

AI-Assisted Coding

Week-3.1

2303A51018

Batch-28

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.

Question 1: Zero-Shot Prompting – Palindrome Number Program

Zero-Shot Prompt Used

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

The function should return True if the number is a palindrome, otherwise False.

```
def is_palindrome(num):
```

```
    original = num
```

```
    reverse = 0
```

```
    while num > 0:
```

```
        digit = num % 10
```

```
        reverse = reverse * 10 + digit
```

```
num //= 10  
  
return original == reverse
```

Testing the Code with Multiple Inputs

```
print(is_palindrome(121))  
print(is_palindrome(123))  
print(is_palindrome(0))  
print(is_palindrome(1221))  
print(is_palindrome(10))
```

Output

```
True  
False  
True  
True  
False
```

Observations and Edge-Case Analysis

Correct Behavior

- Correctly identifies palindrome numbers such as 121 and 1221
- Correctly identifies non-palindrome numbers such as 123 and 10
- Works for single-digit numbers like 0

Logical Issues / Missing Edge Cases

1. Negative Numbers

- a. Input: -121
- b. Output: False
- c. Reason: The while loop does not run for negative numbers.
- d. This behavior is acceptable, but not explicitly handled.

2. Non-Integer Input

- a. The function assumes the input is always an integer.

- b. Passing strings or floating-point values may cause errors.

3. No Input Validation

- a. The function does not check input type or handle invalid input gracefully.

Conclusion

The zero-shot prompt successfully generated a correct and efficient palindrome-checking function. However, the code lacks explicit handling for negative numbers and invalid inputs. This demonstrates that while zero-shot prompting can produce functional solutions, it may miss edge-case handling and robustness checks, which must be added manually by the developer.

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.

One-Shot Prompt Used

Example:

Input: 5

Output: 120

Now write a Python function that calculates the factorial of a given number.

The function should return the factorial value.

```
def factorial(n):
```

```
result = 1

for i in range(1, n + 1):
    result = result * i

return result
```

Testing the Code with Multiple Inputs

```
print(factorial(5))
print(factorial(0))
print(factorial(1))
print(factorial(7))
```

Output

```
120
1
1
5040
```

Zero-Shot Solution (For Comparison)

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Aspect	Zero-Shot Solution	One-Shot Solution
Approach	Recursive	Iterative
Readability	Moderate	High
Risk of Error	Stack overflow for large values	Safer for large values
Beginner Friendly	Less	More
Use of Example	None	Clear input-output guidance

Improvements Observed Using One-Shot Prompting

1. Clearer Logic

Providing an example helped the AI generate a straightforward loop-based solution that is easier to understand.

2. Improved Correctness

The iterative approach avoids recursion depth issues and works reliably for larger inputs.

3. Better Edge-Case Handling

The one-shot solution correctly handles 0 and 1 without additional conditions.

4. More Practical Code

The generated function is suitable for real-world use and beginner-level understanding.

Conclusion

One-shot prompting improves the quality of AI-generated code by providing clear expectations through examples. Compared to zero-shot prompting, the one-shot approach produced a more readable, reliable, and beginner-friendly factorial implementation. This demonstrates how even a single example can significantly enhance clarity and correctness in AI-generated programs.

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.

Example:

Input: 5

Output: 120

Expected AI-Generated Code

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

2 Zero-Shot Prompt

Possible AI-Generated Code

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

3 Comparison: One-Shot vs Zero-Shot

Aspect	One-Shot Prompting	Zero-Shot Prompting
Guidance to AI	Clear (example given)	General instruction only
Code clarity	More predictable and simple	May vary (recursive or iterative)
Correctness	High (example confirms expected behavior)	Usually correct, but interpretation may differ
Risk of ambiguity	Very low	Slightly higher
Best for beginners	Yes	Depends on AI output

4 Improvements Observed with One-Shot Prompting

- ✓ Clear expectation of output (5 → 120)
- ✓ Reduces ambiguity about how factorial should behave

- ✓ Often avoids unnecessary recursion
- ✓ Easier to read and understand for learners
- ✓ More consistent and reliable code generation

Conclusion

One-shot prompting improves **clarity, correctness, and consistency** by guiding the AI with an example, whereas zero-shot prompting relies entirely on the AI's interpretation of the task.

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.

1 Few-Shot Prompt for Armstrong Number Check

```
def is_armstrong(num): if num < 0: return "Invalid Input"

digits = str(num)
power = len(digits)
total = 0

for d in digits:
    total += int(d) ** power

if total == num:
```

```
        return "Armstrong Number"
else:
    return "Not an Armstrong Number"
You are an AI that writes Python functions.
```

Example 1:

Input: 153

Output: Armstrong Number

Example 2:

Input: 370

Output: Armstrong Number

Example 3:

Input: 123

Output: Not an Armstrong Number

Now, write a Python function that checks whether a given number is an Armstrong number.

The function should return:

"Armstrong Number" or "Not an Armstrong Number"

3 How Multiple Examples Influence Code Structure & Accuracy

❖ Structural Influence

- AI learns to **extract digits**
- Uses **number of digits as power**
- Applies **loop-based summation**

- Returns **human-readable strings**, not booleans

❖ Accuracy Improvement

- Reduces ambiguity (no guessing output style)
- Handles known Armstrong cases correctly
- Avoids incorrect assumptions (like fixed cube power)

👉 Without multiple examples, the AI might:

- Hard-code logic for 3-digit numbers
- Return True/False instead of required text
- Miss edge cases

4 Testing with Boundary Values

```
print(is_armstrong(0))      # Armstrong Number
print(is_armstrong(1))      # Armstrong Number
print(is_armstrong(9))      # Armstrong Number
print(is_armstrong(10))     # Not an Armstrong Number
print(is_armstrong(9474))   # Armstrong Number
```

✓ Explanation

- Single-digit numbers are Armstrong numbers
- 0 is valid ($0^n = 0$)
- 9474 confirms multi-digit support

5 Testing with Invalid Inputs

```
print(is_armstrong(-153))  # Invalid Input
```

👉 Negative numbers are **not Armstrong numbers**, so input validation improves robustness.

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.

1 Context-Managed Prompt Design

Context-managed prompting means we **explicitly define the role, rules, constraints, and output expectations** so the AI doesn't improvise inefficient or incorrect logic.

🛠️ Context-Managed Prompt Example

You are an AI Python developer.

Context:

You must write an optimized Python program to classify a given integer as Prime, Composite, or Neither.

Instructions & Constraints:

- Accept integer input from the user
- Validate input (handle negative numbers and zero)
- Use an efficient approach (check divisors only up to \sqrt{n})
- Do not use unnecessary loops
- Print exactly one of the following outputs:
"Prime"
"Composite"
"Neither"

The program should be clear, readable, and efficient

This context removes ambiguity and forces correct + optimized logic.

2 Optimized Python Program Generated

```
num = int(input("Enter a number: "))

if num <= 1:
    print("Neither")
else:
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break

    if is_prime:
        print("Prime")
    else:
        print("Composite")
```

3 Input Validation Explanation

Input	Output	Reason
-5	Neither	Negative numbers are neither
0	Neither	Zero is neither
1	Neither	One is neither prime nor composite
2	Prime	Only two factors
9	Composite	More than two factors

- ✓ Invalid or edge values are handled **before** prime checking.

4 Optimization Techniques Used

- Loop runs **only up to \sqrt{n}**
- Early termination using **break**
- No redundant checks
- Avoids checking divisibility beyond necessity

- ⌚ Time Complexity: $O(\sqrt{n})$
- 🧠 Space Complexity: $O(1)$

5 Comparison with Earlier Prompting Strategies

Prompting Type	Characteristics	Result
Zero-Shot	Minimal instruction	Often inefficient ($O(n)$)
Few-Shot	Pattern-based	Correct but may lack optimization
Context-Managed	Clear rules + constraints	Correct, efficient, and robust
		✓

⚡ Key Difference

Context-managed prompting:

- Prevents incorrect assumptions
- Enforces optimization
- Produces predictable, exam-ready output

✓ Final Summary

- Context-managed prompting gives **full control** over AI output
- Ensures proper validation and optimized logic
- Produces superior results compared to zero-shot and few-shot prompting
- Ideal for **production-quality** and **academic 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.

1 Zero-Shot Prompt (No Examples Given)

A zero-shot prompt contains **only instructions**, no sample inputs or outputs.

❖ Zero-Shot Prompt Example

Write a Python function that checks whether a given number is a perfect number.
The function should return an appropriate message indicating whether the number
is a perfect number or not.

→ No hints about:

- Edge cases
- Efficiency
- Output format
- Input validation

This tests how well the AI reasons **without guidance**.

2 AI-Generated Python Code (Typical Zero-Shot Output)

```
def is_perfect(num):  
    total = 0  
    for i in range(1, num):  
        if num % i == 0:
```

```

        total += i

    if total == num:
        return "Perfect Number"
    else:
        return "Not a Perfect Number"

```

3 Testing the Program with Multiple Inputs

```

print(is_perfect(6))      # Perfect Number
print(is_perfect(28))      # Perfect Number
print(is_perfect(12))      # Not a Perfect Number
print(is_perfect(1))       # Not a Perfect Number
print(is_perfect(0))       # Not a Perfect Number
print(is_perfect(-6))      # Not a Perfect Number

```

Q Observations

- Correct for common perfect numbers (6, 28)
- Incorrect handling is *silent* for invalid inputs
- No input validation

4 Missing Conditions in Zero-Shot Output

X Input Validation Missing

- Negative numbers should be rejected
- Zero should be rejected
- 1 should be handled explicitly

X Logical Inefficiencies

- Loop runs from 1 to num-1 → $O(n)$ time
- Checks unnecessary divisors beyond \sqrt{n}
- No early termination

5 Efficiency Issues

Current Complexity

- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

What's Missing

- Optimization using divisor pairs
- Boundary condition checks
- Explicit invalid input handling

6 Improved Version (For Comparison Only)

⚠️ Not part of zero-shot output – shown for analysis purposes.

```
def is_perfect(num):  
    if num <= 1:  
        return "Not a Perfect Number"  
  
    total = 1  
    for i in range(2, int(num ** 0.5) + 1):  
        if num % i == 0:  
            total += i  
            if i != num // i:  
                total += num // i  
  
    if total == num:  
        return "Perfect Number"  
    else:  
        return "Not a Perfect Number"
```

7 Final Comparison Insight

Aspect	Zero-Shot Result
--------	------------------

Correctness	Partial
Efficiency	Poor
Validation	Missing
Reliability	Low

Key Takeaway

Zero-shot prompting often:

- Produces **basic but incomplete logic**
- Misses edge cases
- Uses inefficient algorithms

Final Summary (Exam-Ready)

- Zero-shot prompting uses **only instructions**
- AI-generated code is often **correct but inefficient**
- Missing validation and optimization are common
- Strong prompting strategies improve output quality significantly

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

1 Few-Shot Prompt (With Input-Output Examples)

Few-shot prompting teaches the AI *what to do and how to respond* by showing multiple examples.

❖ Few-Shot Prompt Example

You are an AI that writes Python programs.

Example 1:

Input: 8

Output: Even

Example 2:

Input: 15

Output: Odd

Example 3:

Input: 0

Output: Even

Now write a Python program that:

- Accepts a number as input
- Validates the input
- Prints whether the number is Even or Odd
- Handles invalid (non-integer) inputs gracefully

→ These examples define:

- Expected output words (“Even”, “Odd”)
- Handling of boundary value (0)
- Classification logic

2 Python Program Generated Using Few-Shot Prompting

```
try:  
    num = int(input("Enter a number: "))  
  
    if num % 2 == 0:  
        print("Even")  
    else:  
        print("Odd")  
  
except ValueError:  
    print("Invalid Input")
```

3 How Examples Improve Input Handling & Output Clarity

◊ Input Handling Improvements

- AI learns input should be numeric
- Adds exception handling for non-integer values
- Correctly processes zero as even

◊ Output Clarity Improvements

- Uses **exact words** shown in examples
- Avoids ambiguous outputs like True/False
- Ensures consistent formatting

❖ Without examples, AI might:

- Ignore validation
- Print verbose messages
- Misclassify 0

4 Testing with Negative Numbers

```
print(-4)  # Even  
print(-7)  # Odd
```

- ✓ Negative integers are handled correctly because modulo logic still applies.

5 Testing with Non-Integer Inputs

Input	Output
"abc"	Invalid Input
"4.5"	Invalid Input
""	Invalid Input

- ✓ The program safely handles invalid inputs using try-except.

6 Final Analysis

Aspect	Few-Shot Prompting Effect
Output format	Clear and consistent
Input validation	Automatically included
Edge cases	Correctly handled
Reliability	High

7 Final Summary (Exam-Friendly)

- Few-shot prompting improves accuracy and robustness

- Multiple examples guide:
 - Input expectations
 - Output wording
- Results in cleaner, user-friendly programs
- Superior to zero-shot prompting for validation-based tasks