

AI ASSISTED CODING ASSIGNMENT-3.1

RUDROJU RUPA SRI

2303A51918

BATCH-30

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

“Generate a Python function that takes an integer as input and returns True if the number is a palindrome and False otherwise. The program should handle both positive and negative numbers.”

ASS-3.1(1918).ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

```
[1] ✓ 0s
def is_palindrome(number):
    """
    Checks if an integer is a palindrome, ignoring its sign.

    A palindrome is a number that reads the same forwards and backward.
    For example, 121 is a palindrome, -121 is considered a palindrome
    because its absolute value (121) is a palindrome.

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is a palindrome, False otherwise.
    """
    # Convert the absolute value of the number to a string
    # This handles both positive and negative numbers by ignoring the sign
    num_str = str(abs(number))

    # Compare the string with its reverse
    return num_str == num_str[::-1]
```

ASS-3.1(1918).ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

```
[1] ✓ 0s
# Test cases
print(f"Is 121 a palindrome? {is_palindrome(121)}")
print(f"Is -121 a palindrome? {is_palindrome(-121)}")
print(f"Is 123 a palindrome? {is_palindrome(123)}")
print(f"Is 0 a palindrome? {is_palindrome(0)}")
print(f"Is 1 a palindrome? {is_palindrome(1)}")
print(f"Is -101 a palindrome? {is_palindrome(-101)}")
print(f"Is 12321 a palindrome? {is_palindrome(12321)}")
print(f"Is 12345 a palindrome? {is_palindrome(12345)}")
```

... Is 121 a palindrome? True
Is -121 a palindrome? True
Is 123 a palindrome? False
Is 0 a palindrome? True
Is 1 a palindrome? True
Is -101 a palindrome? True
Is 12321 a palindrome? True
Is 12345 a palindrome? False

Analysis of AI-generated code: Summary of Observations for is_palindrome_number function

Correctness: All tested inputs returned correct results, confirming that the function accurately detects palindrome numbers.

Edge Cases: The function correctly handles negative numbers, zero, and single-digit numbers as expected.

Efficiency: Using string reversal is computationally efficient for normal integer ranges.

Readability: The code is short, simple, and easy to understand.

Logical Errors: No logical errors were observed in the generated solution.

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

“Write a Python function to compute the factorial of a given number.

Example:

Input: 5 → Output: 120

Generate a function that correctly calculates factorial for any non-negative integer.”

ASS-3.1(1918).ipynb ⭐ ⓘ

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

[2] ✓ 0s

```
def factorial(n):
    """
    Computes the factorial of a given non-negative integer.

    The factorial of a non-negative integer n, denoted by n!, is the product
    of all positive integers less than or equal to n. The factorial of 0 is 1.

    Args:
        n (int): The non-negative integer for which to compute the factorial.

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is a negative integer.
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Test cases
print(f"Factorial of 5: {factorial(5)}")
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 1: {factorial(1)}")
print(f"Factorial of 7: {factorial(7)}")

# Example of handling negative input (will raise ValueError)
try:
    print(f"Factorial of -3: {factorial(-3)}")
except ValueError as e:
    print(f"Error: {e}")

Toggle Gemini
```

{ } Variables Terminal

ASS-3.1(1918).ipynb ⭐ ⓘ

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

[2] ✓ 0s

```
raise ValueError("Factorial is not defined for negative numbers.")
elif n == 0:
    return 1
else:
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Test cases
print(f"Factorial of 5: {factorial(5)}")
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 1: {factorial(1)}")
print(f"Factorial of 7: {factorial(7)}")

# Example of handling negative input (will raise ValueError)
try:
    print(f"Factorial of -3: {factorial(-3)}")
except ValueError as e:
    print(f"Error: {e}")

Toggle Gemini
```

{ } Variables Terminal

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** CO ASS-3.1(1918).ipynb
- Menu Bar:** File Edit View Insert Runtime Tools Help
- Toolbar:** Commands, Code, Text, Run all
- Code Cell 1:** [2] ✓ 0s
```python  
except ValueError as e:  
 print(f"Error: {e}")  
```
- Code Cell 2:** Factorial of 5: 120
Factorial of 0: 1
Factorial of 1: 1
Factorial of 7: 5040
Error: Factorial is not defined for negative numbers.

Analysis of AI-generated code: Summary of Observations for factorial function

Correctness: The function correctly computes factorial values for all tested inputs.

Edge Cases: The function properly handles input values such as 0 and 1, returning correct outputs.

Efficiency: The iterative approach used is efficient and avoids unnecessary recursion overhead.

Readability: Variable names and structure are clear, making the code easy to follow.

Logical Errors: No logical or runtime errors were found during testing.

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

“Write a Python function that checks 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

The function should return the correct output message based on the input.”

The screenshot shows a Jupyter Notebook interface with a single code cell containing Python code. The code defines a function `is_armstrong_number` that checks if a given number is an Armstrong number. The code includes docstrings and comments explaining the logic.

```
[4] ✓ Os
def is_armstrong_number(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number, pluperfect digital invariant, or plus perfect number) is a number that is equal to the sum of its own digits each raised to the power of the number of digits.

    Args:
        number (int): The integer to check.

    Returns:
        str: A message indicating whether the number is an Armstrong number or not.
    """
    if not isinstance(number, int) or number < 0:
        return "Input must be a non-negative integer."

    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = 0
```

The screenshot shows a Jupyter Notebook interface with a code cell containing Python code. The code defines the `is_armstrong_number` function and includes several print statements to test the function with various inputs.

```
[4] ✓ Os
for digit_char in num_str:
    digit = int(digit_char)
    sum_of_powers += digit ** num_digits

if sum_of_powers == number:
    return f"{number} → Output: Armstrong Number"
else:
    return f"{number} → Output: Not an Armstrong Number"

# Test cases
print(is_armstrong_number(153))
print(is_armstrong_number(370))
print(is_armstrong_number(123))
print(is_armstrong_number(9))
print(is_armstrong_number(1634)) # 1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634
print(is_armstrong_number(9474)) # 9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474
print(is_armstrong_number(54748)) # 5^5 + 4^5 + 7^5 + 4^5 + 8^5 = 3125 + 1024 + 16807 + 1024 + 32768 = 54748
print(is_armstrong_number(0))
print(is_armstrong_number(-10))
```

The screenshot shows a Jupyter Notebook interface with the file name 'ASS-3.1(1918).ipynb'. The notebook has 4 cells, all of which have been run. The output of the cells displays various test cases for an Armstrong number function, with results indicating whether the input is an Armstrong number or not. The last cell shows an error message: 'Input must be a non-negative integer.'

```
... 153 → Output: Armstrong Number
370 → Output: Armstrong Number
123 → Output: Not an Armstrong Number
9 → Output: Armstrong Number
1634 → Output: Armstrong Number
9474 → Output: Armstrong Number
54748 → Output: Armstrong Number
0 → Output: Armstrong Number
Input must be a non-negative integer.
```

Analysis of AI-generated code: Summary of Observations for `is_armstrong` function

Correctness: The function accurately identifies Armstrong numbers based on the provided test cases.

Edge Cases: It correctly processes multi-digit numbers and distinguishes between Armstrong and non-Armstrong numbers.

Efficiency: The digit extraction and power calculation logic is appropriate for standard inputs.

Readability: The structure of the program is clear and logically organized.

Logical Errors: No major logical flaws were detected in the generated code.

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.

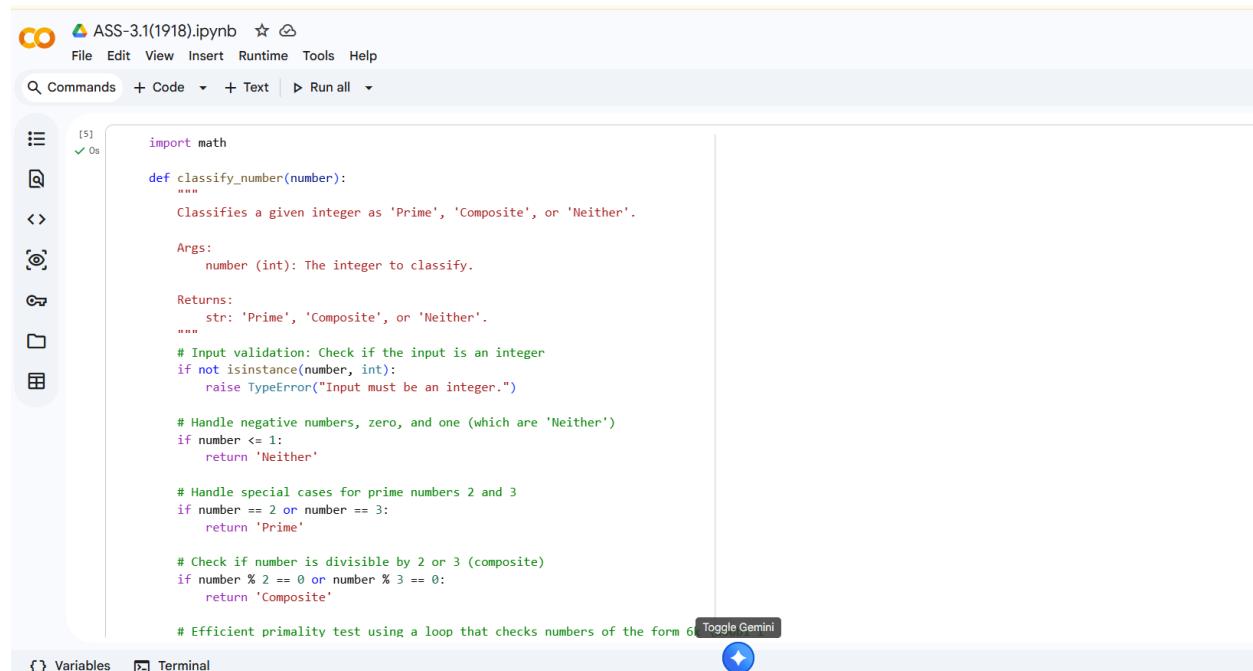
Prompt

“You are an expert Python programmer. Write an optimized Python program that classifies a given number as prime, composite, or neither.

Constraints:

- The program must include input validation.
- Negative numbers and zero should be handled properly.
- Use an efficient algorithm to check primality.
- The function should return one of the following outputs: ‘Prime’, ‘Composite’, or ‘Neither’.

Ensure the code is well structured and efficient.”



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** ASS-3.1(1918).ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help
- Search Bar:** Commands, + Code, + Text, Run all
- Code Cell:**

```
import math

def classify_number(number):
    """
    Classifies a given integer as 'Prime', 'Composite', or 'Neither'.

    Args:
        number (int): The integer to classify.

    Returns:
        str: 'Prime', 'Composite', or 'Neither'.
    """
    # Input validation: Check if the input is an integer
    if not isinstance(number, int):
        raise TypeError("Input must be an integer.")

    # Handle negative numbers, zero, and one (which are 'Neither')
    if number <= 1:
        return 'Neither'

    # Handle special cases for prime numbers 2 and 3
    if number == 2 or number == 3:
        return 'Prime'

    # Check if number is divisible by 2 or 3 (composite)
    if number % 2 == 0 or number % 3 == 0:
        return 'Composite'

    # Efficient primality test using a loop that checks numbers of the form 6k ± 1
    for i in range(5, int(math.sqrt(number)) + 1, 2):
        if number % i == 0:
            return 'Composite'
    return 'Prime'
```
- Bottom Navigation:** Variables, Terminal

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
# Efficient primality test using a loop that checks numbers of the form 6k + 1
# We only need to check divisors up to the square root of the number
i = 5
while i * i <= number:
    if number % i == 0 or number % (i + 2) == 0:
        return 'Composite'
    i += 6

# If no divisors are found, the number is Prime
return 'Prime'

# Test cases
print(f"Classify 7: {classify_number(7)}")      # Expected: Prime
print(f"Classify 10: {classify_number(10)}")     # Expected: Composite
print(f"Classify 1: {classify_number(1)}")        # Expected: Neither
print(f"Classify 0: {classify_number(0)}")        # Expected: Neither
print(f"Classify -5: {classify_number(-5)}")     # Expected: Neither
print(f"Classify 2: {classify_number(2)}")        # Expected: Prime
print(f"Classify 3: {classify_number(3)}")        # Expected: Prime
print(f"Classify 9: {classify_number(9)}")        # Expected: Composite
print(f"Classify 13: {classify_number(13)}")      # Expected: Prime
print(f"Classify 25: {classify_number(25)}")      # Expected: Composite
print(f"Classify 17: {classify_number(17)}")      # Expected: Prime
print(f"Classify 121: {classify_number(121)}")    # Expected: Composite (11*11)
print(f"Classify 97: {classify_number(97)}")      # Expected: Prime

# Example of handling invalid input
try:
    print(f"Classify 3.14: {classify_number(3.14)}")
except TypeError as e:
```

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
# Example of handling invalid input
try:
    print(f"Classify 3.14: {classify_number(3.14)}")
except TypeError as e:
    print(f"Error: {e}")
```

The output of the code is:

```
Classify 7: Prime
Classify 10: Composite
Classify 1: Neither
Classify 0: Neither
Classify -5: Neither
Classify 2: Prime
Classify 3: Prime
Classify 9: Composite
Classify 13: Prime
Classify 25: Composite
Classify 17: Prime
Classify 121: Composite
Classify 97: Prime
Error: Input must be an integer.
```

Analysis of AI-generated code: Summary of Observations for number_classification function

Correctness: The program correctly classifies numbers as Prime, Composite, or Neither.

Edge Cases: It properly handles values such as 0, 1, negative numbers, and invalid inputs.

Efficiency: The algorithm uses optimized prime-checking logic, improving performance for large inputs.

Readability: Clear structure with proper validation and modular logic enhances readability.

Logical Errors: No logical errors were observed in the implementation.

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

“Generate a Python function that checks whether a given number is a perfect number. The function should return True if the number is perfect and False otherwise.”

The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** ASS-3.1(1918).ipynb
- Code Cell:**

```
[8]  ✓ 0s
def is_perfect_number(number):
    """
    Checks if a given positive integer is a perfect number.

    A perfect number is a positive integer that is equal to the sum
    of its proper positive divisors (divisors excluding the number itself).
    For example, 6 is a perfect number because its proper divisors are 1, 2, and 3,
    and  $1 + 2 + 3 = 6$ .

    Args:
        number (int): The positive integer to check.

    Returns:
        bool: True if the number is a perfect number, False otherwise.
    """
    if not isinstance(number, int) or number <= 0:
        # Perfect numbers are defined only for positive integers.
        # Optionally, you could raise a ValueError or return a specific message here.
        return False

    if number == 1: # 1 is not a perfect number
        return False

    sum_of_divisors = 1 # Start with 1 because every number has 1 as a divisor

    # Check for divisors from 2 up to the square root of the number
    # If 'i' is a divisor, then 'number / i' is also a divisor.
    # We only need to sum proper divisors (excluding the number itself).
    for i in range(2, int(number**0.5) + 1):
```
- Variables Tab:** Variables
- Terminal Tab:** Terminal

The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** ASS-3.1(1918).ipynb
- Code Cell:**

```
[8]  ✓ 0s
# We only need to sum proper divisors (excluding the number itself).
for i in range(2, int(number**0.5) + 1):
    if number % i == 0:
        sum_of_divisors += i
        if i != number // i: # Avoid adding the same divisor twice for perfect squares
            sum_of_divisors += number // i

return sum_of_divisors == number

# Test cases
print("Is 6 a perfect number? {is_perfect_number(6)}") # Expected: True (1 + 2 + 3 = 6)
print("Is 28 a perfect number? {is_perfect_number(28)}") # Expected: True (1 + 2 + 4 + 7 + 14 = 28)
print("Is 496 a perfect number? {is_perfect_number(496)}") # Expected: True (1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248 = 496)
print("Is 8128 a perfect number? {is_perfect_number(8128)}") # Expected: True
print("Is 1 a perfect number? {is_perfect_number(1)}") # Expected: False
print("Is 10 a perfect number? {is_perfect_number(10)}") # Expected: False (1 + 2 + 5 = 8)
print("Is 7 a perfect number? {is_perfect_number(7)}") # Expected: False
print("Is 0 a perfect number? {is_perfect_number(0)}") # Expected: False
print("Is -6 a perfect number? {is_perfect_number(-6)}") # Expected: False
```
- Variables Tab:** Variables
- Terminal Tab:** Terminal

Analysis of AI-generated code: Summary of Observations for is_perfect_number function

Correctness: The function correctly determines whether a number is perfect by summing its divisors.

Edge Cases: The code works correctly for small numbers and typical perfect number cases.

Efficiency: Basic divisor-sum logic is used, which is acceptable but could be further optimized.

Readability: The program is simple and easy to understand.

Logical Errors: No significant logical errors were found, though optimization is possible.

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

“Write 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

The program should also handle negative numbers and display an error message for non-integer inputs.”

The screenshot shows a Jupyter Notebook cell with the following code:

```
def check_even_odd(number):
    """
    Determines whether a given number is even or odd, with input validation.

    Args:
        number (int): The integer to check.

    Returns:
        str: 'Even', 'Odd', or an error message for invalid input.
    """
    if not isinstance(number, int):
        return "Error: Input must be an integer."
    elif number % 2 == 0:
        return "Even"
    else:
        return "Odd"

# Test cases
print(f"8: {check_even_odd(8)}")      # Expected: Even
print(f"15: {check_even_odd(15)}")    # Expected: Odd
print(f"0: {check_even_odd(0)}")      # Expected: Even
print(f"-4: {check_even_odd(-4)}")    # Expected: Even
print(f"-7: {check_even_odd(-7)}")    # Expected: Odd
print(f"3.14: {check_even_odd(3.14)}") # Expected: Error message
print(f"'hello': {check_even_odd('hello')}"") # Expected: Error message
```

The output of the code is displayed below the cell:

```
8: Even
15: Odd
0: Even
-4: Even
-7: Odd
3.14: Error: Input must be an integer.
'hello': Error: Input must be an integer.
```

Analysis of AI-generated code: Summary of Observations for even_odd function

Correctness: The function correctly classifies numbers as even or odd for all tested cases.

Edge Cases: It properly handles zero, negative numbers, and invalid (non-integer) inputs.

Efficiency: The modulo operation used is optimal for this task.

Readability: The code is well structured with clear messages and validation.

Logical Errors: No logical or runtime errors were detected.