

AI Assisted Coding

ASSIGNMENT 3.1

Name: A. Tejasri

HT No: 2303A52455

Batch: 31

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:

```
Day_3.py > assignment_3_1 > ...
1 def is_palindrome(n):
2     """
3     Check whether a number is a palindrome.
4
5     Args:
6     | n: An integer to check
7
8     Returns:
9     | True if the number is a palindrome, False otherwise
10    """
11    # Convert to string and remove negative sign if present
12    num_str = str(abs(n))
13
14    # Check if the string is equal to its reverse
15    return num_str == num_str[::-1]
16
17
18 # Test cases
19 if __name__ == "__main__":
20     print(is_palindrome(121))    # True
21     print(is_palindrome(123))    # False
22     print(is_palindrome(1001))    # True
23     print(is_palindrome(9))    # True
24     print(is_palindrome(-121))    # True
25     print(is_palindrome(0))    # True
26
```

Output(O/P)

```
\6th semester\Python\Day_3.py\assignment_3_1'
True
False
True
True
True
True
PS D:\6th semester\Python>
```

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:

```
Day_3.py > assignment3_1 > task2.py > factorial
1  def factorial(n):
2      """
3      Compute the factorial of a given non-negative integer.
4      Args:
5      |   n: A non-negative integer
6
7      Returns:
8      |   The factorial of n
9
10     Raises:
11     |   ValueError: If n is negative
12     """
13     if n < 0:
14         raise ValueError("Factorial is not defined for negative numbers")
15
16     if n == 0 or n == 1:
17         return 1
18
19     result = 1
20     for i in range(2, n + 1):
21         result *= i
22
23     return result
24
25 # Test the function
26 if __name__ == "__main__":
27     print(f"Factorial of 0: {factorial(0)}")
28     print(f"Factorial of 5: {factorial(5)}")
29     print(f"Factorial of 10: {factorial(10)}")
```

Output:

```
ebugpy\launcher' '54806' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task2.py'
Factorial of 0: 1
Factorial of 5: 120
Factorial of 10: 3628800
PS D:\6th semester\Python>
```

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:

```
Day_3.py > assignment3_1 > task3.py > ...
1  def is_armstrong_number(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
5      its own digits each raised to the power of the number of digits.
6      """
7      num_str = str(abs(num))
8      num_digits = len(num_str)
9      armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
10     return armstrong_sum == num
11
12
13     # Test cases
14     if __name__ == "__main__":
15         test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 123, 100]
16
17         for number in test_numbers:
18             if is_armstrong_number(number):
19                 print(f"{number} is an Armstrong number")
20             else:
21                 print(f"{number} is not an Armstrong number")
22
```

Output:

```
ebugpy\launcher' '55332' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task3.py'
153 is an Armstrong number
370 is an Armstrong number
371 is an Armstrong number
407 is an Armstrong number
1634 is an Armstrong number
8208 is an Armstrong number
9474 is an Armstrong number
123 is not an Armstrong number
100 is not an Armstrong number
PS D:\6th semester\Python>
```

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:

```
Day_3.py > assignment3_1 > task4.py > classify_number
1 def classify_number(num):
2     """
3     Classifies a number as Prime, Composite, or Neither.
4     Args:
5     | num: The input to be classified
6     Returns:
7     | str: "Prime", "Composite", or "Neither"
8     """
9     # Validate input - check if it's an integer
10    if not isinstance(num, int) or isinstance(num, bool):
11        return "Neither"
12    # Numbers <= 1 are neither prime nor composite
13    if num <= 1:
14        return "Neither"
15    # 2 is the only even prime number
16    if num == 2:
17        return "Prime"
18    # All other even numbers are composite
19    if num % 2 == 0:
20        return "Composite"
21    # Check odd divisors up to sqrt(num)
22    i = 3
23    while i * i <= num:
24        if num % i == 0:
25            return "Composite"
26        i += 2
27    return "Prime"
28
29 # Test cases
30 if __name__ == "__main__":
31     test_cases = [1, 2, 3, 4, 5, 10, 17, 25, 0, -5, 100, 97]
32
33     for test in test_cases:
34         print(f"{test}: {classify_number(test)}")
35
```

Output:

```
ebugpy\launcher' '50093' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task4.py'
1: Neither
2: Prime
3: Prime
4: Composite
5: Prime
10: Composite
17: Prime
25: Composite
0: Neither
-5: Neither
100: Composite
97: Prime
PS D:\6th semester\Python>
```

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:

```
Day_3.py > assignment3_1 > task5.py > ...
1 def is_perfect_number(n: int) -> bool:
2     if n <= 1:
3         return False
4
5     divisors_sum = 1 # 1 is always a divisor
6     # Only check up to sqrt(n) to optimize
7     for i in range(2, int(n**0.5) + 1):
8         if n % i == 0:
9             divisors_sum += i
10            if i != n // i: # Avoid adding square root twice
11                divisors_sum += n // i
12
13    return divisors_sum == n
14 print(is_perfect_number(6)) # True
15 print(is_perfect_number(28)) # True
16 print(is_perfect_number(12)) # False
17 print(is_perfect_number(496)) # True
```

Output:

```
ebugpy\launcher' '62855' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task5.py'
True
True
False
True
PS D:\6th semester\Python>
```

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.