lists, Stacks, Queues, Trees, Graphs

**Why are Data Structures Important?**

Data structures are important because they:

1. **Improve Efficiency**: Help perform operations like search, insert, and delete faster.
2. **Organize Data**: Keep your data structured for easy access and modification.
3. **Save Resources**: Optimize memory and processing time.
4. **Enable Complex Tasks**: Allow you to solve complex problems like shortest path, balanced search, etc.
5. **Form the Basis of Algorithms**: Many algorithms (like sorting, searching) are built using data structures.

**2.Q: What is the difference between primitive and non-primitive data structures?**

**A:**
Primitive Data Structures are the basic building blocks provided by a programming language. They directly operate upon the machine instructions and are simple to use.

Examples:

- Integer (int)
- Float (float)
- Character (char)
- Boolean (bool)

Non-Primitive Data Structures are more complex and are built using primitive data types. They are used to store and organize large amounts of data efficiently.

Examples:

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

**3.Q: What is an array? List its advantages and disadvantages.**
**A:**
An array is a collection of elements stored at contiguous memory locations. All elements are of the same data type.
Advantages:

- Easy to access elements using indices.
- Memory efficient if the number of elements is known in advance.
- Supports random access.

Disadvantages:

- Fixed size (static memory allocation).
- Insertion and deletion are expensive (need shifting).
- Wastage of memory if not fully used.

**4.Q:Explain what are linked list and the types of linked list?**

**A:**

A linked list is a linear data structure where elements (called nodes) are stored in non-contiguous memory locations and are linked together using pointers. Each node typically contains two parts:

1. Data – the value to be stored.
2. Pointer (or Link) – a reference to the next node in the list.

Linked lists allow efficient insertion and deletion of elements without shifting data like in arrays.

Types of Linked Lists:

1. Singly Linked List

    - Each node points to the next node.
    - The last node points to null.
    - Traversal is one-way.
      Example:
      10 → 20 → 30 → null

2. Doubly Linked List

    - Each node points to both the next and previous nodes.
    - Allows two-way traversal.
      Example:
      null ← 10 ⇄ 20 ⇄ 30 → null

3. Circular Linked List

    - Last node links back to the first node.
    - Can be singly or doubly circular.
      Example:
      10 → 20 → 30 → (back to 10)

**5. Q: How does searching and sorting work on arrays?**

**A:**

Searching in Arrays:

- Finding a specific element.
- Linear Search: Checks each element one by one (works on any array).
- Binary Search: Divides the array into halves (works only on sorted arrays).

Sorting in Arrays:

- Arranging elements in ascending or descending order.
- Common methods: Bubble Sort, Selection Sort, Insertion Sort.
- Sorting helps in faster searching and better organization.

**6.Q:Describe types of linked lists**

**1. Single linked list**

**2.Double linked list**

**3.Circular linked list**

**A:**
A Singly Linked List is a type of linked list where each node points to the next node in the sequence. It is called "singly" because each node has only a single pointer pointing to the next node.

Structure:

- Each node has two parts:

    1. Data: Stores the actual value or information.
    2. Next: A pointer/reference to the next node in the list.

- The last node points to `null`, indicating the end of the list.

Operations:

- Insertion: At the beginning, middle, or end of the list.
- Deletion: At the beginning, middle, or end of the list.
- Traversal: Visit each node starting from the head node.

A Doubly Linked List is an extension of the singly linked list where each node contains two pointers:

- One pointing to the next node.
- One pointing to the previous node.

Structure:

- Each node has three parts:
    1. Data: Stores the value.
    2. Next: A pointer/reference to the next node.
    3. Prev: A pointer/reference to the previous node.

- The first node's `prev` pointer and the last node's `next` pointer both point to `null`.

Operations:

- Insertion: At the beginning, middle, or end of the list (easy insertion at both ends).
- Deletion: At the beginning, middle, or end of the list.
- Traversal: Can be traversed in both forward and backward directions.

A Circular Linked List is a variation of a linked list where the last node points back to the first node, instead of pointing to null.

Types of Circular Linked Lists:

1. Singly Circular Linked List:
    - The last node's next pointer points to the first node.
    - Only one-way traversal (like singly linked list).

2. Doubly Circular Linked List:
    - The last node's next pointer points to the first node and the first node's prev pointer points to the last node.
    - Two-way traversal (like doubly linked list, but circular).

Operations:

- Insertion: Can be performed at any position, especially at the beginning or end.
- Deletion: Can be done at any position
- Traversal: Starts from the head node and can keep going in a loop.

**7.Q:Define a stack. Explain LIFO principle with an example.**

**A:**
A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. In a stack, the last element added (pushed) is the first one to be removed (popped). It works similarly to a stack of plates where you place new plates on top, and you always remove the top plate first.

LIFO Principle (Last In, First Out)

The LIFO principle means that the last element that is inserted (pushed) into the stack will be the first one to be removed. It works like a stack of items where you always remove the most recently added item first.

Eg:

```
stack = []

# Push items onto the stack

stack.append(10)  # Push 10
```

```
stack.append(20)  # Push 20

stack.append(30)  # Push 30

# Pop the top item from the stack

print(stack.pop())  # Output: 30 (Last item pushed)

# Pop again

print(stack.pop())  # Output: 20

# Stack after popping

print(stack)  # Output: [10]
```

**8.Q:What is a circular queue and why is it needed?**

**A:**

A Circular Queue is a type of queue in which the last element points back to the first element, creating a circular structure. This is an improvement over a regular queue, which might leave gaps between elements when items are dequeued. The circular nature ensures that when space is available, the queue can reuse previously vacated spots.

A circular queue is useful because it overcomes the limitations of a regular queue, where there may be unused space after multiple elements are dequeued. This can be inefficient in applications where the queue operates in a continuous loop, such as in buffering, scheduling tasks, and network packet handling.

Advantages of Circular Queue:

1. Efficient Memory Usage: It efficiently utilizes the available space by reusing vacated spaces.
2. Prevents Wastage: In a linear queue, after dequeuing elements, you might end up with unused spaces at the front even though the rear might still have space to accommodate new elements. In a circular queue, this does not happen.
3. Improved Performance: It is ideal in scenarios that require a continuous, fixed-size structure, such as in round-robin scheduling and multitasking systems.

**9.Q:List real-world applications of graphs.**

A:

Graphs are widely used in various real-world applications across different fields, as they help represent and analyze complex relationships between objects. Here are some key real-world applications of graphs:

## 1. Social Networks

- Application: Facebook, Twitter, Instagram
- Explanation: Users are represented as vertices, and relationships (friendships or follows) are represented as edges. Graphs are used to find connections, suggest friends, or detect communities within a network.

## 2. Google Maps/Navigation

- Application: Route finding, GPS navigation
- Explanation: Locations (cities, intersections) are vertices, and roads are edges. Graph algorithms (like Dijkstra's algorithm) help find the shortest path between two points.

## 3. Computer Networks

- Application: Network routing, packet switching
- Explanation: Devices (routers, switches) are represented as vertices, and connections between them (communication lines) are edges. Algorithms like Bellman-Ford help in finding the best route for data transmission.

**10.Q: A university stores student roll numbers in an array. How would you efficiently search for a student if the roll numbers are sorted?**

A:
Use binary search for efficient searching, which reduces the time complexity to O(log n) compared to linear search (O(n)).

# Code

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
```

```
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

**11.Q: Write a program to insert a node at the beginning of a singly linked list.**

**A:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
def insert_at_beginning(self, new_data):
        new_node = Node(new_data)
        new_node.next, self.head = self.head, new_node
def print_list(self):
        temp = self.head
```

```
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
# Example usage
linked_list = SinglyLinkedList()
linked_list.insert_at_beginning(10)
linked_list.insert_at_beginning(20)
linked_list.insert_at_beginning(30)
linked_list.print_list()  # Output: 30 -> 20 -> 10 -> None
```

**12.Q: Write a program to find the length of a singly linked list.**

**A:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
 # Function to insert a node at the end of the list
    def insert_at_end(self, new_data):
        new_node = Node(new_data)
        if not self.head:
            self.head = new_node
```

```
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
    # Function to find the length of the linked list
    def length(self):
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next
        return count
# Example usage
linked_list = SinglyLinkedList()
linked_list.insert_at_end(10)
linked_list.insert_at_end(20)
linked_list.insert_at_end(30)

# Find the length of the linked list
print("Length of the linked list:", linked_list.length())  # Output: 3
```

**13.Q:Write a program to reverse a singly linked list.**

**A:**

```
class Node:
    def __init__(self, data):
        self.data, self.next = data, None
class SinglyLinkedList:
    def __init__(self):
        self.head = None


def insert_at_end(self, data):
        if not self.head: self.head = Node(data)
        else:
            last = self.head
            while last.next: last = last.next
            last.next = Node(data)
    def reverse(self):
        prev, current = None, self.head
        while current:
            current.next, prev, current = prev, current, current.next
        self.head = prev
    def print_list(self):
        temp = self.head
        while temp: print(temp.data, end=" -> "); temp = temp.next
        print("None")
```

```
# Example usage

linked_list = SinglyLinkedList()

for i in [10, 20, 30, 40, 50]: linked_list.insert_at_end(i)

print("Before reversing:")

linked_list.print_list()

linked_list.reverse()

print("After reversing:")

linked_list.print_list()
```

**14.Q: Write a program to reverse a queue using recursion.**

**A:**

```
from collections import deque

# Function to reverse the queue using recursion

def reverse_queue(queue):

    if len(queue) == 0:

        return

    # Remove the front element

    front = queue.popleft()

    # Recursively reverse the rest of the queue

    reverse_queue(queue)

    # Add the removed element to the back of the queue

    queue.append(front)

# Example usage

queue = deque([10, 20, 30, 40, 50])

print("Before reversing:", list(queue))
```

```
# Reverse the queue using recursion
reverse_queue(queue)
print("After reversing:", list(queue))
```

**15.Q: Implement a stack using an array and perform push and pop operations.**

**A:**

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, item): self.stack.append(item)
    def pop(self): return self.stack.pop() if self.stack else "Stack is empty"
    def peek(self): return self.stack[-1] if self.stack else "Stack is empty"
    def is_empty(self): return len(self.stack) == 0
    def display(self): print("Stack:", self.stack if self.stack else "Stack is empty")

# Example usage
stack = Stack()
stack.push(10); stack.push(20); stack.push(30)
stack.display()  # Output: Stack: [10, 20, 30]
print("Popped item:", stack.pop())  # Output: Popped item: 30
stack.display()  # Output: Stack: [10, 20]
print("Top item:", stack.peek())  # Output: Top item: 20
```

**16.Q: Write a program to find the height of a binary tree.**

**A:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to find the height of the binary tree
def height(root):
    if root is None:
        return 0
    else:
        # Compute the height of each subtree and return the larger one
        left_height = height(root.left)
        right_height = height(root.right)
        return max(left_height, right_height) + 1

# Example usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(6)
print("Height of the binary tree:", height(root))  # Output: 4
```

**17.Q: Write a program to count the number of leaves in a binary tree.**

**A:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to count the number of leaf nodes in the binary tree
def count_leaves(root):
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    # Recursively count the leaves in left and right subtrees
    return count_leaves(root.left) + count_leaves(root.right)

# Example usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
print("Number of leaf nodes:", count_leaves(root))  # Output: 3
```

**18.Q: Write a program to represent a graph using an adjacency matrix.**

**A:**

```
class Graph:

    def __init__(self, vertices):

        self.V = vertices  # Number of vertices

        self.matrix = [[0] * vertices for _ in range(vertices)]  # Create a VxV matrix filled with 0

# Function to add an edge between two vertices

    def add_edge(self, u, v):

        self.matrix[u][v] = 1

        self.matrix[v][u] = 1  # For undirected graph, add the reverse edge

# Function to display the adjacency matrix

    def display(self):

        for row in self.matrix:

            print(row)

# Example usage

graph = Graph(5)  # Create a graph with 5 vertices

# Adding edges

graph.add_edge(0, 1)

graph.add_edge(0, 4)

graph.add_edge(1, 2)

graph.add_edge(1, 3)

graph.add_edge(3, 4)

# Display the adjacency matrix

print("Adjacency Matrix:")

graph.display()
```

**19.Q: Write a program to find the common elements between two arrays.**

**A:**

```
def find_common_elements(arr1, arr2):
    # Convert both arrays to sets
    set1 = set(arr1)
    set2 = set(arr2)
  # Find the intersection of the two sets
    common_elements = set1.intersection(set2)
    return list(common_elements)
# Example usage
arr1 = [1, 2, 3, 4, 5]
arr2 = [3, 4, 5, 6, 7]
common_elements = find_common_elements(arr1, arr2)
print("Common elements:", common_elements)
```

**20.Q: Write a program to find the second largest element in an array**.

A:

```
def second_largest(arr):
    # Initialize the two largest values
    first, second = float('-inf'), float('-inf')
# Iterate through the array to find the first and second largest
    for num in arr:
        if num > first:
            second = first
            first = num
```

```
        elif num > second and num != first:
            second = num
    # If second is still -inf, it means there was no second largest element
    if second == float('-inf'):
        return "No second largest element"
    return second
# Example usage
arr = [10, 20, 4, 45, 99]
print("Second largest element:", second_largest(arr))  # Output: 45
```

## MODULE - 5

## Using Generative Al

**1. What is meant by "breaking down the problem" in programming?**

**A:**

Breaking down a problem means dividing a large, complex problem into smaller, manageable parts or modules that are easier to solve individually.

**2. What are programming patterns?**

**A:**

Programming patterns are reusable solutions to common problems that occur in software design, such as Singleton, Factory, and Observer patterns.

**3. You are tasked with building a student registration system in Python. How would you break down the problem for easier AI-based code generation?**

**A:**

**Break into modules like:**

- Collect student information
- Validate input data
- Store data in a file or database
- Display registered students

**4. You want AI to generate a function that sorts names alphabetically. How will you guide it?**

**A:**

Prompt:

"Write a Python function that accepts a list of names and returns the list sorted alphabetically using the sorted() function."

**5. How can AI agents help in writing code functionality?**

**A:**

AI agents can automate code generation by understanding prompts, optimizing performance, suggesting corrections, and even writing unit tests.

**6. Give one example of a basic prompt to generate a C program.**

**A:**

"Write a C program to add two integers entered by the user."

**7. What is the importance of specifying input-output format in prompts?**

**A:**

Specifying input and output format helps AI understand exactly what the program should accept and produce, ensuring correct functionality.

**8. Explain the steps to break down a problem before generating a prompt for code.**

**A:**

- Understand the final goal
- Identify sub-tasks
- Arrange tasks in logical order

- Choose the programming language
- Frame precise prompts for each task

Example: For a student database — collect input → validate input → store data → retrieve data.

**9. Write a prompt for AI to create a Python program that finds the largest of three numbers.**

**A:**

Prompt:

"Create a Python program that accepts three numbers from the user and prints the largest one using if-else statements."

**10. How does prompt engineering differ for C and Python languages?**

**A:**

- In C, prompts need to include data types and memory handling.
- In Python, prompts focus more on logical flow and in-built functions because Python handles memory dynamically.

**11. Mention 3 challenges in using AI agents for real-world coding projects.**

**A:**

- Lack of deep domain knowledge
- Possible security vulnerabilities in generated code
- Need for human validation and optimization

**12. Write a short prompt to generate a login authentication function in Python.**

**A:**

Prompt:

"Write a Python function that asks for a username and password. If both match the predefined ones, print 'Login Successful' else 'Login Failed'."

**13. How does breaking a problem improve the accuracy of AI-generated code? Give an example.**
**Breaking a problem clarifies requirements and allows step-by-step generation.**

**A:**
Example:
Instead of prompting "Build an ATM system," prompt separately for:

- PIN verification
- Balance checking
- Deposit and Withdraw operations

This ensures AI focuses on one module at a time, reducing complexity and errors.

**14. List 5 best practices for writing prompts for generating functional C programs using AI.**

**A:**

- Be specific about the task
- Mention input and output clearly
- Specify if loops, conditionals, or arrays should be used
- Request modular code (functions)
- Ask for basic error handling (e.g., invalid input checking)

**15. Explain with example: How AI agents can write Python code with minimum human intervention?**

A:
AI agents like OpenAI Codex can take detailed prompts and generate almost production-level code.
**Example Prompt:**
"Create a Python Flask API with two endpoints: /hello returning Hello, and /add accepting two numbers and returning their sum."
The agent generates the full working code without manual coding, only minor corrections may be needed.

**16. Build a full prompt for AI to create a Student Grade Management System in Python. Also explain how to break the task into smaller modules.**

**A:**

Breakdown:

- Accept student details
- Enter grades for subjects
- Calculate average and assign grade (A, B, C, etc.)
- Display results

PromptExample:

"Create a Python program where users can input student names and marks in 5 subjects, calculate average marks, and assign grades according to the average."

Explanation:

Breaking tasks ensures each functionality (input, process, output) is separately handled by AI with minimal confusion.

**17. Discuss Prompt Engineering strategies when generating Python vs C programs. Include one example for each.**

**A:**

For Python:

- Focus on logic, use in-built methods, request shorter, cleaner syntax.

  Example: "Create a Python function to remove duplicates from a list using sets."

For C:

- Specify data types, manual memory management, and precise input/output formats.

  Example: "Write a C program to input an array of integers and display the reverse array."

**18. Describe a real-world scenario where an AI agent could automate a complex task by generating code. Give detailed explanation.**

**A:**

Scenario: **Automated invoice generation for an e-commerce site**

- Input: Customer details, purchased items, prices
- Process: Calculate total, taxes, generate invoice template
- Output: Save invoice as PDF or send via email

AI agents can generate Python scripts using libraries like ReportLab or PDFkit based on specific prompts.

**19. Write a detailed step-by-step plan on how you would use an AI agent to generate code for an Employee Payroll System.**

**A:**

Step 1: Break down features:

- Add employee
- Calculate salary based on hours worked
- Apply deductions and bonuses

Step 2: Write specific prompts for each module.
Step 3: Integrate modules manually or via AI.
Step 4: Test each feature independently.

**20. With suitable examples, explain how prompt tuning (modifying prompts) improves the final AI output in code generation tasks.**

**A:**

Example:
Bad Prompt:
"Write a Python program for numbers." → Output unclear.

Good Prompt:
"Write a Python program that accepts a list of numbers and returns their squares using list comprehension." → Output is correct and specific.

Prompt tuning narrows down the expected output, ensuring better quality, fewer mistakes, and faster development.

WEB REFERENCE LINKS :

1.https://www.geeksforgeeks.org/loops-and-control-statements-continue-break-and-pass-in-python/

2.https://www.geeksforgeeks.org/conditional-statements-in-python/

3.https://www.geeksforgeeks.org/understanding-code-reuse-modularity-python-3/

4. https://www.youtube.com/watch?v=pkYVOmU3MgA

5.https://www.youtube.com/watch?v=bZzyPscbtI8