

### Lab Assignment # 03

Course Title : **AI Assistant Coding**  
Name of Student : **R Govardhan Sai Ganesh**  
Enrollment No. : **2303A54044**  
Batch No. : **48**

### Lab 3: Prompt Engineering – Improving Prompts and Context Management

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

❖ Scenario: 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 used:

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

#### • Screenshot of Generated Code:

The screenshot shows the GitHub Copilot interface. On the left, a code editor displays a Python file named `palindrome.py` with the following content:

```
palindrome.py < ...>
def is_palindrome(n):
    """Check whether a given number is a palindrome."""
    s = str(abs(n))
    return s == s[::-1]

if __name__ == "__main__":
    test_cases = [121, 123, 0, -121, 1001, 10, 99, 12321]
    for num in test_cases:
        result = is_palindrome(num)
        print(f"{num}: {result}")
```

On the right, the AI interface shows the generated prompt and the resulting code. The prompt is:

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

The generated code is identical to the one in the code editor. Below the code, there are several status indicators and notes:

- Enable Claude Haiku 4.5 for all clients
- Selected "Enable"
- Generated Python program for palindrome check
- I'll create a Python program that checks whether a given number is a palindrome;
- Created `palindrome.py`
- Done! I've created a Python program that:
- is\_palindrome(n)** - Converts the number to a string (using absolute value to handle negatives) and checks if it equals its reverse
- Test cases** - Demonstrates the function with various numbers (121, 123, 0, -121, 1001, 10, 99, 12321)

At the bottom, it says "Run it with: `python palindrome.py`".

## • Inputs & Outputs:

<b>Number</b>	<b>Result</b>	<b>Reason</b>
121	✓ True	Reads same forwards and backwards
123	✗ False	Not a palindrome
0	✓ True	Single digit
-121	✓ True	Ignores negative sign
1001	✓ True	Symmetrical
10	✗ False	Not a palindrome
99	✓ True	Repeating digits
12321	✓ True	Symmetrical

#### • Short Explanation of the Code:

The code checks if a number is the same when read it forwards or backwards. It turns the number into text, flips it, and compares both versions. If they match, it's a palindrome. The program tests this with example numbers like 121 (palindrome) and 123 (not a palindrome).

## Task 2: One-Shot Prompting (Factorial Calculation)

- ❖ Scenario: 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 used:**

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

- **Screenshot of Generated Code:**

The screenshot shows a VS Code interface with two tabs open: "palindrome.py" and "factorial.py". The "factorial.py" tab contains the following code:

```
1  def factorial(n):
2      """Compute the factorial of a given number."""
3      if n < 0:
4          raise ValueError("Factorial is not defined for negative numbers")
5      if n == 0 or n == 1:
6          return 1
7      result = 1
8      for i in range(2, n + 1):
9          result *= i
10     return result
11
12
13 if __name__ == "__main__":
14     # Test with input 5, expecting output 120
15     print(f"Factorial of 5: {factorial(5)}")
```

Below the code editor, the terminal window shows the command "python3.13 factorial.py" and its output "Factorial of 5: 120".

On the right side, there is an AI Assistant panel titled "PYTHON FUNCTION FOR COMPUTING FACTORIAL". It includes a "CHAT" section with a message from the AI asking to generate a Python function to compute the factorial of a given number where the input is 5 and the output is 120. Below this, there is a "Read factorial.py file" section with a note that the file has been created, and a detailed description of the factorial function's behavior for non-negative integers.

- **Comparison of Code:**

<b>Zero-Shot Code</b>	<b>One-Shot Code</b>
<pre>def factorial(n):     if n &lt;= 1:         return 1     return n * factorial(n - 1)</pre>	<pre>def factorial(n):     """Compute the factorial of a given number."""     if n &lt; 0:         raise ValueError("Factorial is not defined for negative numbers")     if n == 0 or n == 1:         return 1     result = 1     for i in range(2, n + 1):         result *= i     return result</pre>

- **Comparison Analysis of Code:**

Aspect	Generated	Zero-Shot	Winner
<b>Clarity</b>	Clear, explicit loop-based approach; easy to follow	Concise but less obvious; requires understanding recursion	Generated ✓
<b>Error Handling</b>	Validates negative inputs with descriptive message	No error handling; silently fails on negatives	Generated ✓

- **Key Improvements of Code:**

1. **Error Handling:** Validates negative inputs rather than producing incorrect results
2. **Documentation:** Includes docstring for clarity and IDE support
3. **Efficiency:** Uses iteration instead of recursion, avoiding stack overflow for large inputs
4. **Edge Case Handling:** Explicitly manages 0 and 1 as base cases
5. **Maintainability:** Straightforward loop logic easier to understand than recursive approach
6. **Test Coverage:** Includes a test case demonstrating the expected input/output

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

❖ *Scenario: 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 used:**

generate a Python function to check whether a given number is an Armstrong number by considering the given inputs and outputs

- **Screenshot of Generated Code:**

The screenshot shows a VS Code interface with multiple tabs open. The active tab is 'Armstrong.py' which contains Python code for checking if a number is Armstrong. The code has been simplified by removing unnecessary parts like docstrings and comments. A sidebar on the right shows a 'checkmarks' section with a note about simplifying the code. Below the code editor, the terminal shows the command run and the output for three test cases: 153, 370, and 123. The status bar at the bottom indicates the file is 'Armstrong.py'.

```
palindrome.py X factorial.py Armstrong.py X CHAT + × ☰ ...
```

```
Armstrong.py > ...
1 def is_armstrong(n):
2     digits = str(n)
3     num_digits = len(digits)
4     armstrong_sum = sum(int(digit) ** num_digits for digit in digits)
5     return armstrong_sum == n
6
7
8 if __name__ == "__main__":
9     test_cases = [153, 370, 123]
10
11     for num in test_cases:
12         if is_armstrong(num):
13             print(f"Input: {num} → Output: Armstrong Number")
14         else:
15             print(f"Input: {num} → Output: Not an Armstrong Number")
```

```
← PYTHON FUNCTION FOR COMPUTING FACTORIAL
checkmarks
give simple code
```

- Simplified Armstrong number checker implementation
- Edited Armstrong.py +9 -28

Done! I've simplified the code by removing:

- Docstring and comments
- Negative number handling
- Test case validation
- Extra variables

The function now has just the essential logic to check Armstrong numbers.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\rgsga\OneDrive\Desktop\AI Assistant> & C:/Users/rgsga/AppData/Local/Microsoft/WindowsApps/python3.13.exe "C:/Users/rgsga/OneDrive/Desktop/AI Assistant/Armstrong.py"
```

- Input: 153 → Output: Armstrong Number
- Input: 370 → Output: Armstrong Number
- Input: 123 → Output: Not an Armstrong Number

PS C:\Users\rgsga\OneDrive\Desktop\AI Assistant>

```
Python + × ☰ ... | ☰ ×
```

```
Armstrong.py
```

Describe what to build next

Agent Auto ☰

- **Short Explanation of the Code:**

Multiple examples drive **generalized solutions** rather than hard-coded ones. They expose edge cases (zeros in 370, rejection logic in 123) that single examples miss, ensuring the algorithm works universally. This diversity validates correctness early and produces more robust, portable code.

- Function with boundary values and invalid inputs of the Code:

- ✓ Zero (edge case):  $0 \rightarrow \text{True}$
  - ✓ Single digit:  $1 \rightarrow \text{True}$
  - ✓ Single digit boundary:  $9 \rightarrow \text{True}$
  - ✓ Two digits:  $10 \rightarrow \text{False}$
  - ✓ Valid Armstrong:  $153 \rightarrow \text{True}$
  - ✓ Valid Armstrong with 0:  $370 \rightarrow \text{True}$
  - ✓ Not Armstrong:  $123 \rightarrow \text{False}$
  - ✓ 4-digit Armstrong:  $9474 \rightarrow \text{True}$
  - ✓ Negative number:  $-153 \rightarrow \text{None}$
  - ✓ Float input:  $3.14 \rightarrow \text{None}$
  - ✓ String input:  $153 \rightarrow \text{None}$

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

- ❖ *Scenario: 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 used:**

Generate an optimized Python program that classifies an integer as "**Prime**", "**Composite**", or "**Neither**". Constraints: check divisibility only up to  $\sqrt{n}$ , skip even numbers after 2, handle edge cases ( $n \leq 1 \rightarrow$  Neither,  $n=2 \rightarrow$  Prime). Input via `input()`, output must be one of the three strings.

- **Screenshot of Generated Code:**

The screenshot shows a code editor interface with several tabs at the top: palindrome.py, factorial.py, Armstrong.py, and prime.py (which is the active tab). The code in prime.py is as follows:

```

1 import math
2
3 n = int(input())
4
5 if n <= 1:
6     print("Neither")
7 elif n == 2:
8     print("Prime")
9 elif n % 2 == 0:
10    print("Composite")
11 else:
12     is_prime = True
13     sqrt_n = int(math.sqrt(n))
14     for i in range(3, sqrt_n + 1, 2):
15         if n % i == 0:
16             is_prime = False
17             break
18
19     print("Prime" if is_prime else "Composite")
20

```

To the right of the code, there is a large callout box with the title "OPTIMIZED PYTHON INTEGER CLASSIFICATION PROGRAM". It contains the following text:

numbers after 2, handle edge cases ( $n \leq 1 \rightarrow$  Neither,  $n=2 \rightarrow$  Prime).  
Input via `input()`, output must be one of the three strings.

Below this, a list of optimizations is provided:

- ✓ Read prime.py file
- I'll create an optimized prime classification program with all the specified constraints:
- ✓ Created `prime.py`
- Done! The program implements all the optimizations:
- **Edge cases:**  $n \leq 1$  outputs "Neither",  $n = 2$  outputs "Prime"
- **Fast even check:** Immediately returns "Composite" for even numbers > 2
- **Sqrt optimization:** Only checks divisors up to  $\sqrt{n}$
- **Skip even divisors:** Iterates through odd numbers only (3, 5, 7, ...)
- **Early termination:** Breaks immediately when a divisor is found

Test it with examples like:

- Input 1 → "Neither"
- Input 2 → "Prime"
- Input 4 → "Composite"
- Input 17 → "Prime"

At the bottom of the interface, there is a terminal window showing command-line interactions with the generated program:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + | X
PS C:\Users\rgsga\Desktop\AI Assistant> & c:/Users/rgsga/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/rgsga/Desktop/AI Assistant/prime.py"
1
Neither
PS C:\Users\rgsga\Desktop\AI Assistant> & c:/Users/rgsga/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/rgsga/Desktop/AI Assistant/prime.py"
4
Composite
PS C:\Users\rgsga\Desktop\AI Assistant> & c:/Users/rgsga/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/rgsga/Desktop/AI Assistant/prime.py"
3

```

- **Short Explanation of the Code:**

The program takes a number and checks if it's Prime, Composite, or Neither. Numbers 1 or less are "Neither", 2 is "Prime", and even numbers are "Composite". For odd numbers, it checks divisibility only up to the square root of the number, testing only odd divisors. If it finds any divisor, it's "Composite"; otherwise, it's "Prime". This is fast because it skips unnecessary checks and stops early when a divisor is found.

- **Prompting Strategies - Comparison:**

Strategy	Result	Efficiency
<b>Zero-Shot</b> (constraints only)	Optimized, all constraints applied	Best - 19 lines
<b>One-Shot</b> (1 example)	Over-simplified, missing optimizations	Poor - 30+ lines
<b>Few-Shot</b> (multiple examples)	Verbose, defensive coding	Okay - 40+ lines

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

❖ Scenario: Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is perfect number.

### Task:

- Record the AI-generated code.
- Test the program with multiple inputs.
- Identify any missing conditions or inefficiencies in the logic.

### • Prompt used:

Write a zero-shot prompt to generate a Python function that checks whether a given number is a perfect number and test the program with multiple inputs. and identify any missing conditions or inefficiencies in the logic or not.

### • Screenshot of Generated Code:

The screenshot shows a code editor interface with several tabs at the top: palindrone.py, factorial.py, Armstrong.py, prime.py, and perfect.py (the active tab). The code in perfect.py is as follows:

```
perfect.py > ...
1  """
2  Zero-Shot Prompt:
3  "Write a Python function that determines if a given number is a perfect number.
4  A perfect number is a positive integer that is equal to the sum of its proper
5  positive divisors (all divisors excluding the number itself). Include input
6  validation and handle edge cases. The function should return True if the number
7  is perfect, False otherwise. Also return the list of divisors for verification."
8  """
9
10 def is_perfect_number(n):
11     """
12     Check if a number is a perfect number.
13
14     A perfect number is a positive integer equal to the sum of its proper
15     positive divisors (excluding the number itself).
16
17     Args:
18         n: Integer to check
19
20     Returns:
21         Tuple of (is_perfect: bool, divisors: list)
22     """
23     # Input validation
24     if not isinstance(n, int):
25         raise TypeError("Input must be an integer")
...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

The terminal below shows the command: PS C:\Users\rgsga\OneDrive\Desktop\AI Assistant> cd "c:\Users\rgsga\OneDrive\Desktop\AI Assistant" ; python perfect.py. The output indicates "CORRECTNESS VERIFIED" and lists test cases: ✓ Perfect numbers: 6, 28, 496, 8128 correctly identified; ✓ Edge cases: 0, 1, negatives correctly rejected; ✓ Non-perfect numbers: Correctly identified; ✓ Abundant numbers: Correctly rejected; ✓ Divisor calculation: Accurate and verified.

The right side of the interface displays analysis results:

- Comprehensive Testing:** ✓ All 4 known perfect numbers (6, 28, 496, 8128); ✓ Edge cases (0, 1, negative numbers); ✓ Prime numbers; ✓ Abundant numbers; ✓ 100% test pass rate.
- Missing Conditions & Inefficiencies Identified:**
  - Handled:
    - Type validation (integers only)
    - Boundary checks (negative/zero rejected)
    - Proper divisor calculation
  - Inefficiencies Found:
    - Brute force checks all n-1 numbers unnecessarily
    - For n=8128: reduced from 8127 iterations to ~90 with optimization
  - Further Optimizations Possible:
    - Euclid-Euler theorem lookup (perfect numbers =  $2^{p-1} \times (2^p - 1)$  where p and  $2^p - 1$  are prime) - only 51 known perfect numbers exist
- The code is production-ready with proper documentation, error handling, and performance analysis!

Bottom right: Agent ▾ Auto ▾ 🌐 ↗

### • Short Explanation of the Code:

This Python program contains two functions to check if a number is perfect (equal to the sum of its proper divisors). The `is_perfect_number()` function uses a brute-force approach, checking all numbers from 1 to  $n-1$ , while `is_perfect_number_optimized()` improves efficiency by only checking divisors up to  $\sqrt{n}$  using divisor pairs, achieving  $\sim 32x$  speedup. Both return a tuple with a boolean result and the list of divisors. The code includes comprehensive testing with 12 test cases covering perfect numbers (6, 28, 496, 8128), edge cases (0, 1, negatives), and non-perfect numbers, with all tests passing. Input validation ensures type checking, and the analysis section identifies that all critical conditions are properly handled, with the main inefficiency being the  $O(n)$  complexity of the brute-force version versus the  $O(\sqrt{n})$  optimized version.

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

❖ *Scenario: 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 used:

Write a Python program that checks if a number is even or odd with input validation. Test with positive, negative, zero, and non-integer inputs.

- **Screenshot of Generated Code:**

factorial.py Armstrong.py prime.py perfect.py fe.py even\_odd.py CHAT

```
even_odd.py > check_even_odd
26 def get_user_input():
27     try:
28         user_input = int(input("Enter a number: "))
29     except ValueError:
30         print(f"Error: '{user_input}' is not a valid integer. Please try again.")
31
32 def main():
33     """Main function to run the program."""
34     print("=" * 40)
35     print("Even or Odd Number Checker")
36     print("=" * 40)
37
38     while True:
39         try:
40             number = get_user_input()
41             result = check_even_odd(number)
42             print(f"Input: {number} → Output: {result}\n")
43         except TypeError as e:
44             print(f"Error: {e}\n")
45         except KeyboardInterrupt:
46             print("\n\nProgram terminated by user.")
47             break
48
49 def run_tests():
50     """Run test cases to verify the program works correctly."""
51
52 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + x
PS C:\Users\rgsga\OneDrive\Desktop\AI Assistant> & C:/Users/rgsga/AppData/Local/Microsoft/WindowsApps/python3.13.exe "C:/Users/rgsga/OneDrive/Desktop/AI Assistant/even_odd.py"
PASS: multiple values '10 20' - Correctly rejected
=====
Starting interactive mode...
=====
Even or Odd Number Checker
=====
```

← EVEN OR ODD NUMBER CHECKER IN PYTHON

input validation.

Examples:

- Input: 8 → Output: Even
- Input: 15 → Output: Odd
- Input: 0 → Output: Even and Test the program with negative numbers and non-integer inputs.

✓ Read lines 1 to 50 of file

Let me read the rest of the file:

✓ Read even\_odd.py, lines 51 to 116

Great! The file already has a comprehensive implementation. Let me run the program to verify it works correctly with all the test cases:

Run pwsh command?

cd "C:\Users\rgsga\OneDrive\Desktop\AI Assistant"; python even\_odd.py << "8 n15 n0 n-5 nquit"

Allow Skip

even\_odd.py

Describe what to build next

Agent Auto

#### • Short Explanation of the Code:

The program includes a robust automated test suite through the `run_tests()` function that covers all important scenarios: positive numbers like 8, 15, and 100; zero (which correctly returns "Even"); negative numbers including -4, -7, -1, and -100; and invalid inputs such as floats, strings, and empty strings. Each test displays a clear ✓ PASS or ✗ FAIL status to verify correctness.