K M M INSTITUTE OF POSTGRADUATE STUDIES

(Affiliated to Sri Venkateswara University, Tirupati)
RAMIREDDIPALLE, TIRUPATI-517 102

DEAPARTMENT OF COMPUTER APPLICATIONS

CERTIFICATE

REGISTERED NO: _____

This is to certify that _____ *is a bonafide student of MCA (I Semester) course in our Institution. The student has done the Lab work as per this record for this practical fulfillment of the requirement of the MCA course* ***MCA107P-Programming in Python (Subject Code & Subject Title)*** *during the* ***I*** *Semester of the Academic Year 2025-2027.*

Head-of the Department *Lecturer-In-Charge*

Attended and submitted for the University Practical Examination.

Signature of Examiners: *1.*_____
(External Examiner-1)

*2.*_____
(External Examiner-2)

CONTENTS

1. **Write a Python Program to demonstrate shift operators.**

Aim: To Write a Python Program to demonstrate shift operators.

Description:

Python bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators. The result is then returned in decimal format.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Class Fast Calculator to perform multiplication and division by powers of 2 using only the bitwise shift operators. This application demonstrates how shift operators can be used as a low-level, high-performance optimization trick in scenarios where you need to repeatedly multiply or divide by powers of two.

Source Code:

```
classFastCalculator:
    """
    A class to demonstrate fast arithmetic operations
    (multiplication and division) using bitwise shift operators.
    """
def __init__(self, number):
            self.number = number
            print(f"Initial Number: {self.number} (Binary: {bin(self.number)})")

deffast_multiply_by_power_of_two(self, power):
        """
        Multiplies the stored number by 2^power using the left shift operator (<<).
        (number * 2^power) == (number << power)
        """
        # We shift 'power' bits to the left
result = self.number<< power
print(f"\n--- Fast Multiplication by 2^{power} (which is {2**power}) ---")
print(f"Operation: {self.number} << {power}")
print(f"Result (Decimal): {result}")
print(f"Result (Binary): {bin(result)}")
return result
deffast_divide_by_power_of_two(self, power):
        """
Divides the stored number by 2^power using the right shift operator (>>).
        (number // 2^power) == (number >> power)
        Note: This is integer division, truncating any remainder.
        """
        # We shift 'power' bits to the right
result = self.number>> power
print(f"\n--- Fast Division by 2^{power} (which is {2**power}) ---")
print(f"Operation: {self.number} >> {power}")
print(f"Result (Decimal): {result}")
print(f"Result (Binary): {bin(result)}")
return result
```

output:

```
# 1. Create an instance with a starting number
initial_value = 48
calculator = FastCalculator(initial_value)

# 2. Fast Multiplication (x 8)
# We want to multiply by 8. Since 8 = 2^3, the power is 3.
# 120 * 8 = 960
calculator.fast_multiply_by_power_of_two(power=3)

# 3. Fast Division (// 16)
# We want to divide by 16. Since 16 = 2^4, the power is 4.
# 120 // 16 = 7 (integer division)
calculator.fast_divide_by_power_of_two(power=4)
```

Conclusion:

Above application demonstrated how shift operators can be used as a low-level, high-performance optimization trick in scenarios where you need to repeatedly multiply or divide by powers of two.

2. **Write Python program to find roots of Quadratic Equation.**

Aim: To Write Python program to find roots of Quadratic Equation.

Description:

A quadratic equation is a polynomial equation of the second degree. It is written in the standard form:

ax^2 + bx + c = 0

The value of the discriminant determines the **nature** and **number** of the roots.

Using if statement
if statement is a conditional statement. It is used to execute a block of code only when a specific condition is met.

Syntax
```
if condition:
   # body of if statement
else:
   # body of else statement
```

Algorithm:

Step 1: Identify the Coefficients

Identify and write down the numerical values for the three coefficients:

- a: The coefficient of the x^2 term.
- b: The coefficient of the x term.
- c: The constant term.

Example: If the equation is 2x^2 - 3x - 5 = 0,then a=2,b=-3,and c=-5.

Step 2: Calculate the Discriminant (\Delta)

Calculate the value of the discriminant using the formula:

\Delta = b^2 - 4ac

Substitute the values of a, b, and c from Step 1 into this formula and compute the result.

Step 3: Analyze the Value of the Discriminant

Compare the calculated value of Delta to zero to determine the nature of the roots. This is the primary decision point of the algorithm.

Case A: If Delta>0 (The Discriminant is Positive)

Conclusion: The quadratic equation has **Two Real and Distinct Roots**.

- **Meaning:** There are two different, unique real number solutions for x.
- **Graphical Implication:** The parabola crosses the x-axis at two distinct points.

Case B: If Delta = 0 (The Discriminant is Zero)

Conclusion: The quadratic equation has **One Real and Equal Root** (a single repeated root).

- **Meaning:** There is exactly one real number solution for x.
- **Graphical Implication:** The parabola touches the x-axis at exactly one point, which is the vertex.

Case C: If Delta <0 (The Discriminant is Negative)

Conclusion: The quadratic equation has **Two Complex Conjugate Roots** (Imaginary Roots).

- **Meaning:** There are no real number solutions for x. The solutions involve the imaginary unit i.
- **Graphical Implication:** The parabola never intersects the x-axis.

Source code:

```
import math
defsolve_quadratic_equation(a, b, c):
    """
    Solves for the roots of a quadratic equation in the form ax^2 + bx + c = 0
    using the quadratic formula: x = (-b +/- sqrt(b^2 - 4ac)) / 2a

    Args:
        a (float): The coefficient of the x^2 term.
        b (float): The coefficient of the x term.
        c (float): The constant term.

    Returns:
        tuple: A tuple containing the two roots (x1, x2).
    """
    print(f"\nSolving the equation: {a}x^2 + {b}x + {c} = 0")

    # Check if 'a' is zero, which would make it a linear equation
    if a == 0:
        if b == 0:
            return "Error: This is not a valid equation (a and b are zero)."
        else:
            x = -c / b
            return (f"Linear Equation Root: x = {x:.4f}",)

    # Calculate the Discriminant: delta = b^2 - 4ac
    # The discriminant determines the nature of the roots (real or complex).
    discriminant = (b**2) - (4 * a * c)

    print(f"Discriminant (b^2 - 4ac) = {discriminant:.4f}")

    # Case 1: Real and Distinct Roots (Discriminant > 0)
    if discriminant > 0:
        print("Nature of Roots: Real and Distinct (Two unique real roots)")

        # Calculate the square root of the discriminant
        sqrt_discriminant = math.sqrt(discriminant)

        # Calculate the two roots
        x1 = (-b + sqrt_discriminant) / (2 * a)
        x2 = (-b - sqrt_discriminant) / (2 * a)

        return (f"x1 = {x1:.4f}", f"x2 = {x2:.4f}")

    # Case 2: Real and Equal Roots (Discriminant = 0)
    elif discriminant == 0:
        print("Nature of Roots: Real and Equal (One real root/Two identical real roots)")

        # Only one root since the +/- part is zero
        x1 = -b / (2 * a)

        return (f"x1 = x2 = {x1:.4f}",)

    # Case 3: Complex (Imaginary) Roots (Discriminant < 0)
    else:
```

```
print("Nature of Roots: Complex (Two complex conjugate roots)")

# Since discriminant is negative, we take the square root of its absolute value
sqrt_abs_discriminant = math.sqrt(abs(discriminant))

# The complex roots are: (-b/2a) +/- (sqrt(|discriminant|)/2a) * i
real_part = -b / (2 * a)
imaginary_part = sqrt_abs_discriminant / (2 * a)

x1_complex = f"{real_part:.4f} + {imaginary_part:.4f}i"
x2_complex = f"{real_part:.4f} - {imaginary_part:.4f}i"

return (x1_complex, x2_complex)
```

Output:

Solving the equation: 1x^2 + 5x + 6 = 0
Discriminant (b^2 - 4ac) = 1.0000
Nature of Roots: Real and Distinct (Two unique real roots)
Roots for Example 1: ('x1 = -2.0000', 'x2 = -3.0000')

Solving the equation: 1x^2 + -4x + 4 = 0
Discriminant (b^2 - 4ac) = 0.0000
Nature of Roots: Real and Equal (One real root/Two identical real roots)
Roots for Example 2: ('x1 = x2 = 2.0000',)

Conclusion:

The Above Program Successfully Executed.

3. Write Python Program to generate Pascal Triangle.

Aim: To Write Python Program to generate Pascal Triangle.

Description:

Pascal's Triangle is a triangular array of binomial coefficients where each number is the sum of the two numbers directly above it. It starts with '1' at the top and is deeply connected to the expansion of binomials like (x+y)^n and to combinations in probability.

ALGORITHM:

1. Initialize an empty list, `triangle`, to store all rows.
2. Loop from i = 0 up to num_rows - 1 (for each row).
3. Inside the loop, initialize a new list, `current_row`, for the current row i.
4. For each element j in the `current_row`:
   a. If j is the first element (j == 0) or the last element (j == i), the value is 1.
   b. Otherwise, the value is calculated as the sum of the element directly
      above it (j-1) and the element to the left of the one above it (j)
      from the *previous* row (`triangle[i-1]`).
      Value = triangle[i-1][j-1] + triangle[i-1][j]
5. Append the completed `current_row` to the `triangle` list.
6. After the loops complete, return the full `triangle` list.

Source Code:

```
import math
defgenerate_pascals_triangle(num_rows):
    """
    Generates Pascal's Triangle up to the specified number of rows.

    The algorithm relies on the property that each element (except for the
    ones at the edges, which are 1) is the sum of the two elements directly
    above it from the previous row.
   """
    ifnum_rows<= 0:
        return []

    triangle = []

    for i in range(num_rows):
        current_row = []

        # Calculate elements for the current row (Row i)
        for j in range(i + 1):
            if j == 0 or j == i:
                # The first and last elements of every row are always 1
                current_row.append(1)
            else:
                # Every other element is the sum of the two elements above it
                # from the previous row (triangle[i-1])
                prev_row = triangle[i - 1]
                new_element = prev_row[j - 1] + prev_row[j]
                current_row.append(new_element)

        triangle.append(current_row)

    return triangle

defdisplay_pascals_triangle(triangle):
    """Prints the triangle in a visually aligned format."""
    if not triangle:
        print("No rows to display.")
        return

    # Calculate the maximum width needed for alignment
    last_row = triangle[-1]
    # Convert numbers to strings and find the length of the longest element
    max_width = len(str(max(last_row)))

    # Calculate total width for center alignment
    # The total number of elements in the last row is len(last_row)
    # The total width is approximated by (max_width * len(last_row)) + (len(last_row) - 1 for spaces)
```

```
    total_print_width = (max_width + 1) * len(last_row)

    print("\n--- Generated Pascal's Triangle ---")

    for row in triangle:
        row_str = ""
        # Format each number with the calculated max_width
        fornum in row:
            row_str += str(num).center(max_width + 1)

        # Print the row centered based on the width of the last row
        print(row_str.center(total_print_width))
    print("-----------------------------------")

# --- Example Usage ---

ROWS_TO_GENERATE = 10
print(f"Generating Pascal's Triangle with {ROWS_TO_GENERATE} rows...")

# 1. Generate the data structure
p_triangle = generate_pascals_triangle(ROWS_TO_GENERATE)

# 2. Display the result in a clean format
display_pascals_triangle(p_triangle)

# 3. Example of using the stored list
print("\nList Representation of the Triangle (first 5 rows):")
for i in range(min(5, len(p_triangle))):
    print(f"Row {i}: {p_triangle[i]}")
```

Output:

Generating Pascal's Triangle with 10 rows...

--- Generated Pascal's Triangle ---

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  2015  6   1
  1   7   21  3535  21  7   1
1   8   28  5670  5628  8   1
1   9   36  84 126 126  8436  9   1
-----------------------------------
```

List Representation of the Triangle (first 5 rows):
Row 0: [1]
Row 1: [1, 1]
Row 2: [1, 2, 1]
Row 3: [1, 3, 3, 1]
Row 4: [1, 4, 6, 4, 1]

Conclusion:

The Above Program Successfully Executed.

4. Write python program to find repeated words in give lines of text count words that are repeated anywhere within a combined set of input lines.

Aim: To Write python program to find repeated words in give lines of text count words that are repeated anywhere within a combined set of input lines.

Description:

Step-by-Step Algorithm

1. **Initialization:**
    - Define a function, count_repeated_words(num_lines).
    - Initialize an empty list, all_words, to store every single word token from all inputs.
2. **Input and Collection Loop (Repeat num_lines times):**
    - For each input line required: a. **Get Input:** Prompt the user to enter a line of text. b. **Normalize Text:** Convert the entire input line to **lowercase** for case-insensitive counting. c. **Tokenize:** Use a word extraction utility (like re.findall) to create a list of all word tokens, removing punctuation. d. **Aggregate:** Add the list of words from the current line to the master all_words list.
3. **Frequency Counting:**
    - Use Python's collections.Counter (or build a standard dictionary manually) on the all_words list.
    - This process iterates through the entire list and creates a map where the **key is the unique word** and the **value is its total count** across all lines.
4. **Filter Results:**
    - Iterate through the frequency map generated in Step 3.
    - Select only those entries (word-count pairs) where the **count is greater than 1**. This isolates the words that were actually repeated.
5. **Output:**
    - Sort the filtered results alphabetically by the word.
    - Print each repeated word along with its total count.
    - If no words have a count greater than 1, report that no words were repeated.

Source Code:

```
import re
defcount_repeated_words(num_lines=3):
    """
    Reads a specified number of input lines and counts the frequency of every
    word (case-insensitive) using only basic Python structures (no external
    libraries like re or collections.Counter).
    """
    print(f"--- Enter {num_lines} lines of text below ---")

    # Use a standard dictionary to store word frequencies
    word_counts = {}

    # List of common punctuation marks to remove/replace
    punctuation = ['.', ',', '!', '?', ':', ';', '(', ')', "", """]

    # 1. Collect and Normalize Input
    for i in range(1, num_lines + 1):
        line_input = input(f"Enter Line {i}: ")

        # Convert to lowercase for case-insensitive counting
        normalized_text = line_input.lower()

        # Simple punctuation removal: replace common punctuation with a space
        for p in punctuation:
            normalized_text = normalized_text.replace(p, ' ')

        # Split the string by whitespace to get words (handles multiple spaces)
        words = normalized_text.split()

        # 2. Count Word Frequencies using a standard dictionary
        for word in words:
            # Use .get(key, default) to simplify the counting logic:
            # If the word is already in the dictionary, increase its count;
            # otherwise, start the count at 1.
            word_counts[word] = word_counts.get(word, 0) + 1

    print("\n--- Processing Complete ---")

    if not word_counts:
        print("No words were entered.")
        return {}

    # 3. Identify and Collect Repeated Words
    repeated_words = {}
    for word, count in word_counts.items():
        # A word is repeated if its count is greater than 1
        if count > 1:
            repeated_words[word] = count

    # 4. Output the Results
    ifrepeated_words:
        # Sort the results alphabetically by word for clean output
        # Using sorted() on dictionary items returns a list of tuples
```

```
        sorted_repeated_words_list = sorted(repeated_words.items())

        print(f"Words repeated (count > 1) across all {num_lines} lines:")

        for word, count in sorted_repeated_words_list:
            print(f"-> '{word}' appears {count} times")

        returndict(sorted_repeated_words_list)
    else:
        print("No words were repeated across the combined input.")
        return {}

# Run the program with the required 3 lines of input
count_repeated_words(3)
```

Output:

```
Enter 3 lines of text below ---
Enter Line 1:  college education is good and useful
Enter Line 2:  computer science is useful for future
Enter Line 3:  all are joining computer science

--- Processing Complete ---
Words repeated (count > 1) across all 3 lines:
-> 'computer' appears 2 times
->'is' appears 2 times
-> 'science' appears 2 times
-> 'useful' appears 2 times
{'computer': 2, 'is': 2, 'science': 2, 'useful': 2}
```

Conclusion:

The program executed successfully and returned number of repeated words and its count.

5. Write a Python program using functions to find Largest and smallest of 5 elements.

Aim: To Write a Python program using functions to find Largest and smallest of 5 elements.

Description:

Given python application is designed to collect a fixed number of inputs (defaulting to five) from the user and determine the minimum and maximum values among them.
Key Features

- Functional Decomposition: The program is divided into two distinct functions for clarity and reusability:
    1. get_input_list(): Handles all user interaction, ensuring that the necessary number of valid numerical inputs are collected.
    2. find_min_max(): Performs the core logic of finding the extreme values.

Algorithm:

A. Input Collection Function (get_input_list)

1. Initialization: Start an empty list, input_list, to store the numbers.
2. Loop for Input: Repeat the input process until the desired number of elements (N, typically 5) has been collected.
3. Prompt and Validate: a. Prompt the user to enter a number. b. Attempt to convert the input into a floating-point number. c. If conversion fails (due to invalid characters), display an error and loop again without incrementing the count. d. If conversion succeeds, append the number to input_list.
4. Return: Return the completed list of numbers.

B. Min/Max Finding Function (find_min_max)

1. Check for Empty List: Check if the input list is empty. If it is, return None for both minimum and maximum.
2. Find Maximum: Use the built-in Python function max() on the input list to efficiently determine the largest element. Store the result in a variable named largest.
3. Find Minimum: Use the built-in Python function min() on the input list to efficiently determine the smallest element. Store the result in a variable named smallest.
4. Return: Return the smallest and largest values as a tuple.

C. Main Program Flow

1. Execute the **Input Collection Function (A)** to retrieve the 5 user-provided numbers.
2. Execute the **Min/Max Finding Function (B)**, passing the list of numbers.
3. Display the original input list and the returned smallest and largest values.

Source Code:

```
deffind_min_max(numbers):
    """
    Finds the smallest and largest element in a list of numbers using built-in
    Python functions (min() and max()).
    Args:
        numbers (list): A list of numerical elements.
    Returns:
        tuple: A tuple containing (smallest, largest) element, or None if the list is empty.
    """
    if not numbers:
        return None, None
    # Find the largest element
    largest = max(numbers)

    # Find the smallest element
    smallest = min(numbers)
        return smallest, largest
defget_input_list(count=5):
    """
    Prompts the user to input a specified number of numerical elements.
    Includes basic error handling for non-numeric input.
    """
    input_list = []
    print(f"\n--- Enter {count} numbers ---")

    whilelen(input_list) < count:
        try:
            # Prompt user for the current element
            prompt = f"Enter number {len(input_list) + 1} of {count}: "
            value = float(input(prompt))
            input_list.append(value)
        exceptValueError:
            print("Invalid input. Please enter a valid number.")
                returninput_list
# --- Main Program Execution ---

# 1. Get the list of 5 numbers from the user
NUM_ELEMENTS = 5
data = get_input_list(NUM_ELEMENTS)

# 2. Call the function to find the minimum and maximum
min_val, max_val = find_min_max(data)

# 3. Output the results
ifmin_val is not None:
    print("\n--- Results ---")
    print(f"Input Data: {data}")
    print(f"The Smallest Element is: {min_val}")
    print(f"The Largest Element is: {max_val}")
else:
    print("Failed to process input.")
```

Output:

```
--- Enter 5 numbers ---
Enter number 1 of 5:  45
Enter number 2 of 5:  12
Enter number 3 of 5:  5
Enter number 4 of 5:  7
Enter number 5 of 5:  10

--- Results ---
Input Data: [45.0, 12.0, 5.0, 7.0, 10.0]
The Smallest Element is: 5.0
The Largest Element is: 45.0
```

Conclusion:

The Above Program Successfully Executed.

6. Write a Python program using functions to find LCM and GCM.

Aim: To Write a Python program using functions to find LCM and GCM.

Description:

A.Function: Calculate GCD (calculate_gcd(a, b))

The GCD is found using the highly efficient **Euclidean Algorithm**.

1. **Normalization:** Ensure both input integers (aand b) are positive (take the absolute value).
2. **Iterative Calculation:** Initiate a loop that continues as long as b is not zero. a. Calculate the remainder of a divided by b (i.e., a \pmodb). b. Simultaneously update a to the current value of b, and update b to the calculated remainder. c. **Loop Termination:**When b finally becomes 0, the value currently held by a is the GCD.
3. **Return:** Return the final value of a as the GCD.

B. Function: Calculate LCM (calculate_lcm(a, b))

The LCM is derived from the GCD using a fundamental mathematical relationship.

1. **Zero Check:** If either aor b is zero, the LCM is 0.
2. **Find GCD:** Call the calculate_gcd(a, b) function to determine the GCD of the two numbers.
3. **Apply Formula:** Calculate the LCM using the formula:

{LCM}(a, b) = {|a X b|}/{{GCD}(a, b)}

Note: Integer division (// in Python) is used to ensure the result is an integer.

4. **Return:** Return the calculated LCM value.

C. Main Program Flow

1. **Input:** Prompt the user to enter two non-negative integers (aand b) and validate the input (ensuring they are integers).
2. **Execution:** Call the calculate_gcd(a, b) function and store the result.
3. **Execution:** Call the calculate_lcm(a, b) function and store the result.
4. **Output:** Display the calculated GCD and LCM to the user.

Source Code:

```
import math
defcalculate_gcd(a, b):
    """
    Calculates the Greatest Common Divisor (GCD) of two integers using
    the Euclidean Algorithm (iterative method).

    The Euclidean algorithm is based on the principle that the GCD of two numbers
    does not change if the larger number is replaced by its difference with
    the smaller number.

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The Greatest Common Divisor.
    """
    # Ensure inputs are non-negative for the algorithm
    a = abs(int(a))
    b = abs(int(b))

    # Loop until one of the numbers becomes 0
    while b:
        # Standard Euclidean Algorithm step:
        # Replace 'a' with 'b' and 'b' with the remainder of (a / b)
        a, b = b, a % b

    return a

defcalculate_lcm(a, b):
    """
    Calculates the Least Common Multiple (LCM) of two integers.

    It uses the relationship: LCM(a, b) = (|a * b|) / GCD(a, b).

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The Least Common Multiple.
    """
    # Handle the trivial case where either number is zero
    if a == 0 or b == 0:
        return 0

    # Calculate GCD using the separate function
    gcd = calculate_gcd(a, b)

    # Apply the formula: LCM = (|a * b|) / GCD
    # Use abs() to ensure positive result
    lcm_result = abs(a * b) // gcd
        returnlcm_result
```

```
defget_valid_input():
    """Prompts user for two valid integers."""
    while True:
        try:
            num1 = int(input("Enter the first positive integer (a): "))
            num2 = int(input("Enter the second positive integer (b): "))
            if num1 < 0 or num2 < 0:
                print("Please enter non-negative integers.")
                continue
            return num1, num2
        exceptValueError:
            print("Invalid input. Please enter whole numbers.")

# --- Main Program Execution ---

# 1. Get user input
num_a, num_b = get_valid_input()

# 2. Check for zero case before calculation (though handled inside functions too)
ifnum_a == 0 and num_b == 0:
    print("\nGCD(0, 0) is typically undefined or 0, LCM(0, 0) is 0.")
    print("Results: GCD = 0, LCM = 0")
else:
    # 3. Calculate GCD and LCM using the defined functions
    gcd_result = calculate_gcd(num_a, num_b)
    lcm_result = calculate_lcm(num_a, num_b)

    # 4. Output the results
    print("\n--- Calculation Results ---")
    print(f"Numbers: a = {num_a}, b = {num_b}")
    print(f"Greatest Common Divisor (GCD) is: {gcd_result}")
    print(f"Least Common Multiple (LCM) is: {lcm_result}")
```

Output:

Enter the first positive integer (a): 35
Enter the second positive integer (b): 56

--- Calculation Results ---
Numbers: a = 35, b = 56
Greatest Common Divisor (GCD) is: 7
Least Common Multiple (LCM) is: 280

Enter the first positive integer (a): 78
Enter the second positive integer (b): 12

--- Calculation Results ---
Numbers: a = 78, b = 12
Greatest Common Divisor (GCD) is: 6
Least Common Multiple (LCM) is: 156

Conclusion:

The Above Program Is Successfully Executed.

7. Write Python program to sort elements in List.

Aim: To Write Python program to sort elements in List.

Description:

A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ]

Searching (Existence): The program checks if a specific element is present within the list using the in operator, which is highly efficient.
Searching (Position): It then demonstrates how to find the index (position) of a specific element using the .index() method. This operation is handled within a try...except block to gracefully manage cases where the element is not found, preventing a program crash.

Algorithm:

This algorithm outlines the steps for initializing the list, performing the sorting operations, and executing the search operations.

Initialization and Setup

1. **START**
2. **Initialize** a list variable, my_list, with a set of unsorted integer values.
3. **Output** the my_list to show its original state.

Sorting Operations

4. **Sort Ascending:** Call the **built-in sort method** on my_list (i.e., my_list.sort()) to arrange elements from smallest to largest.
5. **Output** the list after the ascending sort.
6. **Sort Descending:** Call the **built-in sort method** again on my_list, passing the parameter reverse=True (i.e., my_list.sort(reverse=True)) to arrange elements from largest to smallest.
7. **Output** the list after the descending sort.

Searching Operations

8. **Define** a variable, element_to_find, for the first search (existence check).
9. **Search 1 (Existence):**
    - **Check:** Use the if...in structure to determine if element_to_find exists in my_list.
    - **Output:** Print a message indicating whether the element was **found** or **not found**.
10. **Define** a variable, element_to_find_position, for the second search (index check).
11. **Search 2 (Position/Index):**
    a) **Begin** a **try block**.
    b) **Find Index:** Inside the try block, call the .index() method on my_list with element_to_find_position to get its position.
    c) **Output:** Print a message indicating the element and its **index**.
    d) **Begin** an **except ValueError block**.
    e) **Error Handling:** If the .index() call raises a ValueError (meaning the element is not in the list), print a message indicating the element was **not found**.
12. **END**

Source Code:

```
## 1. Get Initial List Input
# The user enters numbers separated by commas (e.g., 50, 20, 90, 10)
input_str = input("Enter list elements (comma-separated numbers, e.g., 50,20,90): ")
try:
   # Convert the comma-separated string input into a list of integers
my_list = [int(item.strip()) for item in input_str.split(',')]
exceptValueError:
print("Invalid input. Please ensure all elements are numbers.")
   # Exit or use a default list if input is invalid
my_list = [50, 20, 90, 10, 60]
print(f"Using default list: {my_list}")
print(f"\nOriginal List: {my_list}")
# --- Sorting the List ---
print("\n--- Sorting Operations ---")
## 2. Sort the List (Ascending)
# The .sort() method modifies the list directly (in-place)
my_list.sort()
print(f"List after Ascending Sort: {my_list}")

## 3. Sort in Descending Order
my_list.sort(reverse=True)
print(f"List after Descending Sort: {my_list}")

# --- Searching for Elements ---
print("\n--- Searching Operations ---")
## 4. Search using the 'in' operator (Checks for existence)
try:
element_to_find = int(input("Enter the element to check for existence (e.g., 40): "))
exceptValueError:
print("Invalid input for search. Using default element 40.")
element_to_find = 40

ifelement_to_find in my_list:
print(f"\nSearch 1: The element {element_to_find} is **found** in the list.")
else:
print(f"\nSearch 1: The element {element_to_find} is **not found** in the list.")

## 5. Search using the .index() method (Finds the position)
try:
element_to_find_position = int(input("Enter the element to find the index of (e.g., 90): "))
exceptValueError:
print("Invalid input for index search. Using default element 90.")
element_to_find_position = 90
try:
   # We use the list that was last sorted (Descending) for this search
index = my_list.index(element_to_find_position)
print(f"Search 2: The element {element_to_find_position} is found at **index {index}** in the Descending list.")
exceptValueError:
print(f"Search 2: The element {element_to_find_position} is not found in the list.")
```

Output:

Enter list elements (comma-separated numbers, e.g., 50,20,90): 30,56,60,70,90

Original List: [30, 56, 60, 70, 90]

--- Sorting Operations ---
List after Ascending Sort: [30, 56, 60, 70, 90]
List after Descending Sort: [90, 70, 60, 56, 30]

--- Searching Operations ---
Enter the element to check for existence (e.g., 40): 56

Search 1: The element 56 is **found** in the list.
Enter the element to find the index of (e.g., 90): 91
Search 2: The element 91 is not found in the list.

Conclusion:

The Above Program Is Successfully Executed.

8. Write Python Program for Matrix multiplication.

Aim: ToWrite Python Program for Matrix multiplication.

Description:

This program implements matrix multiplication, a core operation in linear algebra, using nested Python lists to represent matrices.

1. **Input:** The program first prompts the user to define the **dimensions (rows and columns)** and the **elements** for two matrices, Matrix **A** and Matrix **B**. Input validation ensures elements are integers and that each row has the correct number of columns.
2. **Dimension Check:** The core mathematical requirement for multiplication is checked: the **number of columns in the first matrix {cols} in A** must equal the **number of rows in the second matrix {rows}B**. If this condition is not met, the program raises a Value Error

Algorithm

A. Matrix Input Algorithm (get_matrix_input)

1. **START**
2. **Prompt** the user to enter the number of **rows** and **columns** for the matrix.
3. **Initialize** an empty list, matrix.
4. Loop from $i=0$ up to (number of rows - 1):

   a. Prompt the user to enter space-separated elements for the current row i.

   b. Validate the input: ensure all elements are integers and that the number of elements equals the required number of columns.

   c. If valid, convert the elements to integers and append the resulting list (row) to matrix.

   d. If invalid, prompt the user to re-enter the row.

5. **Return** the final matrix.

B. Matrix Multiplication Algorithm (matrix_multiply)

1. **START**
2. **Get Dimensions:** Determine rows_A, cols_A, rows_B, and cols_B.
3. **Dimension Check:**
   - **IF** $\text{cols}_A \ne \text{rows}_B$: Raise a ValueError (incompatible dimensions).
4. **Initialize Result:** Create the result matrix **C** with dimensions A {cols}_B)$ and fill all elements with zero.
5. for i in range(rows_A):
6. # j iterates through columns of B
7. for j in range(cols_B):
8. # k iterates through columns of A (or rows of B)
9. for k in range(cols_A):
10. C[i][j] += A[i][k] * B[k][j]
11. Return the result matrix C.
12. END

Source Code:

```
defmatrix_multiply(A, B):
   """
Multiplies two matrices, A and B.
   A must have dimensions (m x k) and B must have (k x n).
   The resulting matrix C will have dimensions (m x n).
   """
rows_A = len(A)
cols_A = len(A[0])
rows_B = len(B)
cols_B = len(B[0])

   # Check for valid dimensions (Columns of A must equal Rows of B)
ifcols_A != rows_B:
raiseValueError("Matrix dimensions are incompatible for multiplication. "
                   "Columns of the first matrix must equal rows of the second matrix.")

   # Initialize the result matrix C with dimensions (rows_A x cols_B) filled with zeros
   C = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

   # Perform the matrix multiplication
   # i iterates through rows of A
for i in range(rows_A):
      # j iterates through columns of B
for j in range(cols_B):
         # k iterates through columns of A (or rows of B)
for k in range(cols_A):
            C[i][j] += A[i][k] * B[k][j]

return C

defget_matrix_input(name):
"""Gets matrix dimensions and elements from user input."""
print(f"\n--- Input for Matrix {name} ---")
try:
rows = int(input(f"Enter the number of rows for Matrix {name}: "))
cols = int(input(f"Enter the number of columns for Matrix {name}: "))
exceptValueError:
print("Invalid dimension input. Using default 2x2 matrix.")
rows, cols = 2, 2

matrix = []
print(f"Enter elements for Matrix {name} ({rows}x{cols}):")
for i in range(rows):
while True:
try:
row_str = input(f"Enter elements for row {i+1} (space-separated): ")
row = [int(x) for x in row_str.split()]
iflen(row) == cols:
matrix.append(row)
break
else:
print(f"Error: Row must contain {cols} elements. Try again.")
exceptValueError:
```

```
print("Error: Elements must be integers. Try again.")

return matrix

# Main execution block
try:
# 1. Get input matrices from the user
matrix_A = get_matrix_input("A")
matrix_B = get_matrix_input("B")

print("\nMatrix A:")
for row in matrix_A: print(row)
print("\nMatrix B:")
for row in matrix_B: print(row)

# 2. Perform multiplication
result_matrix = matrix_multiply(matrix_A, matrix_B)

# 3. Print result
print("\n--- Result Matrix (A x B) ---")
for row in result_matrix:
print(row)

exceptValueError as e:
print(f"\nError: {e}")
except Exception as e:
print(f"\nAn unexpected error occurred: {e}")
```

Input and Output:

```
--- Input for Matrix A ---
Enter the number of rows for Matrix A:  3
Enter the number of columns for Matrix A:  3
Enter elements for Matrix A (3x3):
Enter elements for row 1 (space-separated):  5 6 7
Enter elements for row 2 (space-separated):  7 8 9
Enter elements for row 3 (space-separated):  10 11 12

--- Input for Matrix B ---
Enter the number of rows for Matrix B:  3
Enter the number of columns for Matrix B:  3
Enter elements for Matrix B (3x3):
Enter elements for row 1 (space-separated):  2 3 4
Enter elements for row 2 (space-separated):  5 6 7
Enter elements for row 3 (space-separated):  7 6 5

Matrix A:
[5, 6, 7]
[7, 8, 9]
[10, 11, 12]

Matrix B:
[2, 3, 4]
[5, 6, 7]
[7, 6, 5]

--- Result Matrix (A x B) ---
[89, 93, 97]
[117, 123, 129]
[159, 168, 177]
```

Conclusion:
The Above Program Is Successfully Executed.

9. Write program in Python using Sets to find correct answers for Choice based questions.

Aim: To write program in Python using Sets to find correct answers for Choice based questions.

Description:

This program simulates the scoring of a multiple-choice test by comparing a list of a student's answers against a list of correct answers.

The primary goal is to **calculate the score** (number correct, number wrong, and total marks) by performing a sequential, index-based comparison.

The secondary goal is to use the **Set data structure** to:

1. Store and identify all the **unique correct answer choices** (e.g., all the 'A's, 'B's, 'C's, and 'D's that appeared as correct keys).
2. Store and identify all the **unique wrong choices** the student selected.
3. Illustrate set operations (like **Difference**) to find answer choices that were keys but were **never selected correctly**.

Algorithm:

1. Initialization and Setup

1. **Define Inputs:** Create two ordered lists: correct_answers_list and student_answers_list.
2. **Define Constants:** Set TOTAL_QUESTIONS (length of the lists) and MARKS_PER_CORRECT.
3. **Initialize Sets:** Create two empty sets to store unique choices:
    - correct_choices_set
    - wrong_student_choices_set
4. **Initialize Score Counter:** Set correct_count = 0.

2. Scoring and Set Population (The Loop)

1. **Iterate:** Loop through the indices i from 0 to TOTAL_QUESTIONS - 1.
2. **Compare Answers:**In each iteration, compare student_answers_list[i] with correct_answers_list[i].
3. **If Correct:**
    - Increment correct_count.
    - Use the .add() method to add the correct_answers_list[i] (the key) to the correct_choices_set.
4. **If Incorrect:**
    - Use the .add() method to add the student_answers_list[i] (the student's wrong selection) to the wrong_student_choices_set.

3. Score Calculation

1. Calculate **Number of Wrongs**: num_wrong = TOTAL_QUESTIONS - correct_count.
2. Calculate **Total Marks**: $total_marks = correct_count \times MARKS_PER_CORRECT.
3. Calculate **Maximum Marks**: $max_marks = TOTAL_QUESTIONS times MARKS_PER_CORRECT.
4. **Display** the scoring results.

Source Code:

```
# --- 1. SETUP ---
# The official correct answers (List)
correct_answers_list = [
    'A', 'C', 'B', 'D', 'A',
'C', 'D', 'B', 'A', 'C'
]

# The student's submitted answers (List)
student_answers_list = [
    'A', 'D', 'B', 'D', 'B',
'C', 'C', 'B', 'A', 'D'
]

TOTAL_QUESTIONS = len(correct_answers_list)
MARKS_PER_CORRECT = 2

# To track the correct and wrong *answer choices* for set analysis later
correct_choices_set = set() # Stores unique correct answers (e.g., {'A', 'B', 'C', 'D'})
wrong_student_choices_set = set() # Stores unique *wrong* answers given by the student (e.g., {'D', 'B', 'C'})
correct_count = 0

# --- 2. SCORING LOOP (Uses Lists for ORDERED comparison) ---
for i in range(TOTAL_QUESTIONS):
correct_ans = correct_answers_list[i]
student_ans = student_answers_list[i]

ifstudent_ans == correct_ans:
        # Tally the score
correct_count += 1
        # Add the correct answer CHOICE (e.g., 'A') to a set
correct_choices_set.add(correct_ans)
else:
        # Add the wrong answer CHOICE the student gave to a set
wrong_student_choices_set.add(student_ans)


# --- 3. FINAL CALCULATIONS ---
num_correct = correct_count
num_wrong = TOTAL_QUESTIONS - num_correct
total_marks = num_correct * MARKS_PER_CORRECT
max_marks = TOTAL_QUESTIONS * MARKS_PER_CORRECT

print("## □ Test Score Summary ##")
print(f"* **Total Questions:** {TOTAL_QUESTIONS}")
print(f"* **Maximum Possible Marks:** {max_marks}")
print("-" * 30)
print(f"✅ **Number of Correct Answers:** {num_correct}")
print(f"❌ **Number of Wrong Answers:** {num_wrong}")
print(f"□ **Total Marks Obtained:** {total_marks} / {max_marks}")
print("-" * 30)

# --- 4. SET-BASED ANALYSIS ---
```

```
print("## □ Set Analysis of Answer Choices ##")

# A. Unique Correct Choices
print(f"1. **Unique Correct Answer Choices (Keys to the test):** {correct_choices_set}")
# Example: {'A', 'B', 'C', 'D'}

# B. Unique Wrong Choices by Student
print(f"2. **Unique Wrong Choices Given by Student:** {wrong_student_choices_set}")
# Example: {'C', 'D', 'B'}

# C. Intersection: What the student got right AND wrong (impossible, just for illustration)
# This will be an empty set, confirming the choices are disjoint.
common_choices = correct_choices_set&wrong_student_choices_set
print(f"3. **Intersection (Common to both sets):** {common_choices}")

# D. Difference: What the student *never chose* correctly
# This finds the correct keys they did NOT get right (e.g., if 'D' was a key, but they never selected 'D' correctly)
all_possible_choices = {'A', 'B', 'C', 'D'}
never_chosen_correctly = all_possible_choices - correct_choices_set
print(f"4. **Choices NOT Selected Correctly:** {never_chosen_correctly}")
```

Output:

□ Test Score Summary
* **Total Questions:** 10
* **Maximum Possible Marks:** 20

✓ **Number of Correct Answers:** 6
✗ **Number of Wrong Answers:** 4
□ **Total Marks Obtained:** 12 / 20

□ Set Analysis of Answer Choices
1. **Unique Correct Answer Choices (Keys to the test):** {'A', 'B', 'C', 'D'}
2. **Unique Wrong Choices Given by Student:** {'B', 'C', 'D'}
3. **Intersection (Common to both sets):** {'B', 'C', 'D'}
4. **Choices NOT Selected Correctly:** set()

Conlusion:
The Above Program Is Executed Successfully.

10. Write a Python program using Dictionary to find Train details for a given Train number.

Aim: To Write a Python program using Dictionary to find Train details for a given Train number.
Description: The Python program is designed to look up train details using a **dictionary** structure.

- **Data Structure:** It utilizes a **nested dictionary** named train_database.
  - The **outer dictionary's keys** are the unique **Train Numbers** (integers).
  - The **outer dictionary's values** are **inner dictionaries** that hold the specific details: "Name", "Source", "Destination", and "Platform".
- **Functionality:**
  1. It defines a function, find_train_details(), which takes a train_number as input.
  2. The program prompts the user to **input a train number**. It includes basic error handling (try-except) to ensure the input is a valid integer.
  3. It uses the highly reliable .get() method on the train_database to search for the entered number.
  4. **Success Case:** If the train number is found (i.e., .get() returns details), it iterates through the inner dictionary and prints all the associated details in a clear format.
  5. **Failure Case:** If the train number is **not found** (i.e., .get() returns None), it prints a user-friendly error message indicating that the train is missing from the database.

Algorithm:
Step-by-Step Procedure

1. Program Setup and Definition

Step	Action	Details
1.1	**Define Data Dictionary**	Create and store the **nested dictionary** (train_database) in your Python script. This dictionary holds all the train numbers (keys) and their corresponding details (nested values).
1.2	**Define Function**	Define the function find_train_details(train_number) which contains the core logic for looking up and displaying the information.

2. Getting Input from the User

Step	Action	Details
2.1	**Prompt for Input**	The program displays a message asking the user to **"Enter the Train Number"**.
2.2	**Capture Input**	The program waits for the user to type in a number (e.g., 12723) and presses Enter.
2.3	**Validate and Convert**	The program attempts to convert the captured input into an **integer**. If this fails (e.g., the user types text), the program prompts the user again with an error message until a valid number is entered.

3. Executing the Search

Step	Action	Details
3.1	**Call the Function**	The program calls the find_train_details() function, passing the validated integer (user_train_number) to it.
3.2	**Perform Dictionary Lookup**	Inside the function, the program uses the **.get() method** on the train_database dictionary to search for the input train number.
3.3	**Store Result**	The details (the inner dictionary) are retrieved and stored in a variable (e.g., details). If the train number doesn't exist

Source Code:

```
deffind_train_details(train_number):
    """
    Finds and prints the details of a train given its number using a dictionary.
    """
    # Dictionary to store train details
    # Key: Train Number (int)
    # Value: A dictionary containing Name, Source, and Destination (string)
train_database = {
        12137: {
            "Name": "Mumbai Express",
            "Source": "CSMT, Mumbai",
            "Destination": "HWH, Kolkata",
            "Platform": 5
        },
        12723: {
            "Name": "Telangana Express",
            "Source": "HYB, Hyderabad",
            "Destination": "NDLS, New Delhi",
            "Platform": 3
    },
        12695: {
            "Name": "Chennai Express",
            "Source": "MAS, Chennai",
            "Destination": "TVC, Trivandrum",
            "Platform": 8
        },
        12861: {
            "Name": "Visakha Express",
            "Source": "VSKP, Visakhapatnam",
            "Destination": "SC, Secunderabad",
            "Platform": 1
        }
    }
print("-" * 30)
    # Use .get() method for safe dictionary lookup
details = train_database.get(train_number)
if details:
print(f"**Details for Train No. {train_number}:**")
        # Iterate over the key-value pairs in the found details dictionary
for key, value in details.items():
print(f"  {key}: **{value}**")
else:
print(f"✕ Error: Train number **{train_number}** not found in the database.")
print("Please check the number and try again.")
print("-" * 30)
# --- Main part of the program ---
# 1. Get input from the user
while True:
try:
        # Convert input to an integer, as train numbers are typically numeric
user_train_number = int(input("Enter the Train Number (e.g., 12723): "))
break # Exit the loop if conversion is successful
exceptValueError:
print("Invalid input. Please enter a numerical train number.")
# 2. Call the function to find and display the details
find_train_details(user_train_number)
```

Input and Output:

```
Enter the Train Number (e.g., 12723):  12695
-----------------------------
**Details for Train No. 12695:**
  Name: **Chennai Express**
  Source: **MAS, Chennai**
  Destination: **TVC, Trivandrum**
  Platform: **8**
-----------------------------

Enter the Train Number (e.g., 12723):  9099
-----------------------------
✕ Error: Train number **9099** not found in the database.
Please check the number and try again.
-----------------------------

Enter the Train Number (e.g., 12723):  12861
-----------------------------
**Details for Train No. 12861:**
  Name: **Visakha Express**
  Source: **VSKP, Visakhapatnam**
  Destination: **SC, Secunderabad**
  Platform: **1**
-----------------------------
```

Conclusion :
The Above Program Is Successfully Executed.

11. Write a Python Program for Pass by Value and Pass by Object (reference).

Aim: To Write a Python Program for Pass by Value and Pass by Object (reference).

Description:

This program demonstrates three core concepts of how functions interact with data in Python:
Pass by Object Reference: The two final functions, change_number() and append_to_list(), illustrate Python's unique method of passing arguments.

- **Immutable Objects (like integers):** When change_number() tries to modify the number, a **new** integer object is created locally, leaving the original variable (my_number) unchanged outside the function.
- **Mutable Objects (like lists):** When append_to_list() modifies the list using methods like .append(), it modifies the original object **in place** because the function and the calling code both share the exact same reference to the list object in memory, resulting in a permanent change to data_list.

Algorithm:

Pass by Object Reference (Immutable)

Step	Action	Details
2.1	Define change_number(num)	Function takes an integer (num) and attempts to increment it by 1.
2.2	Initialize my_number	Set my_number = 10 and print its initial value and memory ID.
2.3	**Call Function**	Call change_number(my_number). Inside the function, num = num + 1 creates a **new** integer object in memory (new ID).
2.4	**Check Result**	Print my_number again. It remains 10 because the change was local to the function's scope.

Pass by Object Reference (Mutable)

Step	Action	Details
3.1	Define append_to_list(my_list)	Function takes a list (my_list) and appends the value 100 to it.
3.2	Initialize data_list	Set data_list = [1, 2, 3] and print its initial state and memory ID.
3.3	**Call Function**	Call append_to_list(data_list). Inside the function, the .append(100) method modifies the **original object** at the same memory location (same ID).
3.4	**Check Result**	Print data_list again. It is now [1, 2, 3, 100] because the function changed the mutable object referenced by the variable.

Source Code:

```
# --- 1. Immutable Object Example (Integer) ---
defchange_number(num):
print(f"Inside function (start): num={num}, ID={id(num)}")
    # When you perform an operation like `num = num + 1`,
    # Python creates a *new* integer object in memory and updates `num` to reference it.
num = num + 1
print(f"Inside function (end): num={num}, ID={id(num)}")
my_number = 10
print(f"Outside function (before): my_number={my_number}, ID={id(my_number)}")
change_number(my_number)
print(f"Outside function (after): my_number={my_number}")
# RESULT: my_number remains 10. The function only changed its *local* copy of the reference.
# --- 2. Mutable Object Example (List) ---
defappend_to_list(my_list):
print(f"\nInside function (start): my_list={my_list}, ID={id(my_list)}")
    # The .append() method modifies the object *in place* using the existing reference.
my_list.append(100)
print(f"Inside function (end): my_list={my_list}, ID={id(my_list)}")

data_list = [1, 2, 3]
print(f"Outside function (before): data_list={data_list}, ID={id(data_list)}")
append_to_list(data_list)
print(f"Outside function (after): data_list={data_list}")
# RESULT: data_list is changed to [1, 2, 3, 100]. The function modified the original object
# because it was mutable.
```

Input and Output:

Outside function (before): my_number=10, ID=140733226826456
Inside function (start): num=10, ID=140733226826456
Inside function (end): num=11, ID=140733226826488
Outside function (after): my_number=10
Outside function (before): data_list=[1, 2, 3], ID=1988836906112

Inside function (start): my_list=[1, 2, 3], ID=1988836906112
Inside function (end): my_list=[1, 2, 3, 100], ID=1988836906112
Outside function (after): data_list=[1, 2, 3, 100]

Conclusion:
The Above Program Is Executed Successfully.

12. **Write a Python program for Employee Class and pay calculations.**

Aim: To Write a Python program for Employee Class and pay calculations

Description:

The program implements a basic payroll system using **Object-Oriented Programming (OOP)** concepts in Python.

- **Employee Class:** This class serves as a blueprint for all employee objects. It encapsulates both the **data** (attributes) and the **behavior** (methods/functions) related to an employee.
    - **Class Variables:** DA_RATE, HRA_RATE, and PF_RATE are defined as constants at the class level, making them easy to adjust globally.
    - **__init__ Method (Constructor):** This method runs when a new Employee object is created. It takes the required inputs (empno, empname, dept, basic) and assigns them to the object's instance variables.
    - **calculate_payroll Method:** This is the core logic. It uses the defined rates to calculate Dearness Allowance (DA), House Rent Allowance (HRA), Provident Fund (PF), **Gross Pay** (earnings), and finally, **Net Pay** (Gross Pay minus PF).
    - **display_details Method:** This prints a formatted summary of the employee's personal information and the calculated payroll components.
- **Main Execution Block:**This block handles user input, creates an Employee object, calls the calculate_payroll method to compute the results, and then calls display_details to present the final output.

Algorithm:

The algorithm outlines the procedural steps taken by the program to define the class and execute the payroll calculation for a single employee.

1. Class Definition (Setup)

Step	Action	Details
1.1	**Define Class**	Start the Employee class definition.
1.2	**Define Rates**	Set fixed class constants: DA_RATE (0.15), HRA_RATE (0.10), PF_RATE (0.08).
1.3	**Define Constructor**	Define __init__(self, empno, empname, dept, basic) to initialize input attributes.
1.4	**Define calculate_payroll**	Define the method for computation (see Step 3).
1.5	**Define display_details**	Define the method for formatted output.

2. Main Execution and Input

Step	Action	Details
2.1	**Gather Input**	Prompt the user to enter empno, empname, dept, and basic salary.
2.2	**Validate Input**	Convert basic salary to a float, handling any ValueError.
2.3	**Instantiate Object**	Create a new instance of the Employee class, passing the gathered inputs to the constructor.

3. Calculation (calculate_payroll Method)

Step	Formula	Result
3.1	**Calculate DA**	$DA = Basic \times DA\_RATE$
3.2	**Calculate HRA**	$HRA = Basic \times HRA\_RATE$
3.3	**Calculate PF**	$PF = Basic \times PF\_RATE$ (This is a deduction)
3.4	**Calculate Gross Pay**	$Gross Pay = Basic + DA + HRA$
3.5	**Calculate Net Pay**	$Net Pay = Gross Pay - PF$

4. Output

Step	Action	Details
4.1	**Call Display**	Call the employee.display_details() method.
4.2	**Print Data**	Display the employee's personal data (empno, empname, dept).
4.3	**Print Results**	Display the calculated values: DA, HRA, Gross Pay, PF, and the final **Net Pay**.

Source Code:

```
class Employee:
    """
    Represents an employee and performs DA, HRA, and Net Pay calculations.
    """
    # Define fixed rates for calculations (can be changed here)
    DA_RATE = 0.15  # 15% of Basic
    HRA_RATE = 0.10 # 10% of Basic
    PF_RATE = 0.08  # 8% deduction for Provident Fund

def __init__(self, empno, empname, dept, basic):
        """
        Constructor to initialize employee attributes.
        """
self.empno = empno
self.empname = empname
self.dept = dept
self.basic = basic

        # Initialize calculated fields to None (they will be calculated later)
self.da = None
        self.hra = None
self.pf = None
self.gross_pay = None
self.net_pay = None

defcalculate_payroll(self):
        """
        Calculates Dearness Allowance (DA), House Rent Allowance (HRA),
Provident Fund (PF), Gross Pay, and Net Pay.
        """
# 1. Calculate Allowances/Deductions
self.da = self.basic * Employee.DA_RATE
self.hra = self.basic * Employee.HRA_RATE
        self.pf = self.basic * Employee.PF_RATE # Deduction

# 2. Calculate Gross Pay (Total Earnings)
        # Gross Pay = Basic + DA + HRA
self.gross_pay = self.basic + self.da + self.hra

# 3. Calculate Net Pay (Earnings - Deductions)
        # Net Pay = Gross Pay - PF
self.net_pay = self.gross_pay - self.pf

defdisplay_details(self):
        """
        Prints all employee details and calculated payroll components.
```

```
        """
print("\n" + "=" * 40)
print("□ Employee Payroll Details")
print("=" * 40)
print(f"Employee No: {self.empno}")
print(f"Name: {self.empname}")
print(f"Department: {self.dept}")
print("-" * 40)

        # Display Inputs
print(f"Basic Salary (Input): ₹{self.basic:,.2f}")
print("-" * 40)

        # Display Calculations
print(f"DA (15%):          ₹{self.da:,.2f}")
print(f"HRA (10%):         ₹{self.hra:,.2f}")
print(f"Gross Pay:         ₹{self.gross_pay:,.2f}")
print(f"PF Deduction (8%): -₹{self.pf:,.2f}")
print("-" * 40)
print(f"**Net Pay:** **₹{self.net_pay:,.2f}**")
print("=" * 40)

# --- Main Execution ---
# 1. Gather Input
print("--- Enter Employee Information ---")
try:
emp_no = input("Enter Employee Number (e.g., E101): ")
emp_name = input("Enter Employee Name: ")
emp_dept = input("Enter Department: ")
basic_salary = float(input("Enter Basic Salary (e.g., 50000): "))
exceptValueError:
print("Error: Basic Salary must be a valid number.")
exit()


# 2. Create Object
employee1 = Employee(emp_no, emp_name, emp_dept, basic_salary)

# 3. Perform Calculations
employee1.calculate_payroll()

# 4. Display Output
employee1.display_details()
```

Input and Output:

--- Enter Employee Information ---
Enter Employee Number (e.g., E101): E1901
Enter Employee Name: K MAHESH
Enter Department: ACCOUNTS
Enter Basic Salary (e.g., 50000): 24000

==========
 Employee Payroll Details
==========
Employee No: E1901
Name: K MAHESH
Department: ACCOUNTS

Basic Salary (Input): ₹24,000.00

DA (15%): ₹3,600.00
HRA (10%): ₹2,400.00
Gross Pay: ₹30,000.00
PF Deduction (8%): -₹1,920.00

Net Pay: **₹28,080.00**
==========

Conclusion:

The Above Program Executed Successfully.

13. **Write a Python program for Bank Applications using functions.**

Aim: To Write a Python program for Bank Applications using functions.

Description:

The program is structured around a main function (run_bank_app) that presents a persistent menu to the user, allowing them to interact with the simulated bank account through several dedicated functions.

1. Data Management (The Global Dictionary)

- **current_account = {}:** An empty dictionary is initialized globally. This dictionary acts as the simple database for the single active bank account.
- The dictionary stores four key-value pairs upon account creation: 'accno', 'name', 'account_type', and 'balance'.
- The use of the global keyword within create_account and the modification of dictionary contents in deposit and withdraw allow all functions to manipulate this single, shared account data.

2. Core Functions

- **create_account():**
  - This function is responsible for gathering all necessary customer details (Account Number, Name, Type, and Initial Deposit) from the user.
  - It includes robust **input validation** using while loops and try-except blocks to ensure the Account Number is a whole number, the Account Type is either "Savings" or "Current," and the Initial Balance is a non-negative number.
  - It populates the current_account dictionary and automatically calls check_balance() upon success.
- **check_balance():**
  - This function retrieves and neatly prints the current account holder's name, account type, and balance from the current_account dictionary.
  - It includes a check to ensure an account has actually been created before attempting to access details.
- **deposit(amount):**
  - Accepts a positive amount.
  - Adds the amount to the 'balance' key in the current_account dictionary.
  - Prints a success message and updates the balance via check_balance().
- **withdraw(amount):**
  - Accepts a positive amount.
  - Includes a critical check for **insufficient funds**: if the withdrawal amount exceeds the current balance, the transaction is rejected.
  - If funds are sufficient, the amount is subtracted from the 'balance' key.
  - Prints success or error messages accordingly.

3. Main Application Loop (run_bank_app)

- The run_bank_app function contains the main execution loop, which continually displays the **Main Menu** (Options 1-5).
- It takes the user's choice and calls the appropriate function (create_account, check_balance, deposit, or withdraw).
- The loop continues until the user selects **Option 5 (Exit)**.
- The functions for deposit and withdrawal also include input validation within the main loop to handle non-numeric inputs for the transaction amount.

Source Code:

```
# Initialize the account details storage as an empty dictionary
current_account = {}

defcreate_account():
    """
    Prompts the user for account details and updates the global current_account.
    """
globalcurrent_account
print("\n---  Account Creation ---")

# 1. Account Number (Input Validation)
while True:
try:
accno = int(input("Enter Account Number (e.g., 12345): "))
break
exceptValueError:
print("✕ Invalid input. Please enter a valid whole number for the Account Number.")

# 2. Account Holder Name
name = input("Enter Account Holder Name: ")

# 3. Account Type (Input Validation)
while True:
account_type = input("Enter Account Type (Savings or Current): ").strip().title()
ifaccount_type in ["Savings", "Current"]:
break
else:
print("✕ Invalid account type. Please enter 'Savings' or 'Current'.")

# 4. Initial Balance (Input Validation)
while True:
try:
balance = float(input("Enter Initial Deposit Amount (must be $0 or more): $"))
if balance >= 0:
break
else:
print("✕ Initial balance cannot be negative. Please enter $0 or more.")
exceptValueError:
print("✕ Invalid input. Please enter a valid number for the balance.")

    # Store all details in the global dictionary
current_account['accno'] = accno
current_account['name'] = name
current_account['account_type'] = account_type
current_account['balance'] = balance
```

```
print("\n✓ Account successfully created!")
check_balance()

# --- Transaction Functions (Modified to use the dictionary) ---
defcheck_balance():
"""Prints the current account balance and details."""
if not current_account:
print("\n⚠ Please create an account first (Option 1).")
return

print("\n Account Details:")
print(f"   Account Holder: {current_account['name']}")
print(f"   Account Type: {current_account['account_type']}")
print(f"   Balance: ${current_account['balance']:.2f}")

def deposit(amount):
"""Deposits the specified amount into the account."""
if not current_account:
print("\n⚠ Please create an account first (Option 1).")
return

if amount > 0:
current_account['balance'] += amount
print(f"\n✓ Deposit successful.")
print(f"   ${amount:.2f} has been added to the account.")
check_balance()
else:
print("\n✗ Error: Deposit amount must be a positive number.")

def withdraw(amount):
"""Withdraws the specified amount from the account, if sufficient funds are available."""
if not current_account:
print("\n⚠ Please create an account first (Option 1).")
return

current_balance = current_account['balance']

if amount <= 0:
print("\n✗ Error: Withdrawal amount must be a positive number.")
elif amount >current_balance:
print("\n✗ Error: Insufficient funds.")
print(f"   You tried to withdraw ${amount:.2f}, but your balance is only ${current_balance:.2f}.")
else:
current_account['balance'] -= amount
print(f"\n✓ Withdrawal successful.")
```

```
print(f"   ${amount:.2f} has been withdrawn from the account.")
check_balance()
# --- Main Application Loop ---
defrun_bank_app():
"""Main function to run the menu-driven application."""
print("----------------------------------------")
print("□ Welcome to the Simple Python Bank App")
print("----------------------------------------")

while True:
print("\n### Main Menu ###")
print("1. Create New Account")
print("2. Check Balance")
print("3. Deposit Amount")
print("4. Withdraw Amount")
print("5. Exit")

choice = input("Enter your choice (1-5): ")

if choice == '1':
create_account()

elif choice == '2':
check_balance()

elif choice == '3':
        # Get deposit amount from user
ifcurrent_account:
while True:
try:
amt = float(input("Enter amount to deposit: $"))
deposit(amt)
break
exceptValueError:
print("✕ Invalid input. Please enter a valid number.")
else:
print("\n⚠ Please create an account first (Option 1).")

elif choice == '4':
        # Get withdrawal amount from user
ifcurrent_account:
while True:
try:
amt = float(input("Enter amount to withdraw: $"))
withdraw(amt)
break
exceptValueError:
```

```
print("❌ Invalid input. Please enter a valid number.")
else:
print("\n⚠ Please create an account first (Option 1).")

elif choice == '5':
print("\n Thank you for using the Simple Python Bank App. Goodbye!")
break
else:
print("\n❌ Invalid choice. Please enter a number between 1 and 5.")
# Run the application
if __name__ == "__main__":
run_bank_app()
```

Input and output:

```
---------------------------------------
 Welcome to the Simple Python Bank App
---------------------------------------

### Main Menu ###
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5):  1

---  Account Creation ---
Enter Account Number (e.g., 12345):  190911
Enter Account Holder Name:  Ravi kumar
Enter Account Type (Savings or Current):  Savings
Enter Initial Deposit Amount (must be $0 or more): $ 9000

✅ Account successfully created!

 Account Details:
   Account Holder: Ravi kumar
   Account Type: Savings
   Balance: $9000.00

### Main Menu ###
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5):  2

 Account Details:
   Account Holder: Ravi kumar
   Account Type: Savings
   Balance: $9000.00

### Main Menu ###
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5):  3
Enter amount to deposit: $ 2000

✅ Deposit successful.
   $2000.00 has been added to the account.

 Account Details:
```

Account Holder: Ravi kumar
Account Type: Savings
Balance: $11000.00

Main Menu
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5): 2

 Account Details:
Account Holder: Ravi kumar
Account Type: Savings
Balance: $11000.00

Main Menu
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5): 4
Enter amount to withdraw: $ 1000

✅ Withdrawal successful.
$1000.00 has been withdrawn from the account.

 Account Details:
Account Holder: Ravi kumar
Account Type: Savings
Balance: $10000.00

Main Menu
1. Create New Account
2. Check Balance
3. Deposit Amount
4. Withdraw Amount
5. Exit
Enter your choice (1-5): 2

 Account Details:
Account Holder: Ravi kumar
Account Type: Savings
Balance: $10000.00

Conclusion:
The Above Program Is Successfully Executed.

14. Write a Python program to implement Stack Operations.

Aim: To Write a Python program to implement Stack Operations.

Description:

A **Stack** is an abstract data type (ADT) that follows the **Last-In, First-Out (LIFO)** principle. Think of a stack of dinner plates: the last plate added is the first one removed.

The program implements the stack using a Python **list**, where the end of the list is treated as the **"top"** of the stack.

Operation	Description	Python Implementation
push(item)	Adds a new item to the top of the stack.	Uses list.append(item).
pop()	Removes and returns the item from the top of the stack.	Uses list.pop().
peek()	Returns the top item without removing it.	Accesses the last element using list[-1].
is_empty()	Checks if the stack contains any elements.	Checks if len(list) == 0.
is_full()	(Optional) Checks if the stack has reached its capacity.	Checks if len(list) >= MAX_SIZE.
Stack Overflow	Occurs when attempting to push an item onto a full stack.	Handled by is_full().
Stack Underflow	Occurs when attempting to pop or peek from an empty stack.	Handled by is_empty().

Algorithm:

The algorithm defines the logical steps for each core stack operation.

1. Push Operation

1. **START**push(item)
2. **CHECK** if the stack is full (if len(stack) == MAX_SIZE).
3. **IF** full: Print "Stack Overflow" error.
4. **ELSE**: Append the item to the end of the list (stack.append(item)).
5. **END**

2. Pop Operation

1. **START**pop()
2. **CHECK** if the stack is empty (if len(stack) == 0).
3. **IF** empty: Print "Stack Underflow" error and return None.
4. **ELSE**: a. Remove the last element from the list (item = stack.pop()). b. Return the removed item.
5. **END**

3. Peek Operation

1. **START**peek()
2. **CHECK** if the stack is empty (if len(stack) == 0).
3. **IF** empty: Print a warning and return None.
4. **ELSE**: a. Access the last element of the list (top_item = stack[-1]). b. Print the top_item. c. Return the top_item.
5. **END**

4. IsEmpty Operation

1. **START**is_empty()
2. **CHECK** if len(stack) == 0.
3. **RETURN**True if the length is zero, otherwise return False.
4. **END**

Source Code :

```
# The Stack is represented by a Python list
stack = []
MAX_SIZE = 5

defis_empty():
"""Checks if the stack is empty."""
returnlen(stack) == 0

defis_full():
"""Checks if the stack has reached its maximum size."""
returnlen(stack) >= MAX_SIZE

def push(item):
"""Adds an item to the top of the stack."""
ifis_full():
print(f"✗ Error: Stack Overflow! Cannot push. Max size is {MAX_SIZE}.")
else:
stack.append(item)
print(f"✓ Pushed '{item}' onto the stack.")

def pop():
    """Removes and returns the item from the top of the stack (LIFO)."""
ifis_empty():
print("✗ Error: Stack Underflow! Cannot pop from an empty stack.")
else:
item = stack.pop()
print(f"✓ Popped '{item}' from the stack.")

def peek():
"""Returns the item at the top of the stack without removing it."""
ifis_empty():
print("⚠ Stack is empty. Nothing to peek.")
else:
top_item = stack[-1]
print(f"□ Peek: Top item is '{top_item}'.")

defdisplay_stack():
"""Displays the current state of the stack."""
ifis_empty():
print("\nStack: [] (Empty)")
else:
      # Displaying the stack vertically for better visualization (Top is first)
print("\n--- Current Stack (TOP) ---")
for item in reversed(stack):
print(f"| {item} |")
print("-----------")
```

```
defrun_stack_app():
"""Main function to run the menu-driven stack application."""
print("\n----------------------------------")
print("□ Welcome to the Stack Simulator")
print("----------------------------------")

while True:
display_stack()
print("\n### Stack Menu ###")
print("1. Push (Add item)")
print("2. Pop (Remove item)")
print("3. Peek (View top item)")
print("4. Exit")

choice = input("Enter your choice (1-4): ")

if choice == '1':
item_to_push = input("Enter item to push: ")
push(item_to_push)

elif choice == '2':
pop()

elif choice == '3':
peek()

elif choice == '4':
print("\n□ Exiting Stack Simulator. Goodbye!")
break

else:
print("\n✗ Invalid choice. Please enter a number between 1 and 4.")

# Run the application
if __name__ == "__main__":
run_stack_app()
# --- Demonstration ---

print("--- Stack Operations Demo ---")

# 1. Push Operations
push(10)
push('Apple')
push(3.14)
print(f"Current Stack: {stack}")
peek()
```

```
# 2. Pop Operations
popped_item = pop()
popped_item = pop()
print(f"Current Stack: {stack}")

# 3. Boundary Condition Check (Empty)
peek()
pop() # Pop the last item (10)
pop() # Attempt to pop from an empty stack (Underflow)

# 4. Boundary Condition Check (Full - if MAX_SIZE is defined)
print("\n--- Testing MAX_SIZE (5) ---")
stack = [1, 2, 3, 4, 5]
push(6) # Should trigger Stack Overflow
print(f"Final Stack: {stack}")
```

Input and Output:

```
---------------------------------
 Welcome to the Stack Simulator
---------------------------------

Stack: [] (Empty)

### Stack Menu ###
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4):  1
Enter item to push:  34
✓ Pushed '34' onto the stack.

--- Current Stack (TOP) ---
| 34 |
-----------

### Stack Menu ###
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4):  1
Enter item to push:  56
✓ Pushed '56' onto the stack.

--- Current Stack (TOP) ---
| 56 |
| 34 |
-----------

### Stack Menu ###
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4):  1
Enter item to push:  78
✓ Pushed '78' onto the stack.

--- Current Stack (TOP) ---
| 78 |
| 56 |
| 34 |
-----------

### Stack Menu ###
1. Push (Add item)
2. Pop (Remove item)
```

3. Peek (View top item)
4. Exit
Enter your choice (1-4): 1
Enter item to push: 88
✓ Pushed '88' onto the stack.

--- Current Stack (TOP) ---
| 88 |
| 78 |
| 56 |
| 34 |

Stack Menu
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4): 1
Enter item to push: 70
✓ Pushed '70' onto the stack.

--- Current Stack (TOP) ---
| 70 |
| 88 |
| 78 |
| 56 |
| 34 |

Stack Menu
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4): 2
✓ Popped '70' from the stack.

--- Current Stack (TOP) ---
| 88 |
| 78 |
| 56 |
| 34 |

Stack Menu
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4): 3
 Peek: Top item is '88'.

--- Current Stack (TOP) ---

```
| 88 |
| 78 |
| 56 |
| 34 |
-----------

### Stack Menu ###
1. Push (Add item)
2. Pop (Remove item)
3. Peek (View top item)
4. Exit
Enter your choice (1-4):  4
```

Conclusion:

The Above Program Is Successfully Executed.

15. Write Python Program for Queue Operations.

Aim: To Write Python Program for Queue Operations.

Description:

A **Queue** is an abstract data type (ADT) that follows the **First-In, First-Out (FIFO)** principle. Think of a line at a ticket counter or items waiting in a print spooler: the first one in is the first one out.

The program implements the queue using Python's **collections.deque**.

Operation	Description	Python Implementation
enqueue(item)	Adds a new item to the **rear** (end) of the queue.	Uses deque.append(item).
dequeue()	Removes and returns the item from the **front** (beginning) of the queue.	Uses deque.popleft().
front()	Returns the item at the front without removing it.	Accesses the first element using deque[0].
is_empty()	Checks if the queue contains any elements.	Checks if len(deque) == 0.
Queue Full	Occurs when attempting to enqueue an item onto a full queue.	Handled by is_full().
Queue Empty	Occurs when attempting to dequeue or peek from an empty queue.	

Queue Implementation Algorithm

The algorithm defines the logical steps for each core queue operation, focusing on the FIFO principle.

1. Enqueue Operation (Adding to the Rear)

1. **START**enqueue(item)
2. **CHECK** if the queue is full (if len(queue) == MAX_SIZE).
3. **IF** full: Print "Queue Full" error.
4. **ELSE**: Add the item to the **right side** of the deque (queue.append(item)).
5. **END**

2. Dequeue Operation (Removing from the Front)

1. **START**dequeue()
2. **CHECK** if the queue is empty (if len(queue) == 0).
3. **IF** empty: Print "Queue Empty" error and return None.
4. **ELSE**: a. Remove the element from the **left side** of the deque (item = queue.popleft()). b. Return the removed item.
5. **END**

3. Front Operation

1. **START**front()
2. **CHECK** if the queue is empty (if len(queue) == 0).

3. **IF** empty: Print a warning and return None.
4. **ELSE**: a. Access the **first element** of the deque (front_item = queue[0]). b. Print the front_item. c. Return the front_item.
5. **END**

4. IsEmpty Operation

1. **START**is_empty()
2. **CHECK** if len(queue) == 0.
3. **RETURN**True if the length is zero, otherwise return False.
4. **END**

Source Code:

```
from collections import deque

# The Queue is represented by a deque object
queue = deque()
MAX_SIZE = 5

defis_empty():
"""Checks if the queue is empty."""
returnlen(queue) == 0

defis_full():
"""Checks if the queue has reached its maximum size."""
returnlen(queue) >= MAX_SIZE

defenqueue(item):
"""Adds an item to the rear (end) of the queue."""
ifis_full():
print(f"✗ Error: Queue Full! Cannot enqueue. Max size is {MAX_SIZE}.")
else:
    # appends to the right end (the "rear")
queue.append(item)
print(f"✓Enqueued '{item}' into the queue.")

defdequeue():
  """Removes and returns the item from the front (beginning) of the queue (FIFO)."""
ifis_empty():
print("✗ Error: Queue Empty! Cannot dequeue from an empty queue.")
else:
    # popleft() removes the left-most element, which is the 'front'
item = queue.popleft()
print(f"✓Dequeued '{item}' from the queue.")

def front():
"""Returns the item at the front of the queue without removing it."""
ifis_empty():
print("⚠ Queue is empty. Nothing at the front.")
else:
front_item = queue[0]
print(f"□ Front: Item at the front is '{front_item}'.")

defdisplay_queue():
"""Displays the current state of the queue."""
ifis_empty():
print("\nQueue: [] (Empty)")
else:
    # Displaying the queue horizontally with clear indicators
queue_list = list(queue)
print("\n--- Current Queue ---")
print(f"FRONT -> {queue_list} <- REAR")
print("-" * (len(str(queue_list)) + 12))

defrun_queue_app():
```

```
"""Main function to run the menu-driven queue application."""
print("\n---------------------------------")
print("□ Welcome to the Queue Simulator")
print("---------------------------------")

while True:
display_queue()
print("\n### Queue Menu (FIFO) ###")
print("1. Enqueue (Add to Rear)")
print("2. Dequeue (Remove from Front)")
print("3. Front (View Front item)")
print("4. Exit")

choice = input("Enter your choice (1-4): ")

if choice == '1':
item_to_enqueue = input("Enter item to enqueue: ")
enqueue(item_to_enqueue)

elif choice == '2':
dequeue()

elif choice == '3':
front()

elif choice == '4':
print("\n□ Exiting Queue Simulator. Goodbye!")
break

else:
print("\n✕ Invalid choice. Please enter a number between 1 and 4.")

# Run the application
if __name__ == "__main__":
run_queue_app()
```

Input and Output:

```
---------------------------------
 Welcome to the Queue Simulator
---------------------------------

Queue: [] (Empty)

### Queue Menu (FIFO) ###
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4):  1
Enter item to enqueue:  anil
✅Enqueued 'anil' into the queue.

--- Current Queue ---
FRONT -> ['anil'] <- REAR
--------------------

### Queue Menu (FIFO) ###
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4):  1
Enter item to enqueue:  kamal
✅Enqueued 'kamal' into the queue.

--- Current Queue ---
FRONT -> ['anil', 'kamal'] <- REAR
-----------------------------

### Queue Menu (FIFO) ###
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4):  1
Enter item to enqueue:  naveen
✅Enqueued 'naveen' into the queue.

--- Current Queue ---
FRONT -> ['anil', 'kamal', 'naveen'] <- REAR
-------------------------------------

### Queue Menu (FIFO) ###
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4):  3
```

 Front: Item at the front is 'anil'.

--- Current Queue ---
FRONT -> ['anil', 'kamal', 'naveen'] <- REAR

Queue Menu (FIFO)
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4): 1
Enter item to enqueue: mahendra
✅Enqueued 'mahendra' into the queue.

--- Current Queue ---
FRONT -> ['anil', 'kamal', 'naveen', 'mahendra'] <- REAR

Queue Menu (FIFO)
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4): 2
✅Dequeued 'anil' from the queue.

--- Current Queue ---
FRONT -> ['kamal', 'naveen', 'mahendra'] <- REAR

Queue Menu (FIFO)
1. Enqueue (Add to Rear)
2. Dequeue (Remove from Front)
3. Front (View Front item)
4. Exit
Enter your choice (1-4): 4

Conclusion:

The Above Program Is Successfully Executed.