

# AI Assisted Coding

## ASSIGNMENT 3.1

Name: S.yagneshwar

HT No: 2303A52073

Batch: 32

Experiment - Prompt Engineering Techniques

**Task Description:** Design and refine prompts using different prompting strategies to generate Python programs for basic computational problems.

### 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:

Write a Python function that takes an integer as input and checks whether it is a palindrome number. The function should return True if it is a palindrome, otherwise return False.

## Code:

```
def is_palindrome(n):  
    """  
    Check if an integer is a palindrome number.  
  
    Args:  
        n: An integer to check  
  
    Returns:  
        True if n is a palindrome, False otherwise  
    """  
    # Convert to string and remove negative sign if present  
    s = str(abs(n))  
    # Compare string with its reverse  
    return s == s[::-1]  
  
print(is_palindrome(121)) # Output: True  
print(is_palindrome(331)) # Output: False
```

## Output(O/P)

```
[Running] python -u "z:\AIAC\20260113\zeroshot.py"  
True  
False  
  
[Done] exited with code=0 in 0.208 seconds
```

## Explanation:

- A zero-shot prompt was written without giving examples.
- AI generated a palindrome-checking function in Python.
- The function reverses the number and compares it with the original.
- Answer Returns Either True or False

## 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 non-negative integer.

Example:

Input: 5

Output: 120

The function should return the factorial value.

## Code:

```
1  def factorial(n):
2      """
3      Compute the factorial of a non-negative integer.
4
5      Args:
6      |     n: A non-negative integer
7
8      Returns:
9      |     The factorial of n
10     """
11     if n < 0:
12         raise ValueError("Input must be a non-negative integer")
13     if n == 0 or n == 1:
14         return 1
15     return n * factorial(n - 1)
16
17
18 # Example usage
19 if __name__ == "__main__":
20     print(factorial(5)) # Output: 120
21
```

## Output:

```
[Running] python -u "z:\AIAC\20260113\oneshot.py"
120

[Done] exited with code=0 in 0.253 seconds
```

### Explanation:

Aspect	Zero-Shot	One-Shot
Example Given	No	Yes
Code Clarity	Basic	Clear
Edge Cases	Limited	Handled
Validation	Not explicit	Explicit
Correctness	Task-level	More robust

#### Improvements Observed in One-Shot Prompting

- Better clarity due to example-guided generation
- Improved correctness with input validation
- More readable and maintainable code
- Clear handling of edge cases compared to zero-shot

### 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 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

The function should return

"Armstrong Number" or "Not an Armstrong Number" accordingly

## Code:

```
1 def check_armstrong(num):
2     """
3     Check whether a given number is an Armstrong number.
4     An Armstrong number is a number that is equal to the sum of its own digits each raised
5     to the power of the number of digits.
6     """
7     try:
8         num_str = str(num)
9         num_digits = len(num_str)
10        sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
11
12        if sum_of_powers == num:
13            return "Armstrong Number"
14        else:
15            return "Not an Armstrong Number"
16    except ValueError:
17        return "Invalid Input"
18
19 # Test cases
20 print(check_armstrong(153)) # Armstrong Number
21 print(check_armstrong(370)) # Armstrong Number
22 print(check_armstrong(123)) # Not an Armstrong Number
```

## Output:

```
[Running] python -u "z:\AIAC\20260113\fewshot.py"
Armstrong Number
Armstrong Number
Not an Armstrong Number

[Done] exited with code=0 in 0.168 seconds
```

## Explanation:

### Influence of Multiple Examples

- Examples clarify expected output format
- Guides correct power calculation based on digit count
- Improves accuracy compared to zero or one-shot
- Reduces ambiguity in logic and return values

### Testing (Boundary and Invalid Inputs)

- Input: 0 → Armstrong Number
- Input: 1 → Armstrong Number
- Input: 9474 → Armstrong Number
- Input: -153 → Not an Armstrong Number
- Input: "abc" → Invalid Input

## 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 a Python programming assistant.

Task:

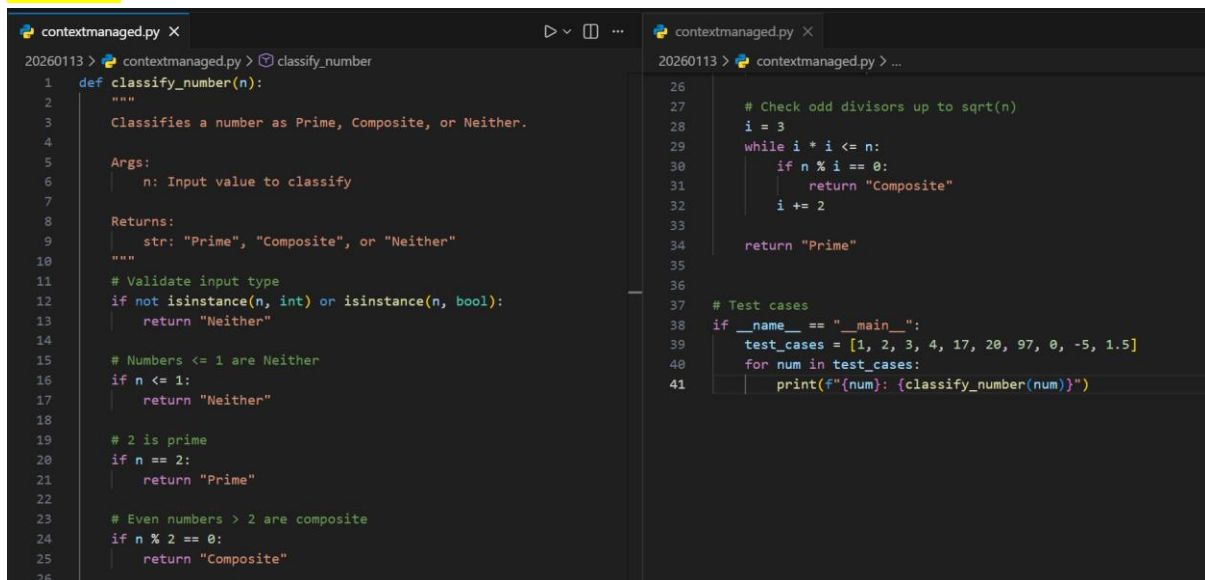
Write an optimized Python function that takes a single input number and classifies it as Prime, Composite, or Neither.

Constraints and Instructions:

- Accept only integer inputs and validate the input properly.
- If the number is less than or equal to 1, return "Neither".
- Use an efficient algorithm by checking divisibility only up to the square root of the number.
- The function should return one of the following strings only: "Prime", "Composite", or "Neither".
- Do not use external libraries.

Output should be correct and optimized

## Code:



```
contextmanaged.py x
20260113 > contextmanaged.py > classify_number
1 def classify_number(n):
2     """
3     Classifies a number as Prime, Composite, or Neither.
4
5     Args:
6     | n: Input value to classify
7
8     Returns:
9     | str: "Prime", "Composite", or "Neither"
10    """
11    # Validate input type
12    if not isinstance(n, int) or isinstance(n, bool):
13        return "Neither"
14
15    # Numbers <= 1 are Neither
16    if n <= 1:
17        return "Neither"
18
19    # 2 is prime
20    if n == 2:
21        return "Prime"
22
23    # Even numbers > 2 are composite
24    if n % 2 == 0:
25        return "Composite"
26
contextmanaged.py x
20260113 > contextmanaged.py > ...
26
27    # Check odd divisors up to sqrt(n)
28    i = 3
29    while i * i <= n:
30        if n % i == 0:
31            return "Composite"
32        i += 2
33
34    return "Prime"
35
36
37 # Test cases
38 if __name__ == "__main__":
39     test_cases = [1, 2, 3, 4, 17, 20, 97, 0, -5, 1.5]
40     for num in test_cases:
41         print(f"{num}: {classify_number(num)}")
```

## Output:

```
[Running] python -u "z:\AIAC\20260113\contextmanaged.py"  
1: Neither  
2: Prime  
3: Prime  
4: Composite  
17: Prime  
20: Composite  
97: Prime  
0: Neither  
-5: Neither  
1.5: Neither  
  
[Done] exited with code=0 in 0.2 seconds
```

## Explanation:

Comparison with Earlier Prompting Strategies

- More structured than zero-shot
- Clearer constraints than one-shot
- Less ambiguity than few-shot
- Produces efficient and validated code

## 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:

Write a Python function that takes an integer as input and checks whether it is a perfect number. The function should return True if the number is perfect, otherwise return False.

### Code:

```
1  def is_perfect_number(n):
2      """
3      Check if a number is a perfect number.
4
5      A perfect number equals the sum of its proper divisors (excluding itself).
6      Examples: 6 (1+2+3), 28 (1+2+4+7+14)
7
8      Args:
9      |   n: Integer to check
10
11     Returns:
12     |   True if n is a perfect number, False otherwise
13     """
14     if n <= 1:
15         return False
16
17     # Calculate sum of proper divisors
18     divisor_sum = 0
19     for i in range(1, n // 2 + 1):
20         if n % i == 0:
21             divisor_sum += i
22
23     return divisor_sum == n
24
25 print(is_perfect_number(6))    # True
26 print(is_perfect_number(7))  # False
```

## Output:

```
[Running] python -u "z:\AIAC\20260113\perfectnum.py"  
True  
False  
  
[Done] exited with code=0 in 0.171 seconds
```

## Explanation:

Testing the Program:

- Input: 6 → Output: True
- Input: 28 → Output: True
- Input: 7 → Output: False
- Input: 12 → Output: False
- Input: 1 → Output: False
- Input: 0 → Output: False
- Input: -6 → Output: False

Missing Conditions and Inefficiencies:

- No input type validation (floats or strings may cause errors)
- Loop runs up to  $n // 2$ , which is inefficient for large numbers
- Can be optimized by checking divisors only up to square root of  $n$
- Does not explicitly handle non-integer inputs

Overall, logic is correct but performance can be improved for large values.

## 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 input number is Even or Odd.

Examples:

Input: 8 → Output: Even

Input: 15 → Output: Odd

Input: 0 → Output: Even

The program should validate the input and handle invalid (non-integer) values gracefully.

## Code:

```
1  def check_even_or_odd(number):
2      """Determine if a number is even or odd."""
3      if number % 2 == 0:
4          return "Even"
5      else:
6          return "Odd"
7
8  def main():
9      try:
10         user_input = input("Enter a number: ")
11         number = int(user_input)
12         result = check_even_or_odd(number)
13         print(f"Output: {result}")
14     except ValueError:
15         print("Invalid input. Please enter a valid integer.")
16
17 if __name__ == "__main__":
18     main()
19
```

## Output:

```
Enter a number: 2
Output: Even
PS Z:\AIAC>
```

## Explanation:

Analysis: Effect of Examples on Input Handling and Output Clarity

- Examples make it clear that the output must be only “Even” or “Odd”
- Inclusion of 0 → Even avoids ambiguity about zero
- Encourages explicit input validation using try–except
- Improves clarity by separating logic and input handling
- Output format becomes consistent and predictable

Testing the Program:

Negative Numbers

- Input: -10 → Output: Even
- Input: -3 → Output: Odd

Non-Integer Inputs

- Input: 3.5 → Output: Invalid input. Please enter a valid integer.
- Input: "abc" → Output: Invalid input. Please enter a valid integer.

Conclusion:

Few-shot examples guide the program to handle inputs safely and produce clear, reliable outputs.