

LAB ASSIGNMENT – 2.5

HALLTICKET NO : 2303A51474

BATCH NO : 29

TASK – 01

PROMPT : Refactoring Odd/Even Logic (List Version)

CODE :

The screenshot shows a Jupyter Notebook interface with the file 'Untitled0.ipynb' open. The code cell [4] contains the following Python code:

```
[4] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum = 0
odd_sum = 0

for num in numbers:
    if num % 2 == 0:
        even_sum = even_sum + num
    else:
        odd_sum = odd_sum + num

print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```

The output cell shows the results:

```
... Sum of even numbers: 30
Sum of odd numbers: 25
```

OUTPUT:

The screenshot shows a Jupyter Notebook interface with the file 'Untitled0.ipynb' open. The code cell [5] contains the following Python code:

```
[5] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_sum = sum(num for num in numbers if num % 2 == 0)
odd_sum = sum(num for num in numbers if num % 2 != 0)

print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```

The output cell shows the results:

```
... Sum of even numbers: 30
Sum of odd numbers: 25
```

OBSERVATION:

Gemini and Cursor AI both suggested simplifying the loop-based logic using Python's built-in functions and comprehensions. The refactored code is more readable, concise, and efficient, making it easier for future maintenance.

TASK - 02

PROMPT : Area Calculation Explanation

Double-click (or enter) to edit

```
[6] ✓ Os
def calculate_area(shape, value1, value2=0):
    if shape == "circle":
        return 3.14 * value1 * value1
    elif shape == "rectangle":
        return value1 * value2
    elif shape == "triangle":
        return 0.5 * value1 * value2
    else:
        return "Invalid shape"
```

This `calculate_area` function is designed to compute the area of three different geometric shapes: a circle, a rectangle, and a triangle.

Here's a breakdown of how it works:

- **shape parameter:** This string argument determines which shape's area needs to be calculated. It accepts "circle", "rectangle", or "triangle". If an unsupported shape is provided, it returns "Invalid shape".
- **value1 parameter:** This is the primary value used in the area calculation:
 - For a "circle", it represents the **radius**.
 - For a "rectangle", it represents the **length**.
 - For a "triangle", it represents the **base**.
- **value2 parameter:** This parameter has a default value of `0` and is used for shapes that require a second dimension:
 - For a "circle", it is **not used**.
 - For a "rectangle", it represents the **width**.
 - For a "triangle", it represents the **height**.

How the function calculates the area for each shape:

- **Circle:** If `shape` is "circle", it calculates the area using the formula `$\pi * radius * radius$` (approximating π as 3.14).
- **Rectangle:** If `shape` is "rectangle", it calculates the area using the formula `$length * width$` .
- **Triangle:** If `shape` is "triangle", it calculates the area using the formula `$0.5 * base * height$` .

Example Usage:

```
print(calculate_area("circle", 5))      # Area of a circle with radius 5
print(calculate_area("rectangle", 4, 6)) # Area of a rectangle with length 4, width 6
print(calculate_area("triangle", 10, 5)) # Area of a triangle with base 10, height 5
```

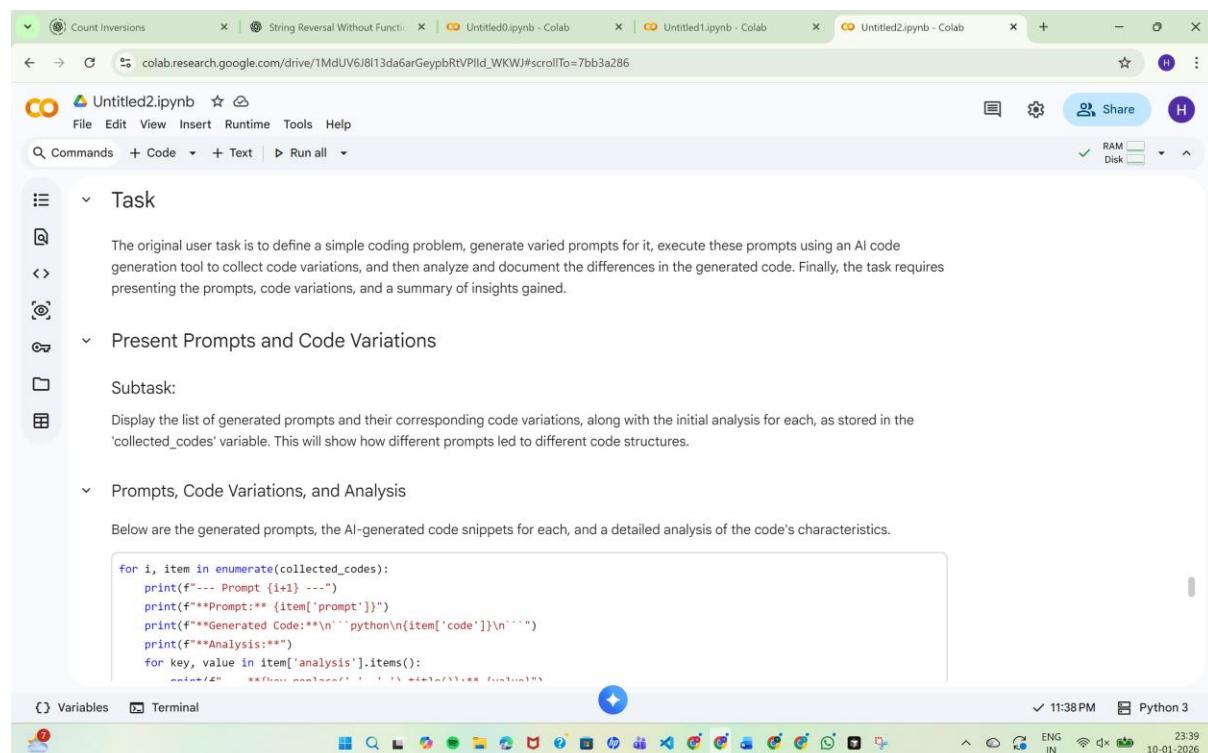
OBSERVATION:

This function calculates the area of different shapes based on the input provided. The parameter shape determines which formula is applied. For a circle, value1 represents the radius. For a rectangle and triangle, value1 and value2 represent dimensions such as length and breadth. The function uses conditional statements to select the correct formula and returns the calculated area.

Gemini provided a **clear and beginner-friendly explanation**, making it suitable for onboarding junior developers and explaining mathematical logic step by step.

TASK – 03

PROMPT: Prompt Sensitivity Experiment (Cursor AI)



The screenshot shows a Google Colab notebook titled "Untitled2.ipynb". The interface includes a top navigation bar with tabs for "Count Inversions", "String Reversal Without Function", "Untitled0.ipynb - Colab", "Untitled1.ipynb - Colab", and "Untitled2.ipynb - Colab". Below the tabs is a toolbar with icons for file operations, sharing, and help. The main workspace contains a collapsible sidebar with sections for "Task", "Present Prompts and Code Variations", and "Prompts, Code Variations, and Analysis". The "Task" section describes the goal of generating varied prompts for a user-defined task. The "Present Prompts and Code Variations" section contains a "Subtask:" entry and a note about displaying generated prompts and their corresponding code variations. The "Prompts, Code Variations, and Analysis" section provides a template for generating prompts and AI-generated code snippets. At the bottom, there are buttons for "Variables" and "Terminal", and a status bar showing the time as 11:38 PM, the language as Python 3, and the date as 10-01-2026.

```
for i, item in enumerate(collected_codes):
    print(f"--- Prompt {i+1} ---")
    print(f"**Prompt:** {item['prompt']}")
    print(f"**Generated Code:**\n{item['code']}\n")
    print(f"**Analysis:**")
    for key, value in item['analysis'].items():
        print(f"  {key}: {value}\n")
```

Count Inversions | String Reversal Without Function | Untitled0.ipynb - Colab | Untitled1.ipynb - Colab | Untitled2.ipynb - Colab

colab.research.google.com/drive/1MdUV6J8l13da6arGeypbRtVPlld_WKwJ#scrollTo=7bb3a286

Prompts, Code Variations, and Analysis

Below are the generated prompts, the AI-generated code snippets for each, and a detailed analysis of the code's characteristics.

```

for i, item in enumerate(collected_codes):
    print(f"--- Prompt {i+1} ---")
    print(f"Prompt: {item['prompt']}")
    print(f"Generated Code:\n```python\n{item['code']}\n```")
    print(f"Analysis:")
    for key, value in item['analysis'].items():
        print(f" - {key.replace('_', ' ').title()}: {value}")
    print("\n")

```

Reasoning: To fulfill the subtask of presenting the prompts, code variations, and analysis, I will execute the Python code to iterate through the `collected_codes` list and print each item's prompt, code, and analysis in a well-formatted manner.

```

for i, item in enumerate(collected_codes):
    print(f"--- Prompt {i+1} ---")
    print(f"Prompt: {item['prompt']}")
    print(f"Generated Code:\n```python\n{item['code']}\n```")
    print(f"Analysis:")
    for key, value in item['analysis'].items():
        print(f" - {key.replace('_', ' ').title()}: {value}")
    print("\n")

```

-- Prompt 1 --
Prompt: Write a Python function to reverse a string using string slicing.

Variables Terminal 11:38 PM Python 3 ENG IN 23:39 10-01-2026

Count Inversions | String Reversal Without Function | Untitled0.ipynb - Colab | Untitled1.ipynb - Colab | Untitled2.ipynb - Colab

colab.research.google.com/drive/1MdUV6J8l13da6arGeypbRtVPlld_WKwJ#scrollTo=4b337de3

Prompts, Code Variations, and Analysis

Below are the generated prompts, the AI-generated code snippets for each, and a detailed analysis of the code's characteristics.

```

    return s[::-1]

```

... Analysis:
- Algorithm Choice: String slicing (Python's built-in feature for reversing sequences).
- Code Structure Readability: Clear function definition and return statement. Generally readable and concise.
- Libraries Built In Functions: Relies on Python's powerful string slicing feature.
- Error Handling Edge Cases: Implicitly handles empty strings and single-character strings due to Python's string behavior (e.g., slicing, iteration). No explicit error handling provided.
- Efficiency Considerations: Highly optimized, typically O(N) time complexity due to C-level implementation of built-in functions. Low overhead.

-- Prompt 2 --
Prompt: Implement a function to reverse a string iteratively.
Generated Code:

```

python
def reverse_string_iterative(s):
    reversed_s = ''
    for char in s:
        reversed_s = char + reversed_s
    return reversed_s

```

Analysis:
- Algorithm Choice: Undetermined or another iterative/manual approach.
- Code Structure Readability: Clear function definition and return statement. Generally readable and concise.
- Libraries Built In Functions: Primarily uses basic iteration, no specific complex built-in functions for reversal.
- Error Handling Edge Cases: Implicitly handles empty strings and single-character strings due to Python's string behavior (e.g., slicing, iteration). No explicit error handling provided.
- Efficiency Considerations: Efficient for practical purposes, often O(N) or similar.

-- Prompt 3 --
Prompt: Reverse a string using the reversed() function and join().
Generated Code:

```

python
def reverse_string_reversed_join(s):
    return ''.join(reversed(s))

```

Variables Terminal 11:38 PM Python 3 ENG IN 23:40 10-01-2026

The screenshot shows a Google Colab notebook titled 'Untitled2.ipynb'. In the top right corner, there are sharing and settings icons. Below the toolbar, there's a 'RAM Disk' indicator. The main area displays an 'Analysis' section with a list of bullet points:

- Algorithm Choice: Using the built-in `reversed()` function (returns an iterator) combined with `join()`.
- Code Structure Readability: Clear function definition and return statement. Generally readable and concise.
- Libraries Built In Functions: Utilizes Python's `reversed()` built-in function and string `join()` method.
- Error Handling Edge Cases: Implicitly handles empty strings and single-character strings due to Python's string behavior (e.g., slicing, iteration). No explicit error handling is present.
- Efficiency Considerations: Highly optimized, typically $O(N)$ time complexity due to C-level implementation of built-in functions. Low overhead.

Present Summary of Insights

Subtask:

Display the comprehensive summary of insights derived from analyzing the code variations. This will include observations on algorithm choice, efficiency, and the impact of prompt specificity.

Present Summary of Insights

Subtask:

Display the comprehensive summary of insights derived from analyzing the code variations. This will include observations on algorithm choice, efficiency, and the impact of prompt specificity.

Summary of Insights from Code Variation Analysis:



The screenshot shows a Google Colab notebook titled 'Untitled2.ipynb'. The main content area contains a section titled 'Based on the analysis of the generated code snippets for reversing a string, here are some key insights:' followed by four numbered sections:

- 1. Prompt Specificity Directly Influences Algorithm Choice:**
 - The prompt "Write a Python function to reverse a string using string slicing." directly led to the most concise and Pythonic solution (`s[::-1]`). This demonstrates the AI's ability to follow explicit instructions for specific methods.
 - Similarly, "Reverse a string using the `reversed()` function and `join()`." resulted in code precisely using those built-in features (`''.join(reversed(s))`).
 - However, the prompt "Implement a function to reverse a string iteratively." generated an $O(N^2)$ iterative solution (`reversed_s = char + reversed_s`) due to repeated string concatenation. This highlights that while the AI can produce iterative solutions, it may not always prioritize the most efficient iterative approach without further guidance (e.g., building a list of characters and then joining, or using a two-pointer swap). The prompt did not specify optimization, only iteration.
- 2. Conciseness vs. Verbosity:**
 - The direct prompts asking for specific methods (`slicing`, `reversed()`) resulted in very concise, often one-liner, function bodies. This showcases Python's expressive power and the AI's understanding of Pythonic idioms.
 - The iterative solution was more verbose, as expected, due to the explicit loop construction.
- 3. Default Handling of Edge Cases:**
 - In all analyzed solutions, edge cases like empty strings or single-character strings were handled implicitly by Python's string behavior (slicing, `reversed()`, iteration). No explicit `if` statements for `len(s) == 0` or `len(s) == 1` were generated, indicating that for simple problems, the AI relies on the language's inherent robustness unless explicitly asked for defensive programming or validation.
- 4. Efficiency and Built-in Functions:**

The screenshot shows a Google Colab notebook titled "Untitled2.ipynb". The notebook contains a section of text with bullet points analyzing Python string reversal solutions. It highlights the use of built-in functions like `reversed()` and `join()`, noting their efficiency due to C-level implementations. It also discusses the time complexity of iterative concatenation ($O(N^2)$) versus optimized solutions ($O(N)$). A section on "Lack of Explicit Readability/Docstring Emphasis" is mentioned. The text concludes with a summary of the AI's capabilities in understanding varying prompt specificity.

Final Task

Subtask:

Conclude the task by confirming that the prompt list, code variations, and summary of insights have been presented.

Variables Terminal 11:38 PM Python 3 ENG IN 10-01-2026

OBSERVATION:

- The process successfully displayed a list of generated prompts, their corresponding AI-generated code variations, and an initial analysis for each.
- The initial analysis covered aspects like algorithm choice (e.g., string slicing, iterative approach, `reversed()` function), code structure readability, use of built-in functions, implicit handling of edge cases (e.g., empty strings), and efficiency considerations.
- The output demonstrated how different prompt wordings led to distinct code structures and algorithmic choices for string reversal (e.g., `s[::-1]` for slicing, loop-based concatenation for iteration, `".join(reversed(s))` for `reversed()` and `join()`).
- A comprehensive summary of insights highlighted that prompt specificity directly influenced the algorithm choice and efficiency. Explicit prompts for specific methods (slicing, `reversed()`) resulted in concise and optimized $O(N)$ solutions, while a general iterative prompt led to a less efficient $O(N^2)$ solution due to repeated string concatenation.

- All generated solutions implicitly handled edge cases like empty or single-character strings due to Python's inherent string behavior, without requiring explicit conditional checks.
- The generated code generally had clear function definitions, but docstrings were absent unless explicitly requested in the prompt.

Insights or Next Steps

- To achieve desired code characteristics, such as specific algorithms, optimal efficiency, or documentation (like docstrings), prompts need to be clear and highly specific.
- While AI code generation can produce functional solutions, without explicit optimization cues, it might default to straightforward but less efficient implementations, especially for iterative processes.

TASK-04

PROMPT: Tool Comparison Reflection

```

Tool Comparison Reflection

Gemini (as an AI Assistant like myself):
  • Usability: Highly versatile and accessible within various platforms (like Colab). It excels at generating explanations, completing code, suggesting debugging steps, and even generating entire scripts or notebooks based on natural language prompts. Its strength lies in understanding context and providing comprehensive solutions or guidance.
  • Code Quality: The quality of the code generated by Gemini can be very high, often producing idiomatic and efficient Python code. It's particularly good at adhering to best practices and library conventions when explicitly prompted. However, like all AI models, it can sometimes produce less optimal or even incorrect code, especially for highly specific or complex tasks, requiring human review and iteration.

GitHub Copilot:
  • Usability: Seamlessly integrated into popular IDEs (VS Code, JetBrains IDEs). It provides real-time code suggestions as you type, functions, entire lines, or even blocks of code. Its strength is its immediacy and contextual awareness within the editor, making it feel like a pair programmer.
  • Code Quality: Copilot generally provides good quality code, often leveraging patterns it has learned from vast amounts of public code. For common programming tasks, the suggestions are usually accurate and helpful. However, it can sometimes suggest less optimal solutions, introduce subtle bugs, or produce boilerplate code that needs refinement. The quality often depends on the surrounding context and the clarity of the comments or function names.

Cursor AI:
  • Usability: Cursor AI is an IDE built around AI, offering a more deeply integrated AI experience than just a plugin. It focuses on making it easier to ask questions, refactor, debug, and even generate entire files directly within the editor using natural language. It aims to reduce context switching and keep the developer in a flow state.
  • Code Quality: Cursor's focus on in-IDE AI interaction means it can be very effective at generating targeted code based on specific requirements given within the editor. Its ability to understand and modify existing code within the project context can lead to higher quality, more integrated solutions than a standalone generator. However, like others, its output still requires careful review, especially for critical or performance-sensitive parts of an application.

Summary:
  • For quick explanations, comprehensive answers, and generating larger code blocks from scratch or for conceptual understanding, Gemini (or similar LLM assistants) is excellent.
  • For real-time coding assistance, auto-completion, and speeding up daily coding tasks within an existing IDE, GitHub Copilot is a strong choice.
  • For a deeply integrated AI-first development environment that blends coding, questioning, and refactoring into a single workflow, Cursor AI offers a unique and powerful approach.

Ultimately, the best tool often depends on the specific task and the developer's workflow. Many developers find value in using a combination of these tools.

```

OBSERVATION:

This lab demonstrates that modern AI coding tools significantly enhance productivity, learning, and code quality. Choosing the right tool depends on whether the goal is learning, professional development, refactoring, or experimentation.