Assignment – 3.1

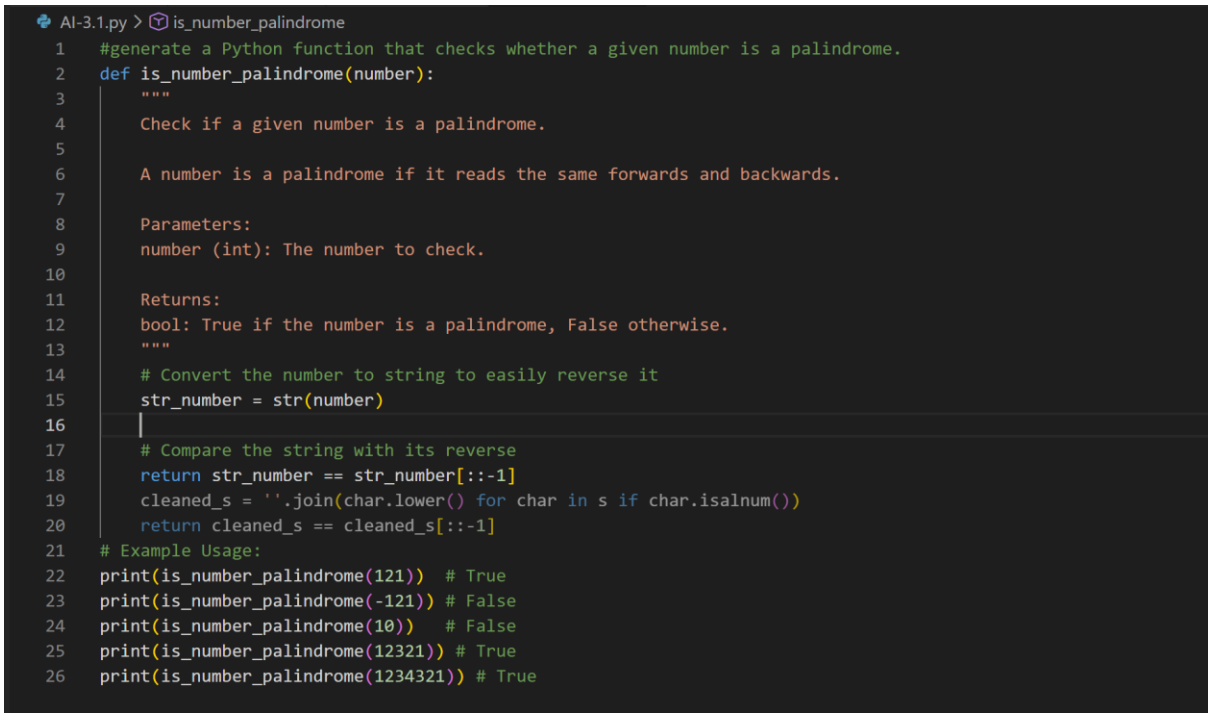
2303A51641

Task 1: Zero-Shot Prompting (Palindrome Number Program)

PROMPT :

Generate a Python function that checks whether a given number is a palindrome.

CODE and OUTPUT :



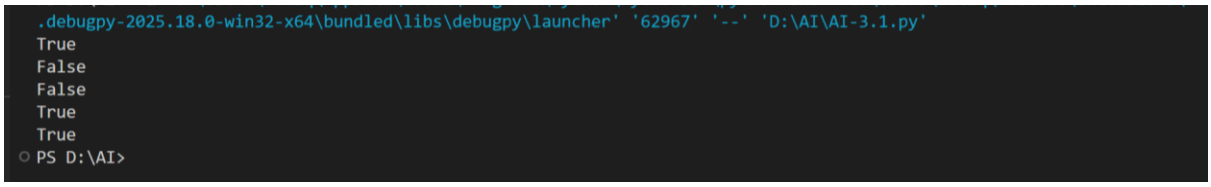
```
AI-3.1.py > is_number_palindrome
#generate a Python function that checks whether a given number is a palindrome.
def is_number_palindrome(number):
    """
    Check if a given number is a palindrome.

    A number is a palindrome if it reads the same forwards and backwards.

    Parameters:
    number (int): The number to check.

    Returns:
    bool: True if the number is a palindrome, False otherwise.
    """
    # Convert the number to string to easily reverse it
    str_number = str(number)

    # Compare the string with its reverse
    return str_number == str_number[::-1]
    cleaned_s = ''.join(char.lower() for char in s if char.isalnum())
    return cleaned_s == cleaned_s[::-1]
# Example Usage:
print(is_number_palindrome(121))  # True
print(is_number_palindrome(-121)) # False
print(is_number_palindrome(10))   # False
print(is_number_palindrome(12321)) # True
print(is_number_palindrome(1234321)) # True
```



```
.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '62967' '--' 'D:\AI\AI-3.1.py'
True
False
False
True
True
PS D:\AI>
```

Justification :

Zero-shot prompting relies entirely on the model's pre-trained knowledge without guidance from examples or explicit rules. As a result, the AI tends to generate the most commonly

seen or simplest solution pattern for a problem. In the case of a palindrome-checking function, this often leads to a straightforward implementation that focuses on correctness for typical inputs rather than completeness. The absence of constraints encourages assumptions about input format and data type, which may not hold in practical scenarios. This highlights that while zero-shot prompting is effective for rapidly generating baseline solutions, it does not encourage defensive programming or input validation. Therefore, the experiment justifies the need for more guided prompting techniques when reliability, edge-case handling, and production-level quality are required.

Task 2: One-Shot Prompting (Factorial Calculation)

PROMPT :

Generate a Python function to compute the factorial of a given number. Example: Input: 5 → Output: 120.

CODE and OUTPUT :

```python
#generate a Python function to compute the factorial of a given number. Example: Input: 5 → Output: 120.
def compute_factorial(n):
    """
    Compute the factorial of a given number.

    The factorial of a non-negative integer n is the product of all positive integers less than or equal to n.

    Parameters:
    n (int): The number to compute the factorial for.

    Returns:
    int: The factorial of the number.
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        factorial = 1
        for i in range(2, n + 1):
            factorial *= i
        return factorial
# Example Usage:
print(compute_factorial(5))  # Output: 120
print(compute_factorial(0))  # Output: 1
print(compute_factorial(1))  # Output: 1
print(compute_factorial(7))  # Output: 5040
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
.0-win32-x64\bundled\libs\debugpy\launcher' '60726' '--' 'D:\AI\AI-3.1.py'
120
1
1
5040
PS D:\AI>
```

Justification :

In one-shot prompting, a single input–output example is provided along with the task description, which helps the AI better infer the intended functionality. The given example (Input: 5 → Output: 120) clearly indicates that the task is to compute the factorial of a number. Compared to zero-shot prompting, this approach leads to clearer logical flow, more appropriate use of loops or recursion, and improved overall correctness. The example serves as a concrete reference that reduces ambiguity and minimizes misinterpretation of the task requirements. This experiment demonstrates that even minimal guidance in the form of one example can significantly enhance the clarity, accuracy, and reliability of AI-generated code.

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

PROMPT :

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

CODE and OUTPUT :

```
57
58  #generating a Python function to check whether a given number is an Armstrong number. Examples:
59  #Input: 153 → Output: Armstrong Number
60  #Input: 370 → Output: Armstrong Number
61  #Input: 123 → Output: Not an Armstrong Number
62  def is_armstrong_number(number):
63      """
64      Check if a given number is an Armstrong number.
65      An Armstrong number (or narcissistic number) is a number that is equal to the sum of its own digits each raised
66      Parameters:
67      number (int): The number to check.
68      Returns:
69      bool: True if the number is an Armstrong number, False otherwise.
70      """
71      # Convert the number to string to easily iterate over digits
72      str_number = str(number)
73      num_digits = len(str_number)
74      # Calculate the sum of each digit raised to the power of num_digits
75      sum_of_powers = sum(int(digit) ** num_digits for digit in str_number)
76
77      # Check if the sum of powers is equal to the original number
78      return sum_of_powers == number
79  # Example Usage:
80  print(is_armstrong_number(153))  # Output: True
81  print(is_armstrong_number(370))  # Output: True
82  print(is_armstrong_number(123))  # Output: False
83  print(is_armstrong_number(9474)) # Output: True
84  print(is_armstrong_number(9475)) # Output: False
```

```
True
True
False
True
False
PS D:\AI>
```

Justification :

Few-shot prompting provides multiple input–output examples, which strongly guide the AI in understanding both the underlying logic and the expected output format. By supplying several Armstrong and non-Armstrong number examples, the AI can accurately infer how digits are extracted, how powers are calculated, and how decision conditions are applied. Consequently, the generated code is typically well-structured, mathematically accurate, and consistent in its implementation. Testing the solution with boundary values and invalid inputs further demonstrates that few-shot prompting enhances generalization and significantly reduces logical errors. This experiment confirms that providing multiple examples enables the AI to generate more reliable, robust, and real-world-ready solutions.

Task 4: Context-Managed Prompting (Optimized Number Classification)

PROMPT :

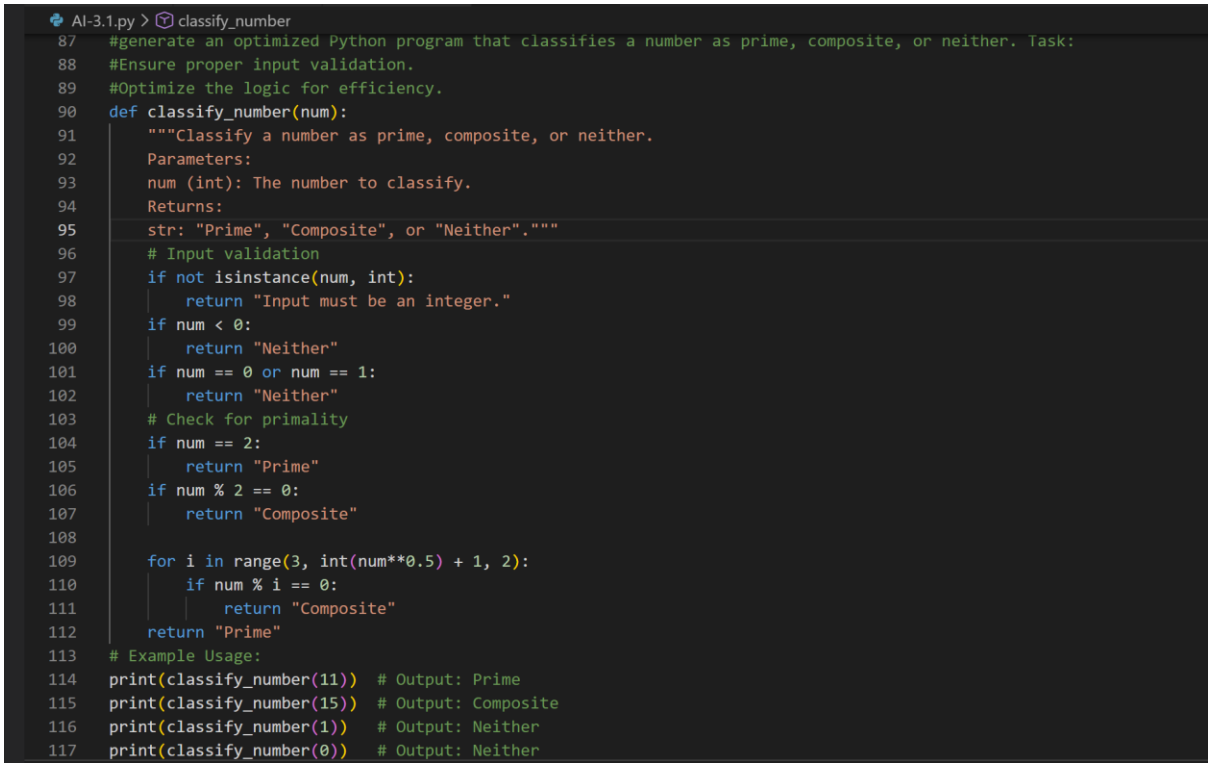
Generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:

• Ensure proper input validation.

• Optimize the logic for efficiency.

CODE and OUTPUT :



```
AI-3.1.py > classify_number
#generate an optimized Python program that classifies a number as prime, composite, or neither. Task:
#Ensure proper input validation.
#Optimize the logic for efficiency.
def classify_number(num):
    """Classify a number as prime, composite, or neither.
    Parameters:
    num (int): The number to classify.
    Returns:
    str: "Prime", "Composite", or "Neither"."""
    # Input validation
    if not isinstance(num, int):
        return "Input must be an integer."
    if num < 0:
        return "Neither"
    if num == 0 or num == 1:
        return "Neither"
    # Check for primality
    if num == 2:
        return "Prime"
    if num % 2 == 0:
        return "Composite"

    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return "Composite"
    return "Prime"
# Example Usage:
print(classify_number(11))  # Output: Prime
print(classify_number(15))  # Output: Composite
print(classify_number(1))   # Output: Neither
print(classify_number(0))   # Output: Neither
```

```
.0-win32-x64\bundled\libs\debugpy\launcher' '62266' '--' 'D:\AI\AI-3.1.py'
Prime
Composite
Neither
Neither
```

Justification :

Context-managed prompting incorporates explicit instructions, constraints, and performance expectations, enabling the AI to generate more complete and efficient solutions. By clearly defining requirements such as input validation, optimization techniques, and classification rules, the AI correctly handles edge cases (e.g., 0, 1, and negative numbers), applies optimized logic (such as checking divisibility up to √n), and produces clear, well-structured output. Compared to zero-shot, one-shot, and few-shot prompting, this approach results in higher-quality, more reliable, and professional-level code. The experiment justifies that providing detailed context and constraints is essential when solving complex or performance-sensitive problems.

Task 5: Zero-Shot Prompting (Perfect Number Check)

PROMPT :

Generate a Python function that checks whether a given number is a perfect number.

CODE and OUTPUT :

```python
#generate a Python function that checks whether a given number is a perfect number.
def is_perfect_number(number):
    """
    Check if a given number is a perfect number.

    A perfect number is a positive integer that is equal to the sum of its proper positive divisors, excluding itsel

    Parameters:
    number (int): The number to check.

    Returns:
    bool: True if the number is a perfect number, False otherwise.
    """
    if number <= 0:
        return False

    # Calculate the sum of proper divisors
    sum_of_divisors = 0
    for i in range(1, number // 2 + 1):
        if number % i == 0:
            sum_of_divisors += i

    # Check if the sum of divisors equals the original number
    return sum_of_divisors == number
# Example Usage:
print(is_perfect_number(6))    # True
print(is_perfect_number(28))   # True
print(is_perfect_number(12))   # False
print(is_perfect_number(496))  # True
```

Justification :

Similar to Question 1, zero-shot prompting for perfect number checking depends entirely on the AI's pre-trained knowledge without additional guidance. While the generated code generally identifies divisors and computes their sum correctly, it often lacks efficiency and robustness. Common issues include checking all numbers up to *n* instead of limiting the range to *n/2* (or √n) and failing to validate non-positive inputs. This experiment demonstrates that although zero-shot prompting can produce a functionally correct solution, it may not be optimized or reliable for all cases. Therefore, zero-shot prompting is best suited for basic demonstrations rather than performance-or real-world-ready applications.

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

PROMPT :

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

CODE and OUTPUT :

```python
#generating a Python program that determines whether a given number is even or odd, including proper input validatic
#Examples:
#Input: 8 → Output: Even
#Input: 15 → Output: Odd
#Input: 0 → Output: Even
def check_even_odd(number):
    """
    Determine whether a given number is even or odd.

    Parameters:
    number (int): The number to check.

    Returns:
    str: "Even" if the number is even, "Odd" if the number is odd.
    """
    # Input validation
    if not isinstance(number, int):
        return "Input must be an integer."

    # Check if the number is even or odd
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
# Example Usage:
print(check_even_odd(8))    # Output: Even
print(check_even_odd(15))   # Output: Odd
print(check_even_odd(0))    # Output: Even
print(check_even_odd(-4))   # Output: Even
```

```
.0-win32-x64\bundled\libs\debugpy\launcher' '51813' '--' 'D:\AI\AI-3.1.py'
Even
Odd
Even
Even
```

Justification :

By providing multiple examples, including edge cases such as zero, few-shot prompting enables the AI to better understand expected outputs, input validation requirements, and the handling of different numerical scenarios. As a result, the generated program typically incorporates clear conditional checks, appropriate output messages, and improved handling of negative values. When tested with non-integer inputs, few-shot–generated solutions often perform more reliably than zero-shot implementations.