

ASSIGNMENT - 3.5

Name: M. Sprusheeth Rao

Roll Number : 2303A51206

Batch - 04

Question 1: Zero-Shot Prompting (Leap Year Check)

Write a zero-shot prompt to generate a Python function that checks whether a given year is a leap year.

Week2 - Task:

- Record the AI-generated code.
- Test with years like 1900, 2000, 2024.
- Identify logical flaws or missing conditions.

The screenshot shows a code editor interface with multiple tabs open. The active tab contains the following Python code:

```
1  # generate a python function that checks whether a given year is leap year or not
2  def is_leap_year(year):
3      """check if a given year is a leap year.
4
5      A year is a leap year if it is divisible by 4,
6      except for end-of-century years, which must be divisible by 400.
7
8      Args:
9          year (int): The year to check.
10     Returns:
11         bool: True if the year is a leap year, False otherwise.
12     """
13     if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
14         return True
15     else:
16         return False
17 # Example usage:
18 year = 1900
19 if is_leap_year(year):
20     print(f"[year] is a leap year.")
21 else:
22     print(f"[year] is not a leap year.")
23
```

The code is annotated with docstrings and comments explaining the leap year rules. Below the code editor, the terminal output shows the function being tested with years 2024, 1900, and 1900 again, with the expected results printed.

AI responses may be inaccurate.
Generate Agent Instructions to onboard AI onto your codebase.

Logical Flaws and Missing Conditions:

The zero-shot generated leap year function correctly implements the Gregorian leap year rules and produces correct results for years such as 1900, 2000, and 2024. However, it lacks input validation, as it assumes the year is always a valid integer. The function does not handle negative years, zero, or non-integer inputs, and it includes example print statements instead of returning a clean, reusable function. These missing conditions reduce the robustness of the solution.

Question 2: One-Shot Prompting (GCD of Two Numbers) Write a one-shot prompt with one example to generate a Python function that finds the Greatest Common Divisor (GCD) of two numbers.

Example:

Input: 12, 18 → Output: 6 Task:

- Compare with a zero-shot solution.
- Analyze algorithm efficiency.

The screenshot shows a code editor interface with multiple tabs. The active tab contains the following Python code:

```
1  """
2  num = 12,18
3  gcd = 6
4  """
5  def compute_gcd(a, b):
6      while b:
7          a, b = b, a % b
8      return a
9  if __name__ == "__main__":
10     a = 12
11     b = 18
12     print("num =", (a, b))
13     print("gcd =", compute_gcd(a, b))
```

The code defines a function `compute_gcd` that implements the Euclidean algorithm. It takes two arguments, `a` and `b`, and repeatedly swaps them until `b` is zero, returning the value of `a`. The code then prints the input pair and the calculated GCD. The AI interface includes a "Run Code (Ctrl+Alt+N)" button and a "CHAT" section with a message bubble icon and the text "Ask about your code".

Below the code editor, the terminal output shows the execution of the code:

```
[Running] python -u "C:\Users\naksh\AppData\Local\Temp\tempCodeRunnerFile.py"
num = (12, 18)
gcd = 6
```

The output indicates that the code ran successfully with a duration of 0.427 seconds.

Comparison and Efficiency Analysis:

The one-shot prompted solution produces clearer and more efficient code compared to a zero-shot solution. By providing an example, the AI selects the Euclidean algorithm, which significantly improves performance. The algorithm runs in logarithmic time $O(\log n)$, making it efficient even for large inputs, whereas a zero-shot solution may use a slower brute-force method.

Question 3: Few-Shot Prompting (LCM Calculation)

Write a few-shot prompt with multiple examples to generate a Python function that computes the Least Common Multiple (LCM).

Examples:

- Input: 4, 6 → Output: 12
- Input: 5, 10 → Output: 10
- Input: 7, 3 → Output: 21 Task:
- Examine how examples guide formula selection.
- Test edge cases.

```

File Edit Selection View Go Run Terminal Help ⏮ ⏯ Search
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PERSISTENT Filter (e.g. text, excludeText, t... Code
[Running] python -u "C:\Users\naksh\AppData\Local\Temp\tempCodeRunnerFile.py"
LCM of 4 and 6 is 12
[Done] exited with code=0 in 0.473 seconds

```

The screenshot shows a code editor interface with multiple tabs open. The active tab contains Python code for calculating the Least Common Multiple (LCM) of two numbers, `a` and `b`. The code uses a while loop to increment a variable `greater` until it is divisible by both `a` and `b`. The output panel at the bottom shows the script running and printing the result 'LCM of 4 and 6 is 12'.

Question 4: Zero-Shot Prompting (Binary to Decimal Conversion) Write a zero-shot prompt to generate a Python function that converts a binary number to decimal.

Task:

- Test with valid and invalid binary inputs.
- Identify missing validation logic.

```
1 #generate a python function that converts a binary number to decimal
2 def binary_to_decimal(binary_str):
3     decimal_value = 0
4     binary_str = binary_str[::-1] # Reverse the string to process from least significant bit
5     for index, digit in enumerate(binary_str):
6         if digit == '1':
7             decimal_value += 2 ** index
8     return decimal_value
9
10 # Example usage:
11 binary_number = "1101"
12 decimal_number = binary_to_decimal(binary_number)
13 print(f"The decimal value of binary {binary_number} is {decimal_number}")
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, excludeText, t...) Code ...

[Running] python -u "C:\Users\naksh\AppData\Local\Temp\tempCodeRunnerFile.python"
The decimal value of binary 1101 is 13
[Done] exited with code=0 in 0.383 seconds

Ask about your code
AI responses may be inaccurate.
Generate Agent Instructions to onboard AI onto your codebase.

Missing Validation Logic:

The zero-shot generated function correctly converts valid binary numbers to decimal but lacks proper input validation. It does not check whether the input contains only binary digits (0 and 1), does not handle empty strings, and fails to raise errors for invalid inputs. As a result, incorrect or non-binary inputs may produce misleading outputs, reducing the robustness of the solution.

Question 5: One-Shot Prompting (Decimal to Binary Conversion)

Write a one-shot prompt with an example to generate a Python function that converts a decimal number to binary.

Example:

Input: 10 → Output: 1010 Task:

- Compare clarity with zero-shot output.
- Analyze handling of zero and negative numbers.

```
1 ...
2 num = 10
3 binary_number = 1010
4 ...
5 def decimal_to_binary(n):
6     if n > 1:
7         decimal_to_binary(n // 2)
8         print(n % 2, end='')
9 num = 10
10 decimal_to_binary(num)
11 |
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, excludeText, t... Code

[Running] python -u "C:\Users\naksh\AppData\Local\Temp\tempCodeRunnerFile.py"
1010
[Done] exited with code=0 in 0.446 seconds

CHAT Ask about your code All responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase.

Comparison of Clarity:

The one-shot prompting output is more understandable than the zero-shot output since the given example explicitly states the desired input-output behavior. This assists the AI in organizing the logic properly, preventing ambiguity, and creating more readable and well-documented code. Conversely, zero-shot prompting tends to produce shorter and less descriptive code without edge-case coverage.

Question 6: Few-Shot Prompting (Harshad Number Check) Write a few-shot prompt to generate a Python function that checks whether a number is a Harshad (Niven) number.

Examples:

- **Input: 18 → Output: Harshad Number**
- **Input: 21 → Output: Harshad Number** • **Input: 19 → Output: Not a Harshad Number**

Number Task:

- **Test boundary conditions.**
- **Evaluate robustness**

The screenshot shows a code editor interface with a dark theme. At the top, there's a menu bar with File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. Below the menu is a tab bar with several tabs: Untitled-4, Untitled-5, Untitled-6, Untitled-7, Untitled-1, Untitled-2, and CHAT. The CHAT tab has a small icon of a speech bubble with a plus sign.

The main area contains Python code:

```
4 num = 21
5 print(num is Harshad number)
6 num = 19
7 print(num is not Harshad number)
8 """
9 def is_harshad_number(num):
10     digit_sum = sum(int(digit) for digit in str(num))
11     return num % digit_sum == 0
12 if __name__ == "__main__":
13     test_numbers = [18, 21, 19]
14     for num in test_numbers:
15         if is_harshad_number(num):
16             print(f"{num} is Harshad number")
17         else:
18             print(f"{num} is not Harshad number")
19 
```

Below the code editor is a terminal window showing the output of the code execution:

```
[Running] python -u "C:\Users\naksh\AppData\Local\Temp\tempcodeRunnerFile.py"
18 is Harshad number
21 is Harshad number
19 is not Harshad number
```

At the bottom right, there's a sidebar with a message: "Ask about your code" and "Generate Agent Instructions to onboard AI onto your codebase." There's also a link to "Explore and understand your code".

Robustness Evaluation:

The Harshad number function is robust if it correctly handles valid inputs, boundary cases such as 0 and single-digit numbers, and prevents division by zero errors. A robust implementation also explicitly handles negative values and restricts inputs to positive integers, ensuring consistent and reliable behavior across all test cases.