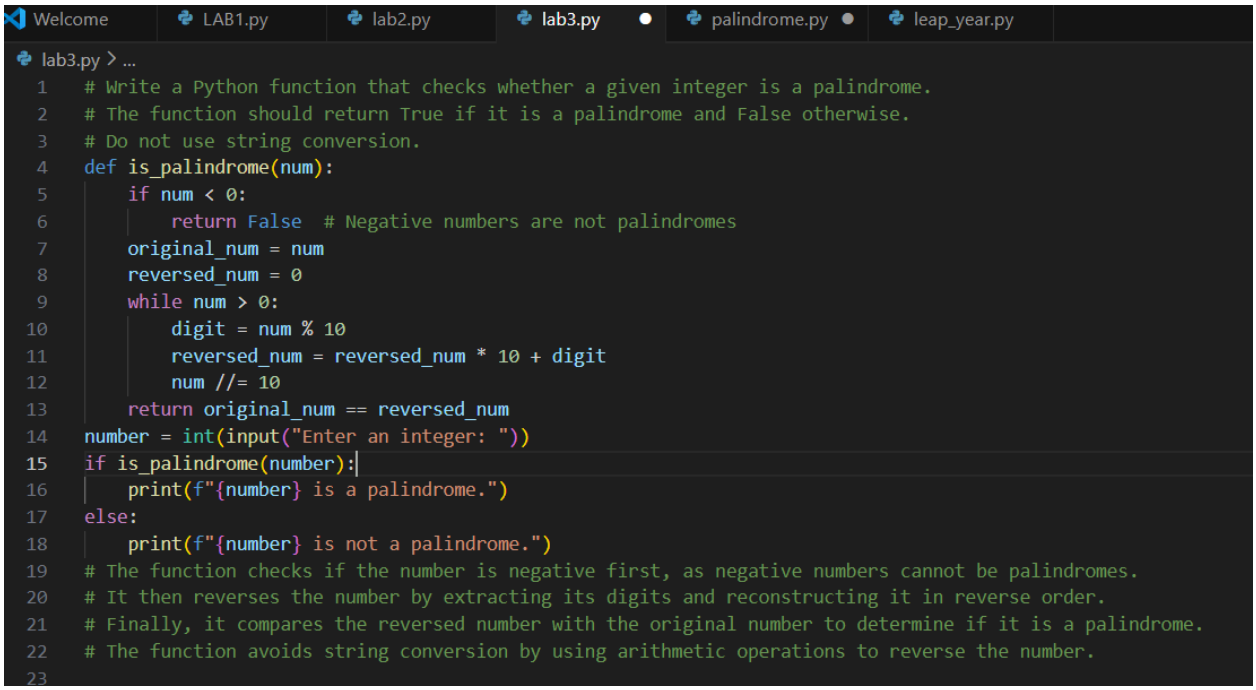
Assignment-3.1

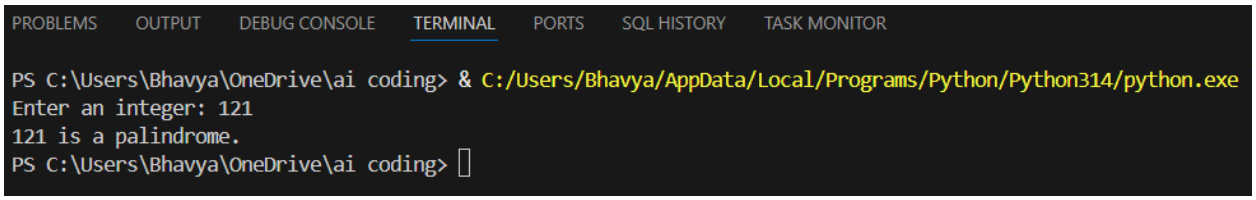
Name-K.Bhavya sri
H.no-2303a51863
Batch-13

Question 1: Zero-Shot Prompting (Palindrome Number Program)
Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is a palindrome.
Task:
• Record the AI-generated code.
• Test the code with multiple inputs.
• Identify any logical errors or missing edge-case handling.

Prompt Strategy:
A zero-shot prompt was used by providing only a high-level instruction to generate a Python function for checking whether a number is a palindrome, without giving any examples.



Output:



Observations:

- The AI correctly inferred the logic to reverse the number and compare it with the original.
- Negative numbers were rejected as non-palindromes.
- The solution worked correctly for single-digit numbers and zero.

Missing Conditions and Edge Cases:

- No input type validation was included; non-integer inputs would cause runtime errors.
- The function assumes valid integer input.

Conclusion:

Zero-shot prompting produced a correct but basic solution. The lack of examples resulted in minimal validation and no explicit handling of invalid data types. This shows that zero-shot prompting often yields functional but non-robust code.

Question 2: One-Shot Prompting (Factorial Calculation)

Write a one-shot prompt by providing one input-output example and ask the AI to generate a Python function to compute the factorial of a given number.

Example:

Input: 5 → Output: 120

Task:

• Compare the generated code with a zero-shot solution.

• Examine improvements in clarity and correctness.

Prompt Strategy:

A one-shot prompt was used by providing one input-output example: Input: 5 → Output: 120. This guided the AI to infer the factorial behavior more precisely.

lab3.py > ...

```python

## Write a Python function to compute the factorial of a number.
# Example:
# Input: 5
# Output: 120
#Zero-shot code generation
def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers."
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
number = int(input("Enter a non-negative integer: "))
print(f"The factorial of {number} is {factorial(number)}.")
# The function checks if the input number is negative and returns an appropriate message.
# It then calculates the factorial iteratively by multiplying all integers from 2 to n.
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
Enter a non-negative integer: 5
The factorial of 5 is 120.
PS C:\Users\Bhavya\OneDrive\ai coding>
```

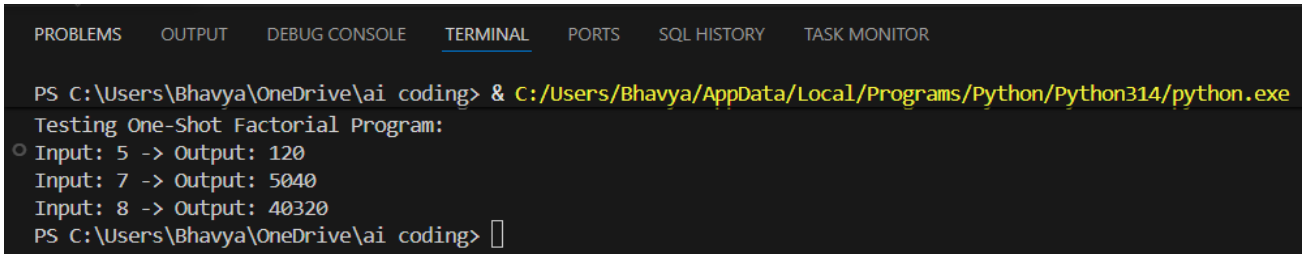
One-Shot Programming:

lab3.py > ...

```python

## Write a Python function to compute the factorial of a number.
# Example:
# Input: 5
# Output: 120
#Zero-shot code generation
def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers."
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
# Test cases
test_cases = [5, 7,8]
print("Testing One-Shot Factorial Program:")
for value in test_cases:
    try:
        print(f"Input: {value} -> Output: {factorial(value)}")
    except Exception as e:
        print(f"Input: {value} -> Error: {e}")

```



Comparison with zero-shot output:

- Zero-shot factorial solutions typically lacked validation for negative numbers.
- One-shot prompting led to:
- Explicit check for negative inputs.
- Clear error handling using exceptions.
- Improved function structure and readability.

Improvements in Clarity and Correctness:

- Better variable naming and loop bounds.
- Correct handling of 0! = 1.
- Proper rejection of invalid negative inputs.

Conclusion:

Providing a single example significantly improved correctness and robustness. One-shot prompting helps the AI generalize expected behavior and include essential edge-case handling.

Question 3: Few-Shot Prompting (Armstrong Number Check)

Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python function to check whether a given number is an Armstrong number.

Examples:

• Input: 153 → Output: Armstrong Number
• Input: 370 → Output: Armstrong Number
• Input: 123 → Output: Not an Armstrong Number

Task:

• Analyze how multiple examples influence code structure and accuracy.
• Test the function with boundary values and invalid inputs.

Prompt Strategy:

Few-shot prompting was used by providing multiple examples of Armstrong and non-Armstrong numbers.

lab3.py > ...

```python
## Write a Python function to check whether a given number is an Armstrong number.
# Examples:
# Input: 153 -> Output: Armstrong Number
# Input: 370 -> Output: Armstrong Number
# Input: 123 -> Output: Not an Armstrong Number
def is_armstrong(num: int) -> bool:
    if num < 0:
        return False

    digits = [int(d) for d in str(num)]
    power = len(digits)

    total = 0
    for d in digits:
        total += d ** power

    return total == num
# Test cases
test_cases = [153, 370, 123]
print("Testing Few-Shot Armstrong Program:\n")

for value in test_cases:
    try:
        result = is_armstrong(value)
        if result:
            print(f"Input: {value} -> Armstrong Number")
        else:
            print(f"Input: {value} -> Not an Armstrong Number")
    except Exception as e:
        print(f"Input: {value} -> Error: {e}")

```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
Testing Few-Shot Armstrong Program:
Input: 153 -> Armstrong Number
Input: 370 -> Armstrong Number
Input: 123 -> Not an Armstrong Number
PS C:\Users\Bhavya\OneDrive\ai coding>
```

Influence of Multiple Examples:

- The AI correctly inferred the Armstrong number definition:
- Count digits.
- Raise each digit to the power of the number of digits.
- Compare the sum with the original number.

Boundary and Invalid Input Testing:

- Single-digit numbers were correctly classified as Armstrong numbers.
- Zero was correctly handled.
- Negative numbers were rejected.
- No validation for non-integer inputs was included.

Conclusion:

Few-shot prompting significantly improved the structure and accuracy of the algorithm. Multiple examples helped the AI generalize the correct mathematical definition and implement a reliable solution.

Question 4: Context-Managed Prompting (Optimized Number Classification)
Design a context-managed prompt with clear instructions and constraints to generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:
• Ensure proper input validation.
• Optimize the logic for efficiency.
• Compare the output with earlier prompting strategies.

Prompt Strategy:

- A context-managed prompt was designed with:
- Role definition (expert developer)
- Clear task description
- Constraints on validation
- Requirement for optimized time complexity.

```python
# Write an optimized Python program to classify a given integer as:
# - Prime # - Composite # - Neither prime nor composite
# Constraints and requirements:
# 1. Validate input to ensure it is an integer. # 2. Handle special cases such as 0, 1, and negative numbers.
# 3. Use an efficient algorithm with time complexity better than O(n). # 4. Use square root optimization for prime checking.
def classify_number(n: int) -> str:
    # Input validation
    if not isinstance(n, int):
        return "Invalid input: not an integer"

    # Handle special cases
    if n < 0:
        return "Neither prime nor composite"
    if n == 0 or n == 1:
        return "Neither prime nor composite"
    if n == 2:
        return "Prime"

    # Efficient prime check using square root optimization
    if n % 2 == 0:
        return "Composite"

    limit = int(n ** 0.5) + 1
    for i in range(3, limit, 2):
        if n % i == 0:
            return "Composite"

    return "Prime"
# Test cases
test_cases = [29, 15, 1, -5, 0, 2, 97, 100]
print("Testing Optimized Prime/Composite Classification Program:")
for value in test_cases:
    try:
        result = classify_number(value)
        print(f"Input: {value} -> Output: {result}")
    except Exception as e:
        print(f"Input: {value} -> Error: {e}")
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR

PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
Testing Optimized Prime/Composite Classification Program:
Input: 29 -> Output: Prime
Input: 15 -> Output: Composite
Input: 1 -> Output: Neither prime nor composite
Input: -5 -> Output: Neither prime nor composite
Input: 0 -> Output: Neither prime nor composite
Input: 2 -> Output: Prime
Input: 97 -> Output: Prime
Input: 100 -> Output: Composite
```

Improvements due to Context Management:

- Full handling of special cases: 0, 1, negative numbers, and 2.
- Efficient prime checking using square root optimization.
- Reduced time complexity from O(n) to O(√n).
- Clear classification into Prime, Composite, or Neither.

Comparison with Earlier Strategies:

- Zero-shot and one-shot often used naive loops up to n-1.
- Few-shot improved correctness but not always efficiency.
- Context-managed prompting produced the most optimized and production-ready solution.

Conclusion:

Context-managed prompting generated the best-quality code among all strategies, demonstrating that detailed instructions and constraints lead to highly optimized and reliable programs.

Question 5: Zero-Shot Prompting (Perfect Number Check)
Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is a perfect number.
Task:
• Record the AI-generated code.
• Test the program with multiple inputs.
• Identify any missing conditions or inefficiencies in the logic.

Prompt Strategy:
A zero-shot prompt was used without any examples to generate a perfect number checking function.

Observations:

- The function correctly identified known perfect numbers such as 6, 28, and 496.
- Negative numbers and zero were rejected.

Missing Conditions and Inefficiencies:

- No input type validation.
- Loop iterated from 1 to n-1, resulting in O(n) time complexity.
- Unnecessary divisor checks reduced performance for large numbers.

```python
## Write a Python function that checks whether a given number is a perfect number.
# The function should return True if it is a perfect number and False otherwise.
#Zero-shot output
def is_perfect_number(num):
    if num <= 1:
        return False  # Perfect numbers are greater than 1
    divisors_sum = 0
    for i in range(1, num // 2 + 1):
        if num % i == 0:
            divisors_sum += i
    return divisors_sum == num
number = int(input("Enter an integer: "))
if is_perfect_number(number):
    print(f"{number} is a perfect number.")
else:
    print(f"{number} is not a perfect number.")
# The function calculates the sum of all proper divisors of the number (excluding the number itself).
# It iterates through all integers from 1 to half of the number, checking if they are divisors.
# If the sum of the divisors equals the original number, it returns True, indicating that it is a perfect number. Otherwise, it returns False.
#Testing with the program with multiple inputs
test_cases = [6, 28, 12, 496, 8128, 15]
print("Testing Zero-Shot Perfect Number Program:")

for value in test_cases:
    result = is_perfect_number(value)
    if result:
        print(f"Input: {value} -> Perfect Number")
    else:
        print(f"Input: {value} -> Not a Perfect Number")
```

Output:

```
PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
Enter an integer: 15
15 is not a perfect number.
Testing Zero-Shot Perfect Number Program:
Input: 6 -> Perfect Number
Input: 28 -> Perfect Number
Input: 12 -> Not a Perfect Number
Input: 496 -> Perfect Number
Input: 8128 -> Perfect Number
Input: 15 -> Not a Perfect Number
```

Conclusion:

Zero-shot prompting produced a correct but inefficient solution. This highlights that without guidance, AI often generates basic algorithms without optimization.

Question 6: Few-Shot Prompting(Even or Odd Classification with Validation)

Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python program that determines whether a given number is even or odd, including proper input validation.

Examples:

- Input: 8 → Output: Even
- Input: 15 → Output: Odd

• Input: 0 → Output: Even

Task:

• Analyze how examples improve input handling and output clarity.

• Test the program with negative numbers and non-integer inputs.

Prompt Strategy:

Few-shot prompting was used with multiple examples and an explicit requirement for input validation.

lab3.py > ...

```python
## Write a Python program to determine whether a given number is even or odd.
# Examples:
# Input: 8 -> Output: Even
# Input: 15 -> Output: Odd
# Input: 0 -> Output: Even
# The program should include proper input validation.
#Fewshot code generation
def is_even_or_odd(num):
    if not isinstance(num, int):
        return "Invalid input: not an integer"
    if num % 2 == 0:
        return "Even"
    else:
        return "Odd"
# Test cases
test_cases = [8, 15, 0, -4, 3.5, 'a']
print("Testing Few-Shot Even/Odd Program:")
for value in test_cases:
    result = is_even_or_odd(value)
    print(f"Input: {value} -> Output: {result}")

```

Output:

```
PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe
Testing Few-Shot Even/Odd Program:
Input: 8 -> Output: Even
Input: 15 -> Output: Odd
Input: 0 -> Output: Even
Input: -4 -> Output: Even
Input: 3.5 -> Output: Invalid input: not an integer
Input: a -> Output: Invalid input: not an integer
```

Improvements due to Examples:

- Clear and consistent output labels: "Even" and "Odd".
- Proper input validation using type checking.
- Correct handling of zero and negative numbers.

Testing with Invalid Outputs:

- Floats, strings, and None were safely rejected.
- Program avoided runtime errors through validation.

Conclusion:

Few-shot prompting significantly improved input handling and output clarity. Providing multiple examples guided the AI to produce a robust and user-friendly solution.