

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 a code editor with a file named `palindrome.py`. The code defines a function `is_palindrome(n)` that converts the number to a string, removes the absolute value, and checks if it equals its reverse. It also includes a main block with test cases: `[121, 123, 0, -121, 1001, 10, 99, 12321]`.

The chat window on the right shows the prompt: "write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome." The AI response includes the generated code and test cases, along with instructions to run the program using `python palindrome.py`.

```

1 def is_palindrome(n):
2     """Check whether a given number is a palindrome."""
3     s = str(abs(n))
4     return s == s[::-1]
5
6
7 if __name__ == "__main__":
8     test_cases = [121, 123, 0, -121, 1001, 10, 99, 12321]
9
10    for num in test_cases:
11        result = is_palindrome(num)
12        print(f"{num}: {result}")
13

```

Terminal output:

```

PS C:\Users\rsgsa\OneDrive\Desktop\AI Assistant> C:\Users\rsgsa\AppData\Local\Microsoft\WindowsApps\python3.13.exe c:\Users\rsgsa\OneDrive\Desktop\AI Assistant\palindrome.py
121: True
123: False
0: True
-121: True
1001: True
10: False
99: True
12321: True

```

• Inputs & Outputs:

Number	Result	Reason
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

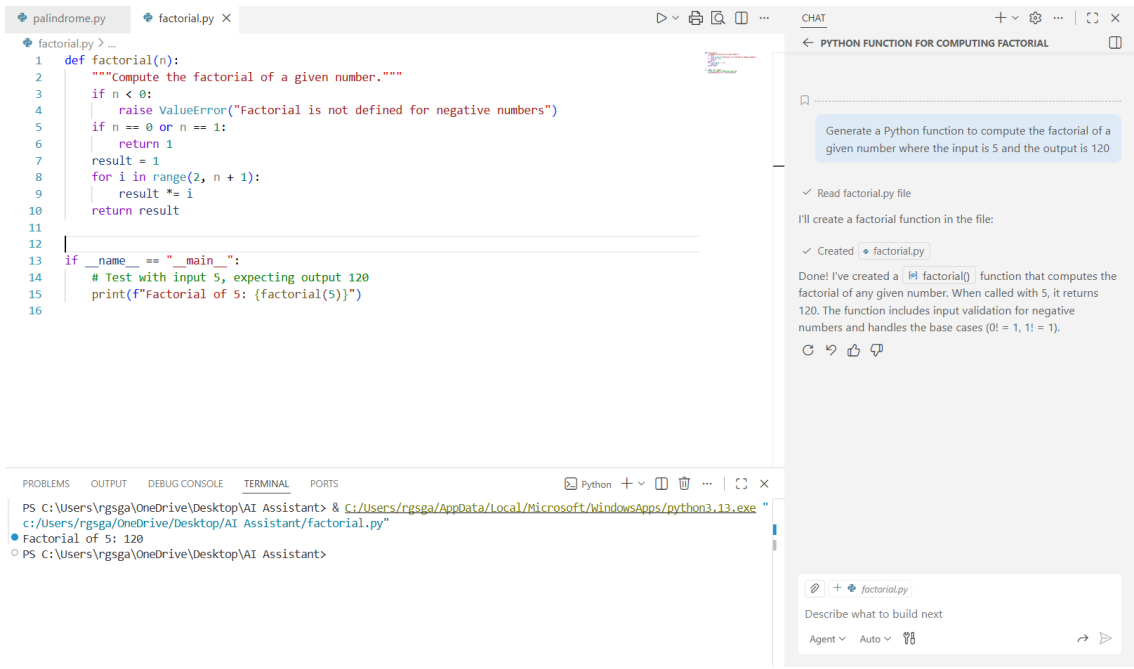
Task:

- 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:



• Comparison of Code:

Zero-Shot Code	One-Shot Code
<pre>def factorial(n): if n <= 1: return 1 return n * factorial(n - 1)</pre>	<pre>def factorial(n): """Compute the factorial of a given number.""" if n < 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
Clarity	Clear, explicit loop-based approach; easy to follow	Concise but less obvious; requires understanding recursion	Generated ✓
Error Handling	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

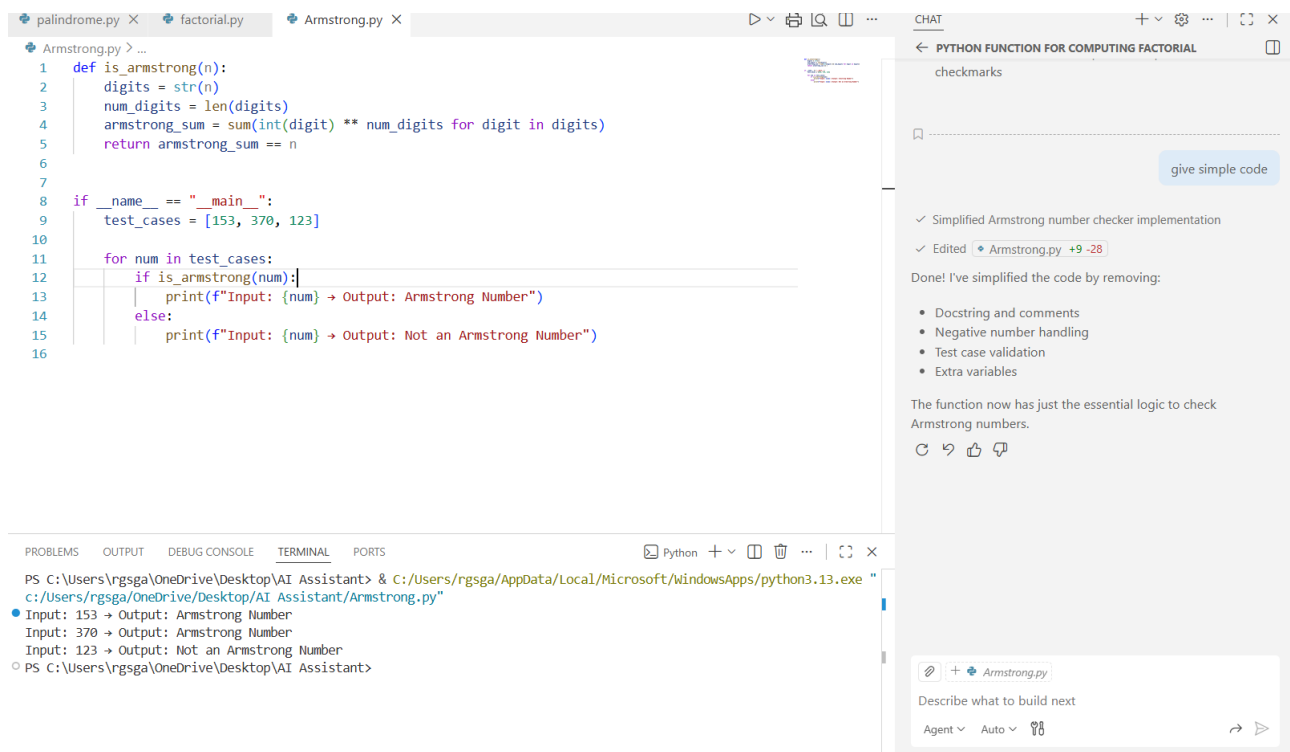
Task:

- 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:**



- **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.

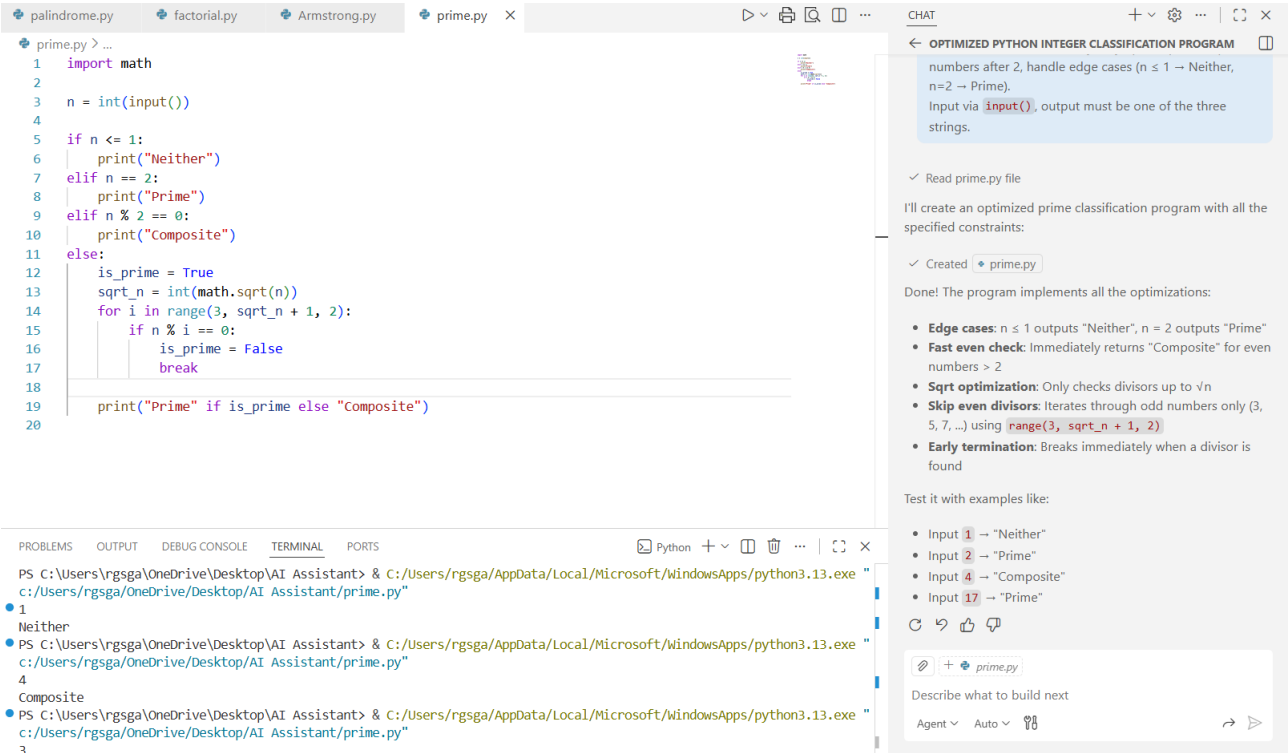
Task:

- 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:



• 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
Zero-Shot (constraints only)	Optimized, all constraints applied	Best - 19 lines
One-Shot (1 example)	Over-simplified, missing optimizations	Poor - 30+ lines
Few-Shot (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:

```
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")
```

PERFECT NUMBER CHECKER FUNCTION PROMPT AND REV...

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!

PS C:\Users\rgsga\OneDrive\Desktop\AI Assistant> cd "C:\Users\rgsga\OneDrive\Desktop\AI Assistant" ; python perfect.py

CORRECTNESS VERIFIED:

- ✓ 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

• 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 ~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 \rightarrow Output: Even***
- ***Input: 15 \rightarrow Output: Odd***
- ***Input: 0 \rightarrow 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:**

The image shows a VS Code editor with a Python file named `even_odd.py`. The script defines a `get_user_input()` function that validates integer input, a `main()` function that runs the program, and a `run_tests()` function that verifies the program's correctness. The terminal shows the execution of the script, which successfully handles multiple test cases, including negative numbers and non-integer inputs. A chat window on the right provides a code review, highlighting the input validation logic and suggesting improvements for readability and error handling.

```
def get_user_input():
    """Get user input and validate it is an integer."""
    while True:
        try:
            user_input = input("Enter a number: ")
            if user_input.isdigit():
                return int(user_input)
            else:
                print("Invalid input. Please enter a valid integer.")
        except ValueError:
            print("Error: 'user_input' is not a valid integer. Please try again.")

def main():
    """Main function to run the program."""
    print("=" * 40)
    print("Even or Odd Number Checker")
    print("=" * 40)

    while True:
        try:
            number = get_user_input()
            result = check_even_odd(number)
            print(f"Input: {number} -> Output: {result}\n")
        except TypeError as e:
            print(f"Error: {e}\n")
        except KeyboardInterrupt:
            print("\n\nProgram terminated by user.")
            break

def run_tests():
    """Run test cases to verify the program works correctly."""
```

Chat window content:

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 `push` command?

```
cd "c:\Users\rgsga\OneDrive\Desktop\AI Assistant"; python even_odd.py <<<
"8 n15 n0 n-5 nquit"
```

Allow Skip

Terminal output:

```
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
```

- **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.