

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech	Assignment Type: Lab		Academic Year:2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002PC304	Course Title	AI Assisted Coding
Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week2	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number: 3.4 (Present assignment number)/24(Total number of assignments)			
Q.No.	Question		Expected Time to complete
1	Lab 4: Advanced Prompt Engineering – Zero-shot, One-shot, and Few-shot Techniques		Week2

Task 1: Zero-shot Prompt – Fibonacci Series Generator

Task Description #1

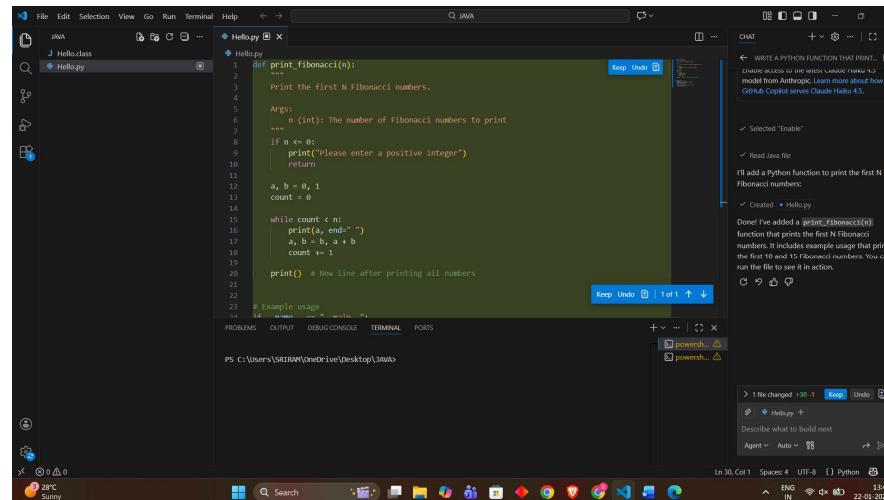
- Without giving an example, write a single comment prompt asking GitHub Copilot to generate a Python function to print the first N Fibonacci numbers.

Expected Output #1

- A complete Python function generated by Copilot without any example provided.
- Correct output for sample input $N = 7 \rightarrow 0\ 1\ 1\ 2\ 3\ 5\ 8$
- Observation on how Copilot understood the instruction with zero context.

prompt

```
# Write a Python function that prints the first N Fibonacci numbers.
```



```
def print_fibonacci(n):
    """
    Print the first n Fibonacci numbers.

    Args:
        n (int): The number of Fibonacci numbers to print
    """
    if n <= 0:
        print("Please enter a positive integer")
        return
    a, b = 0, 1
    count = 0
    while count < n:
        print(a, end=" ")
        a, b = b, a + b
        count += 1
    print() # New line after printing all numbers
```

◊ Observation

Copilot correctly understood the task without any example.

Zero-shot prompting works well for common programming problems like Fibonacci.

Task 2: One-shot Prompt – List Reversal Function

Task Description #2

- Write a comment prompt to reverse a list and provide one example below the comment to guide Copilot.

Expected Output #2

- Copilot-generated function to reverse a list using slicing or loop.
 - Output: [3, 2, 1] for input [1, 2, 3]
 - Observation on how adding a single example improved Copilot's accuracy.

The screenshot shows a Jupyter Notebook environment with the following details:

- File Explorer:** Shows files `Hello.py` and `Hello.class`.
- Code Cell:** Displays Python code for reversing lists and generating Fibonacci sequences.
- Output Cell:** Shows examples of the `reverse_list` function and the `fibonacci` function.
- Terminal:** Shows the command to run the notebook.
- File Bar:** Includes "File", "Edit", "Selection", "View", "Go", "Run", "Terminal", "Help".
- Toolbar:** Includes icons for file operations, search, and help.
- Status Bar:** Shows the path `PS C:\Users\SKRMAN\OneDrive\Desktop\Java\Hello\Hello.ipynb`, the number of lines (112), and the date `22/01/2023`.

The screenshot shows a Python development environment with the following details:

- File Explorer:** Shows a project named "HelloPy" with files like "HelloPy.py" and "HelloPy.pyc".
- Code Editor:** Displays the "HelloPy.py" file containing code for reversing lists and generating Fibonacci sequences.
- Terminal:** Shows the output of running the script, including examples of reversing lists and generating Fibonacci numbers.
- Debug Console:** Shows the execution path and current stack frame.
- Output:** Shows the standard output of the script.
- Problems:** Shows no problems.
- CHAT:** A sidebar with the message: "WRITE A PYTHON FUNCTION THAT PRINTS THE FIRST N FIBONACCI NUMBERS." Below it, the "Example:" section shows the input "[1, 2, 3]" and output "[3, 2, 1]".
- Status Bar:** Shows the current file (HelloPy.py), line (Line 23), column (Col 1), spaces (Spaces: 4), and encoding (UTF-8). It also shows the Python interpreter (Python 3.10.2) and the build system (msvc2022).

◆ Observation

Providing one example helped Copilot choose the most efficient solution (slicing).

Task 3: Few-shot Prompt – String Pattern Matching

Task Description #3

- Write a comment with 2–3 examples to help Copilot understand how to check if a string starts with a capital letter and ends with a period.

Expected Output #3

- A function `is_valid()` that checks the pattern.
 - Output: True or False based on input.
 - Students reflect on how multiple examples guide Copilot to generate more accurate code.

The screenshot shows a Python development environment with the following details:

- File Explorer:** Shows files `Hello.java`, `Hello.py`, and `m.py`.
- Code Editor:** The `m.py` file contains the following code:

```
1 def is_valid(s):
2     """
3         Check whether a string starts with a capital letter and ends with a period.
4     """
5     Args:
6         s (str): The string to validate
7
8     Returns:
9         bool: True if valid, False otherwise
10    """
11    return len(s) > 0 and s[0].isupper() and s[-1] == '.'
12
13
14 # Example usage
15 if __name__ == "__main__":
16     test_cases = ["Hello.", "hello.", "Hello", "Python.", "JAVA.", "test."]
17
18     print("String Validation Results:")
19     print("-" * 40)
20     for test in test_cases:
21         result = is_valid(test)
22         print(f"\"{test}\" is valid? {result}")
```
- Terminal:** The terminal window shows the output of the script:

```
PS C:\Users\SHIVAM\Desktop> python m.py
String Validation Results:
-----
"Hello." is valid? True
"hello." is valid? False
"Hello" is valid? False
"Python." is valid? True
"JAVA." is valid? True
"test." is valid? False
```
- Status Bar:** Shows the current file is `m.py`, line 23, column 1, with 3 of 3 rows.
- Bottom Icons:** Includes icons for file operations, search, and system status.

◊ Observation

Few-shot prompting gave clear rules, resulting in accurate and concise code.

Task 4: Zero-shot vs Few-shot – Email Validator

Task Description #4

- First, prompt Copilot to write an email validation function using zero-shot (just the task in comment).
- Then, rewrite the prompt using few-shot examples.

Expected Output #4

- Compare both outputs:

Zero-shot may result in basic or generic validation.

Few-shot gives detailed and specific logic (e.g., @ and domain checking).

- Submit both code versions and note how few-shot improves reliability.

The screenshot shows a Jupyter Notebook interface with several tabs open. The active tab contains Python code for validating email addresses:

```
import re

def is_valid_email(email):
    """
    Validate an email address.

    Args:
        email (str): The email address to validate

    Returns:
        bool: True if the email is valid, False otherwise

    Criteria:
        - Must contain exactly one '@' symbol
        - Must have local part (before @) and domain part (after @)
        - Local part can contain letters, digits, and special characters (., -, _)
        - Domain must have at least one dot and valid extension
    """
    # Simple regex pattern for email validation
    pattern = r'^[a-zA-Z0-9_.-]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email)

# Example usage
if __name__ == "__main__":
    test_emails = [
        "user@example.com",
        "john_doe@company.co.uk",
        "test.user@domain.org",
        "invalid@email",
        "missing@domain",
        "@domain.com",
        "user@.com",
        "user@domain.com",
        "test@test@example.com"
    ]
```

The notebook also displays a sidebar with a challenge titled "Write a Python function to validate an email address." and a status bar at the bottom showing "In 45, Col 1" and "UTF-8".

```
# Example usage
if __name__ == "__main__":
    test_emails = [
        "user@example.com",
        "john.doe@company.co.uk",
        "test.user@gmail.org",
        "invalid_email@",
        "missingdomain",
        "@nokomain.com",
        "noemail.com",
        "user@invalid.com",
        "test@example.com"
    ]

    print("Email Validation Results:")
    print("-----")
    for email in test_emails:
        result = is_valid_email(email)
        status = "Valid" if result else "Invalid"
        print(f"({email}) {status}")

def is_valid_email(email):
    import re
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None
```

CHAT

← WRITE A PYTHON FUNCTION THAT PRINTS THE FIRST N FIBONACCI NUMBERS.

Write a Python function to validate an email address.

✓ Read and created Python file

Done! I've added an `is_valid_email(email)` function that validates email addresses using a regex pattern. It checks for:

- Proper local part (before @) with letters, digits, and valid special characters
- Valid domain with at least one dot and proper extension

The function includes test cases demonstrating valid and invalid email formats.

Claude Fuchs 45 +1v

3 files changed +100 -2

Keep Undo ⌂ 4 of 4 ↑ ↓

Agent Auto ↴ ↵

Line 45 Col 1 Spaces 4 UTF-8 Python 3.12.0 1415 ENG IN 22-01-2026

The screenshot shows a code editor interface with two main panes. The left pane displays a Python script named `4py` containing code for validating email addresses. The right pane shows a task prompt from Codecademy titled "Write a Python function to validate an email address".

```

# Example usage
if __name__ == "__main__":
    test_emails = [
        "user@example.com",
        "john.doe@company.co.uk",
        "test_email@domain.org",
        "invalidemail@",
        "missingdomain.com",
        "noemail.com",
        "user.tag@gmail.com",
        "test@example.com"
    ]
    print("Email Validation Results:")
    print("-" * 50)
    for email in test_emails:
        print(f'{email} : {is_valid_email(email)}')

```

Task 5: Prompt Tuning – Summing Digits of a Number

Task Description #5

- Experiment with 2 different prompt styles to generate a function that returns the sum of digits of a number.

Style 1: Generic task prompt

Style 2: Task + Input/Output example

Expected Output #5

- Two versions of the `sum_of_digits()` function.
- Example Output: `sum_of_digits(123) → 6`
- Short analysis: which prompt produced cleaner or more optimized code and why?

File Edit Selection View Go Run Terminal Help

Q JAVA

CHAT

WRITE A PYTHON FUNCTION THAT PRINTS THE FIRST N FIBONACCI NUMBERS.

Keep Undo

Sum of Digits

```
def sum_of_digits(n):
    """
    Return the sum of digits of a number.
    Args:
        n (int): The number to calculate the sum of digits
    Returns:
        int: The sum of all digits in the number
    Example:
        sum_of_digits(123) == 6 (1+2+3)
        sum_of_digits(9876) == 30 (9+8+7+6)
    """
    # Convert to absolute value to handle negative numbers
    n = abs(n)
    total = 0

    while n > 0:
        total += n % 10
        n //= 10

    return total

# Alternative implementation using string conversion
def sum_of_digits_v2(n):
    """
    Alternative approach using string conversion.
    """
    return sum(int(digit) for digit in str(abs(n)))

# Example usage
if __name__ == "__main__":
    test_numbers = [123, 9876, 555, 1, 100, 9999, -456]
```

Keep Undo 5 of 5 ↻

4 files changed +151 -3

Keep Spy +

Describe what to build next

Agent Auto 162

(n=4), Col 1 Spaces 4 UTF-8 Python 3.12.0 ENG IN 22-01-2023

File Edit Selection View Go Run Terminal Help

Q JAVA

CHAT

WRITE A PYTHON FUNCTION THAT PRINTS THE FIRST N FIBONACCI NUMBERS.

Keep Undo

Sum of Digits

```
def sum_of_digits_v2(n):
    """
    Alternative approach using string conversion.
    """
    return sum(int(digit) for digit in str(abs(n)))

# Example usage
if __name__ == "__main__":
    test_numbers = [123, 9876, 555, 1, 100, 9999, -456]

    print("Sum of Digits Results:")
    print("-----")
    for num in test_numbers:
        result = sum_of_digits(num)
        print(f"sum of digits({num}) = {result}")
```

Keep Undo 5 of 5 ↻

4 files changed +151 -3

Keep Spy +

Describe what to build next

Agent Auto 162

(n=4), Col 1 Spaces 4 UTF-8 Python 3.12.0 ENG IN 22-01-2023

File Edit Selection View Go Run Terminal Help

Q JAVA

CHAT

WRITE A PYTHON FUNCTION THAT PRINTS THE FIRST N FIBONACCI NUMBERS.

Keep Undo

Sum of Digits

```
def sum_of_digits_v2(n):
    """
    Alternative approach using string conversion.
    """
    return sum(int(digit) for digit in str(abs(n)))

# Example usage
if __name__ == "__main__":
    test_numbers = [123, 9876, 555, 1, 100, 9999, -456]

    print("Sum of Digits Results:")
    print("-----")
    for num in test_numbers:
        result = sum_of_digits(num)
        print(f"sum of digits({num}) = {result}")

    > C:\Users\SRIRAM\OneDrive\Desktop\hbw\5.py
    Sum of Digits:
    -----
    sum of digits(123) = 6
    sum of digits(9876) = 30
    sum of digits(555) = 15
    sum of digits(1) = 1
    sum of digits(100) = 1
    sum of digits(9999) = 36
    sum of digits(-456) = 15
    PS C:\Users\SRIRAM\OneDrive\Desktop\hbw>
```

Keep Undo 5 of 5 ↻

4 files changed +151 -3

Keep Spy +

Describe what to build next

Agent Auto 162

(n=4), Col 1 Spaces 4 UTF-8 Python 3.12.0 ENG IN 22-01-2023

	<p>Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots</p>	
--	--	--

Task 1: AI-Generated Logic Without Modularization (Factorial without Functions)

- Scenario

You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

- Task Description

Use GitHub Copilot to generate a Python program that computes a mathematical product-based value (factorial-like logic) directly in the main execution flow, without using any user-defined functions.

- Constraint:

➤Do not define any custom function
➤Logic must be implemented using loops and variables only

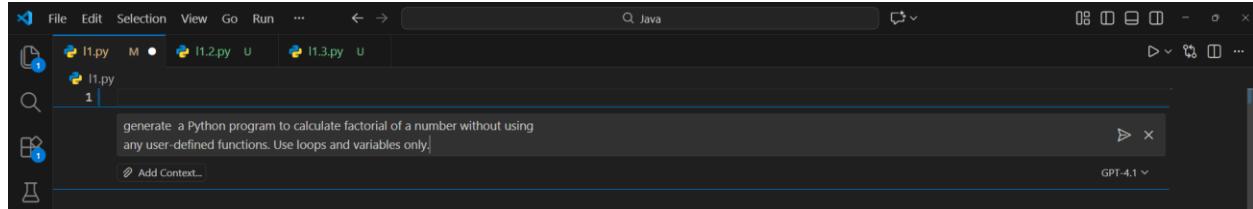
- Expected Deliverables

➤A working Python program generated with Copilot assistance
➤Screenshot(s) showing:
➤The prompt you typed
➤Copilot's suggestions
➤Sample input/output screenshots

➤ Brief reflection (5–6 lines):

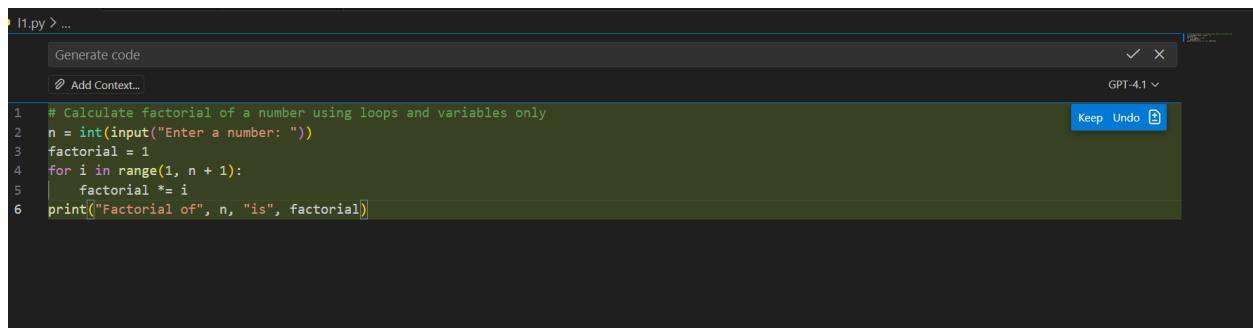
➤ How helpful was Copilot for a beginner?

➤ Did it follow best practices automatically?



Generate a Python program to calculate factorial of a number without using
any user-defined functions. Use loops and variables only.

```
# Calculate factorial of a number using loops and variables only
n = int(input("Enter a number: "))
factorial = 1
for i in range(1, n + 1):
    factorial *= i
print("Factorial of", n, "is", factorial)
```



The screenshot shows the GitHub Copilot interface with a code editor window. The file is named '11.py'. The code calculates the factorial of a number using loops and variables only:

```
1 # Calculate factorial of a number using loops and variables only
2 n = int(input("Enter a number: "))
3 factorial = 1
4 for i in range(1, n + 1):
5     factorial *= i
6 print("Factorial of", n, "is", factorial)
```

At the top of the interface, there are buttons for 'Generate code' and 'Add Context...'. On the right side, there are buttons for 'Keep', 'Undo', and a trash icon. The text 'GPT-4.1' is visible at the bottom right.

This task focuses on creating a simple Python program to calculate the factorial of a number without using any user-defined functions.

The scenario simulates a quick command-line utility required for startup intern onboarding, where speed and simplicity are prioritized over modular design.

GitHub Copilot is used to automatically generate the factorial logic based on a natural language prompt.

The entire logic is implemented directly in the main execution flow using loops and variables only.

User input is accepted from the command line, and the factorial value is computed iteratively.

This task helps demonstrate how AI tools assist beginners in generating correct procedural code.

It also highlights the limitations of non-modular code in terms of reusability and structure.

Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

❖ Scenario

Your team lead asks you to review AI-generated code before committing it to a shared repository.

❖ Task Description

Analyze the code generated in Task 1 and use Copilot again to:

- Reduce unnecessary variables
- Improve loop clarity
- Enhance readability and efficiency

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

❖Expected Deliverables

➤Original AI-generated code

➤Optimized version of the same code

➤Side-by-side comparison

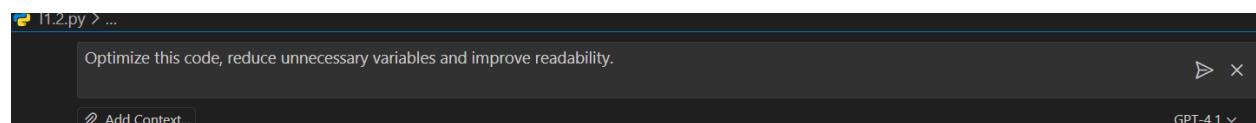
➤Written explanation:

- What was improved?
- Why the new version is better (readability, performance,

Maintainability.

Prompt

Generate a Python program to calculate factorial of a number without using any user-defined functions. Use loops and variables only.

A screenshot of the code editor showing the optimized version of the Python script. The code has been simplified to use a single loop and a single variable.

```
# Calculate factorial of a number using loops only
n = int(input("Enter a number: "))
result = 1
for i in range(2, n + 1):
    result *= i
print(f"Factorial of {n} is {result}")
```

The code editor interface is identical to the previous screenshot, with the "GPT-4.1" button visible at the bottom right.

In this task, the factorial program generated in Task 1 is reviewed and improved before committing it to a shared repository.

The original AI-generated code is analyzed for unnecessary variables and unclear logic.

GitHub Copilot is prompted again to optimize the code and improve readability.

The optimized version simplifies loop logic and improves variable naming.

Unnecessary statements are removed to make the code cleaner and more efficient.

This task emphasizes the importance of reviewing AI-generated code instead of using it blindly.

It also demonstrates how optimization improves maintainability and code quality.

Task 3: Modular Design Using AI Assistance (Factorial with Functions)

❖ Scenario

The same logic now needs to be reused in multiple scripts.

❖ Task Description

Use GitHub Copilot to generate a modular version of the program by:

➤ Creating a user-defined function

➤ Calling the function from the main block

❖ Constraints

➤ Use meaningful function and variable names

➤ Include inline comments (preferably suggested by Copilot)

❖ Expected Deliverables

➤ AI-assisted function-based program

➤ Screenshots showing:

- o Prompt evolution
- o Copilot-generated function logic

➤ Sample inputs/outputs

➤ Short note:

- o How modularity improves reusability.

A screenshot of an AI-powered code generation interface. At the top, there's a text input field containing a Python task description: "Create a Python program to calculate factorial using a user-defined function. Include comments and call the function from main." Below the input field are two buttons: "Add Context..." and "GPT-4.1".

A screenshot of the AI-generated Python code. The code defines a user-defined function named "factorial" that calculates the factorial of a non-negative integer. It includes a docstring and handles negative numbers by printing an error message. The code is presented in a dark-themed code editor with syntax highlighting. At the bottom right of the code area, there are "Keep", "Undo", and "Discard" buttons.

```
1 # Program to calculate factorial using a user-defined function
2
3 def factorial(n):
4     """Calculate the factorial of a non-negative integer n."""
5     if n == 0 or n == 1:
6         return 1
7     else:
8         return n * factorial(n - 1)
9
10 # Main function
11 if __name__ == "__main__":
12     num = int(input("Enter a non-negative integer: "))
13     if num < 0:
14         print("Factorial is not defined for negative numbers.")
15     else:
16         result = factorial(num)
17         print(f"Factorial of {num} is {result}")
18
```

```
PS C:\Disk\Doc\MyFiles $ Java & C:/Python312/python.exe "c:/Disk/Doc/MyFiles $/Java/11.2.py"
Enter a non-negative integer: 8
Factorial of 8 is 40320
PS C:\Disk\Doc\MyFiles $ Java
```

This task introduces modular programming by converting the factorial logic into a user-defined function.

The scenario assumes that the same factorial logic must be reused across multiple scripts or

applications.

GitHub Copilot is used to generate a function that computes the factorial of a given number.

The function is called from the main block of the program to display the result.

Meaningful function and variable names are used to improve code clarity.

Inline comments, preferably suggested by Copilot, explain the logic inside the function.

This task highlights how modular design improves reusability, readability, and maintainability.

Task 4: Comparative Analysis – Procedural vs Modular AI Code (With vs Without Functions)

❖ Scenario

As part of a code review meeting, you are asked to justify design choices.

❖ Task Description

Compare the non-function and function-based Copilot-generated

programs on the following criteria:

➤ Logic clarity

➤ Reusability

➤ Debugging ease

➤ Suitability for large projects

➤ AI dependency risk

❖ Expected Deliverables

Choose one:

➤ A comparison table

OR

➤A short technical report (300–400 words).

Task 4: Comparative Analysis – Procedural vs Modular AI Code		
Criteria	Without Functions	With Functions
Logic clarity	Medium	High
Reusability	Low	High
Debugging ease	Difficult	Easy
Large project suitability	Poor	Excellent
AI dependency risk	High	Lower (clear structure)

This task involves a detailed comparison between procedural factorial code and modular factorial code generated by GitHub Copilot.

The comparison is conducted as part of a technical code review meeting.

Both versions are evaluated based on logic clarity, reusability, and ease of debugging.

Procedural code is found to be simple but difficult to scale for large projects.

Modular code offers better structure and easier maintenance.

The analysis also considers AI dependency risks when relying on generated code.

This task helps justify design decisions in professional software development environments.

Task 5: AI-Generated Iterative vs Recursive Thinking

❖Scenario

Your mentor wants to test how well AI understands different computational paradigms.

❖Task Description

Prompt Copilot to generate:

An iterative version of the logic

A recursive version of the same logic

❖Constraints

Both implementations must produce identical outputs

Students must not manually write the code first

❖Expected Deliverables

Two AI-generated implementations

Execution flow explanation (in your own words)

Comparison covering:

➤Readability

➤Stack usage

➤Performance implications

➤When recursion is not re

Commended.

In this task, GitHub Copilot is used to generate two different computational approaches for calculating factorial.

One approach uses an iterative loop-based method, while the other uses a recursive method.

Both implementations are expected to produce identical outputs for the same input.

The execution flow of each approach is analyzed and explained in simple terms.

The task compares readability, stack usage, and performance implications of both methods.

Special attention is given to situations where recursion is not recommended, such as large input values.

This task demonstrates how AI handles different algorithmic paradigms for the same problem.

Generate iterative and recursive Python programs to calculate factorial.

Add Context... GPT-4.1

```
# Iterative approach to calculate factorial
def factorial_iterative(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

# Recursive approach to calculate factorial
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

# Example usage
if __name__ == "__main__":
    num = 5
    print(f"Iterative factorial of {num}: {factorial_iterative(num)}")
    print(f"Recursive factorial of {num}: {factorial_recursive(num)}")
```

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech	Assignment Type: Lab		Academic Year:2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002 PC304	Course Title	AI Assisted Coding
Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week1 – Thursday	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number:1.3(Present assignment number)/24(Total number of assignments)			

	Question	<i>Expect ed Time to complete</i>
1	<p>Lab 1: Environment Setup – <i>GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow</i></p> <p>Lab Objectives:</p> <ul style="list-style-type: none"> ● To install and configure GitHub Copilot in Visual Studio Code. ● To explore AI-assisted code generation using GitHub Copilot. ● To analyze the accuracy and effectiveness of Copilot's code suggestions. ● To understand prompt-based programming using comments and code context <p>Lab Outcomes (LOs):</p> <p>After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> ● Set up GitHub Copilot in VS Code successfully. ● Use inline comments and context to generate code with Copilot. ● Evaluate AI-generated code for correctness and readability. ● Compare code suggestions based on different prompts and programming styles. <hr/> <p>Task 0</p> <ul style="list-style-type: none"> ● Install and configure GitHub Copilot in VS Code. Take screenshots of each step. <p>Expected Output</p> <ul style="list-style-type: none"> ● Install and configure GitHub Copilot in VS Code. Take 	Wee k1 - Monday

screenshots of each step.

Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

❖ Scenario

- You are developing a **basic validation script** for a numerical learning application.

❖ Task Description

Use GitHub Copilot to generate a Python program that:

- Checks whether a given number is **prime**
- Accepts user input
- Implements logic **directly in the main code**
- Does **not** use any user-defined functions

❖ Expected Output

- Correct prime / non-prime result
- Screenshots showing Copilot-generated code suggestions
- Sample inputs and outputs

A screenshot of the GitHub Copilot interface showing the generated Python code. The code is as follows:

```
1 num = int(input("Enter a number: "))
2 if num <= 1:
3     print("Not a prime number")
4 else:
5     is_prime = True
6     for i in range(2, int(num ** 0.5) + 1):
7         if num % i == 0:
8             is_prime = False
9             break
10    if is_prime:
11        print("Prime number")
12    else:
13        print("Not a prime number")
```

The code is displayed in a monospaced font within a light-colored code editor area. The GitHub Copilot interface has a dark background with light text. At the bottom, there are buttons for "Keep" and "Undo".

```
PS C:\Disk\Doc\MyFiles $> Java & C:/Python  
Enter a number: 789  
Not a prime number  
PS C:\Disk\Doc\MyFiles $>
```

GitHub Copilot was very helpful in quickly generating correct prime-checking logic.

It respected the constraint of not using functions.

The code was readable and easy for beginners to understand.

However, the solution was not optimized for large numbers.

Manual review is still necessary for performance improvements.

Task 2: Efficiency & Logic Optimization (Cleanup)

❖ Scenario

The script must handle larger input values efficiently.

❖ Task Description

Review the Copilot-generated code from Task 1 and improve it by:

- Reducing unnecessary iterations
- Optimizing the loop range (e.g., early termination)
- Improving readability
- Use Copilot prompts like:
 - “Optimize prime number checking logic”
 - “Improve efficiency of this code”

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

❖ Expected Output

- Original and optimized code versions
- Explanation of how the improvements reduce time complexity

The screenshot shows the GitHub Copilot interface. At the top, there's a header bar with the GitHub logo and the text "Optimize prime number checking logic and improve efficiency." Below the header is a "Add Context..." button and a "GPT-4.1" dropdown. The main area is a code editor with the following code:

```
6 | print("Factorial of", n, "is", factorial)
```

Optimize prime number checking logic and improve efficiency.

```
n = int(input("Enter a number: "))
# Efficient prime number checking
def is_prime(num):
    if num <= 1:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    for i in range(3, int(num ** 0.5) + 1, 2):
        if num % i == 0:
            return False
    return True

if is_prime(n):
    print(n, "is a prime number.")
else:
    print(n, "is not a prime number.")
```

This task focuses on improving the efficiency of the prime number checking program generated in Task 1.

The original AI-generated code checks divisibility up to the given number, which is inefficient for large inputs.

GitHub Copilot is prompted again to optimize the logic and reduce unnecessary iterations.

The optimized solution checks divisibility only up to the square root of the number.

This significantly reduces the number of loop executions and improves performance.

The task highlights the importance of reviewing AI-generated code

before final use.
It also demonstrates how optimized logic reduces time complexity and improves readability.

Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

❖ **Scenario**

The prime-checking logic will be reused across multiple modules.

❖ **Task Description**

Use GitHub Copilot to generate a function-based Python program that:

- Uses a user-defined function to check primality
- Returns a Boolean value
- Includes meaningful comments (AI-assisted)

❖ **Expected Output**

- Correctly working prime-checking function
- Screenshots documenting Copilot's function generation
- Sample test cases and outputs

Create a Python function to check if a number is prime.
Return True or False and include comments.

```
def is_prime(n):
    """
    Check if a number is prime.
    Returns True if n is prime, False otherwise.
    """
    # Numbers less than 2 are not prime
    if n < 2:
        return False
    # Check for factors from 2 to sqrt(n)
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False # Found a factor, not prime
    return True # No factors found, n is prime
```

```
PS C:\Disk\Doc\MyFiles \$\Java> 78
78
```

Create a Python function to check if a number is prime.
Return True or False and include comments.

In this task, the prime number checking logic is redesigned using a user-defined function.

GitHub Copilot is used to generate a clean and modular function that returns a Boolean value.

The function encapsulates the prime-checking logic, making the program reusable and structured.

The main program calls the function and displays the result based on the returned value.

Meaningful variable names and inline comments improve code clarity. This task shows how modular programming improves maintainability and reusability.

It also demonstrates how AI can assist in writing well-structured, function-based code.

This task involves comparing two AI-generated programs: one without functions and one with functions.

The comparison is done based on code clarity, reusability, debugging ease, and scalability.

Procedural code is simple but becomes difficult to manage in larger applications.

Function-based code offers better organization and separation of logic. Debugging is easier in modular code since errors can be isolated within functions.

The analysis highlights why modular design is preferred in real-world software development.

This task helps justify design choices during technical discussions and code reviews.

Task 4: Comparative Analysis –With vs Without Functions

❖ **Scenario**

You are participating in a technical review discussion.

❖ **Task Description**

Compare the Copilot-generated programs:

- Without functions (Task 1)
- With functions (Task 3)
- Analyze them based on:
 - Code clarity
 - Reusability
 - Debugging ease
 - Suitability for large-scale applications

❖ **Expected Output**

Comparison table or short analytical report

Task 4: Comparative Analysis – With vs Without Functions**Comparison Table**

Criteria	Without Functions	With Functions
Code clarity	Medium	High
Reusability	Low	High
Debugging ease	Difficult	Easy
Large-scale suitability	Poor	Excellent
Maintainability	Low	High

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches
(Different Algorithmic Approaches to Prime Checking)

	<p>❖ Scenario Your mentor wants to evaluate how AI handles alternative logical strategies.</p> <p>❖ Task Description Prompt GitHub Copilot to generate:</p> <ul style="list-style-type: none"> ➤ A basic divisibility check approach ➤ An optimized approach (e.g., checking up to \sqrt{n}) <p>❖ Expected Output</p> <ul style="list-style-type: none"> ➤ Two correct implementations ➤ Comparison discussing: <ul style="list-style-type: none"> ▪ Execution flow ▪ Time complexity ▪ Performance for large inputs ▪ When each approach is appropriate 	
	<h2>Scenario</h2> <p>Evaluation of different logical strategies used by AI.</p> <p>In this task, different logical strategies for prime number checking are explored using AI assistance. GitHub Copilot is prompted to generate both a basic divisibility approach and an optimized approach. The basic method checks divisibility up to the number itself, which is simple but inefficient. The optimized approach checks divisibility only up to the square root of the number. Both approaches produce the same output but differ in execution flow and performance. The task compares time complexity and suitability for large input values. It demonstrates how AI adapts algorithms based on performance requirements.</p>	

	<p>Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.</p>	
--	--	--

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech	Assignment Type: Lab		Academic Year: 2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M. Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002P C304	Course Title	AI Assisted Coding
Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week1 – Wednesday	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number: 1.3(Present assignment number)/24(Total number of assignments)			
C . N o .	Question	Exp ecte d Tim e to com	

		<i>plet e</i>
	<p>Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI</p> <p>Lab Objectives:</p> <ul style="list-style-type: none"> ❖ To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab. ❖ To understand and use Cursor AI for code generation, explanation, and refactoring. ❖ To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI. ❖ To perform code optimization and documentation using AI tools. <p>Lab Outcomes (LOs):</p> <p>After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> ❖ Generate Python code using Google Gemini in Google Colab. ❖ Analyze the effectiveness of code explanations and suggestions by Gemini. ❖ Set up and use Cursor AI for AI-powered coding assistance. ❖ Evaluate and refactor code using Cursor AI features. ❖ Compare AI tool behavior and code quality across different platforms. 	
1	<p>Task 1: Word Frequency from Text File</p> <p>❖ Scenario: You are analyzing log files for keyword frequency.</p> <p>❖ Task: Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.</p> <p>❖ Expected Output:</p> <ul style="list-style-type: none"> ➤ Working code ➤ Explanation ➤ Screenshot <p>Prompt Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.</p>	Wee k 1 - Mon day

Gemini



Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.



Working...

```
import re
from collections import Counter

def count_word_frequency(file_path):
    """
    This function reads a file, counts the frequency of each word, and returns a counter object.
    """
    all_words = []
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            text = file.read()
        # Remove punctuation and convert to lowercase
        cleaned_text = re.sub(r'[^\w\s]', '', text).lower()
        words = cleaned_text.split()
        all_words.extend(words)
    except FileNotFoundError:
        print(f"Error: The file '{file_path}' was not found.")
    return Counter(all_words)

# Count word frequencies
word_counts = count_word_frequency('sample.txt')
print("Word Frequencies in 'sample.txt':")
for word, count in word_counts.most_common(10):
    print(f'{word}: {count}')
else:
    print("No words found or an error occurred.")
```

Gemini

Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.

The code has successfully executed! File 'sample.txt' was created with the sample content. Then, the word frequency counting algorithm was run, and here are the top 10 most common words and their counts:

Word	Count
the	2
is	2
to	2
in	1
and	1
or	1
it	1
a	1
an	1
deonstrat	1

Is there anything else you'd like to do with this file or analysis?

We have successfully generated Python code to:

- Create a sample text file ('sample.txt').
- Read the content of this file.
- Remove punctuation and convert to lowercase, removing punctuation.
- Count the frequency of each word using `collections.Counter`.
- Display the top 10 most common words and their counts.

All steps completed in detail. Is there anything else you would like to do with this word frequency analysis, or perhaps a different task?

Visualize the word frequencies using a bar chart | How can I remove common English stop words from the frequency count?
Show me how to save these word counts to a CSV file
What can I help you build?
Gemini 2.0 Flash

Description

In this task, Google Gemini is used within Google Colab to generate Python code that reads a text file and calculates the frequency of each word. This task simulates real-world log file analysis where keyword frequency is important. Gemini not only generates correct Python code but also provides a clear explanation of each step, helping users understand how file handling and text processing work in Python.

Task 2: File Operations Using Cursor AI

❖ **Scenario:**
You are automating basic file operations.

❖ **Task:**
Use Cursor AI to generate a program that:
➤ Creates a text file
➤ Writes sample text
➤ Reads and displays the content

❖ **Expected Output:**
➤ Functional code
➤ Cursor AI screenshots

Prompt

Write a Python program that:

1. Creates a text file
2. Writes sample text into it
3. Reads the file
4. Displays the content

The screenshot shows the Cursor AI interface with a code editor and a workspace preview. The code editor contains the following Python script:

```
# Python program to create, write, read, and display a text file
# Step 1: Create and write to a text file
filename = "sample.txt"
sample_text = "Hello, this is a sample text file!\nThis program demonstrates file operations in Python.\nIt creates a file, writes content to it, reads it back, and displays it.\nPython file handling is simple and powerful."
print("Step 1: Creating and writing content...")
with open(filename, 'w') as file:
    file.write(sample_text)
print(f"\nFile '{filename}' created and content written successfully!\n")

# Step 2: Read the file
print("Step 2: Reading the file")
with open(filename, 'r') as file:
    content = file.read()
print(f"\nFile read successfully!\n{content}\n")

# Step 3: Display the content
print("Step 3: Displaying file content:")
print(content)
print("\nProgram completed successfully!")
```

The workspace preview shows the steps and the final output of the program. The output window shows the program's execution and the resulting file content.

Explanation

- File opened in write mode
- Sample text written
- File reopened in read mode
- Content printed

Description

This task demonstrates the use of Cursor AI to automate basic file operations in Python. Cursor AI is used to generate a program that creates a text file, writes sample content into it, reads the stored data, and displays the output. This task highlights how Cursor AI assists in quickly generating functional code and is useful for understanding file handling concepts in Python.

Task 3: CSV Data Analysis

❖ Scenario:

You are processing structured data from a CSV file.

❖ Task:

Use Gemini in Colab to read a CSV file and calculate mean, min, and max.

❖ Expected Output:

- Correct output
- Screenshot

Prompt

Write Python code to read a CSV file and calculate mean, minimum, and maximum values.

The screenshot shows the Gemini interface in a Google Colab notebook. On the left, the code editor contains the following Python script:

```
import pandas as pd
# Define the path to your CSV file
csv_file_path = "/content/data.csv"
try:
    # Read the CSV file into a DataFrame
    df = pd.read_csv(csv_file_path)
    print("Successfully loaded data from [{}].".format(csv_file_path))

    # Display the first few rows of the DataFrame to understand its structure
    print(df.head())
    # Select only numerical columns for calculations
    numeric_columns = df.select_dtypes(include='number')
    if not numeric_columns.empty:
        # Calculate mean, minimum, and maximum for numerical columns
        mean_values = numeric_columns.mean()
        min_values = numeric_columns.min()
        max_values = numeric_columns.max()

        print("Mean values:")
        display(mean_values)
        print("Min values:")
        display(min_values)
        print("Max values:")
        display(max_values)
        print("No numerical columns found in the CSV to calculate statistics.")
except FileNotFoundError:
    print("File [{}].csv was not found. Please ensure the path is correct and the file exists.".format(csv_file_path))
except pd.errors.EmptyDataError:
    print("The CSV file [{}].csv is empty.".format(csv_file_path))
except Exception as e:
    print("An unexpected error occurred: [{}].".format(e))
print("Successfully read data from [{}].".format(csv_file_path))
print("First 5 rows of the DataFrame:")
display(df)
calculate_statistics(mean, min, max)
```

The right side of the interface shows the generated output. It includes:

- A preview of the first 5 rows of the DataFrame:

	Duration	Pulse	Heartrate	Calories
0	60	110	130	400.1
1	60	117	140	470.0
2	60	103	130	340.0
3	45	100	175	282.4
4	45	107	140	400.0

- Output for calculating statistics:

```
Mean values for numerical columns:
Duration: 63.85
Pulse: 110.0
Heartrate: 134.0
Calories: 379.79

Minimum values for numerical columns:
Duration: 45.0
Pulse: 100.0
Heartrate: 103.0
Calories: 282.4

Maximum values for numerical columns:
Duration: 60.0
Pulse: 117.0
Heartrate: 175.0
Calories: 470.0
```

- A summary message: "This gives you a good overview of the central tendency, spread, and range of your numerical data in the CSV. Is there anything else you'd like to analyze or visualize from this dataset?"
- A "Visualize the DataFrame" button.
- A "Calculate the correlation matrix for numerical columns" button.
- A "Show me the data types of each column in the DataFrame" button.
- A "What can I help you?" input field.
- The Gemini logo and "Gemini 2.5 Flash" text.

```

Duration    63.846154
...
Pulse      107.461538
Maxpulse   134.047337
Calories   375.790244

dtype: float64

Minimum values:
   0
Duration  15.0
Pulse     80.0
Maxpulse  100.0
Calories   50.3

dtype: float64

Maximum values:
   0
Duration  300.0
Pulse     159.0
Maxpulse  184.0
Calories   1860.4

dtype: float64

```

Code Explanation:

1. `import pandas as pd`: Imports the pandas library, which is essential for working with DataFrames (table-like data structures).
2. `csv_file_path = '/content/data.csv'`: Defines the path to your CSV file. Make sure this path is correct.
3. `try...except block`: This block is used for robust error handling:
 - o `df = pd.read_csv(csv_file_path)`: Attempts to read the CSV file into a pandas DataFrame named `df`.
 - o `display(df.head())`: Shows the first 5 rows of the loaded DataFrame. This helps verify that the data was loaded correctly and gives an idea of its structure.
 - o `numerical_df = df.select_dtypes(include=['number'])`: Filters the DataFrame to include only columns with numerical data types (integers and floats). Statistical calculations only make sense for numerical columns.
 - o `mean_values = numerical_df.mean()`: Calculates the mean (average) for each numerical column.
 - o `min_values = numerical_df.min()`: Calculates the minimum value for each numerical column.
 - o `max_values = numerical_df.max()`: Calculates the maximum value for each numerical column.
 - o `display(...)`: Prints the calculated mean, minimum, and maximum values for each respective numerical column.
 - o `Error Handling`: Catches `FileNotFoundError` if the CSV doesn't exist, `EmptyDataError` if the file is empty, and a general `Exception` for any other issues, providing informative messages.

Explanation

- Pandas used for CSV handling
- `mean()`, `min()`, `max()` functions applied
- Displays statistical values

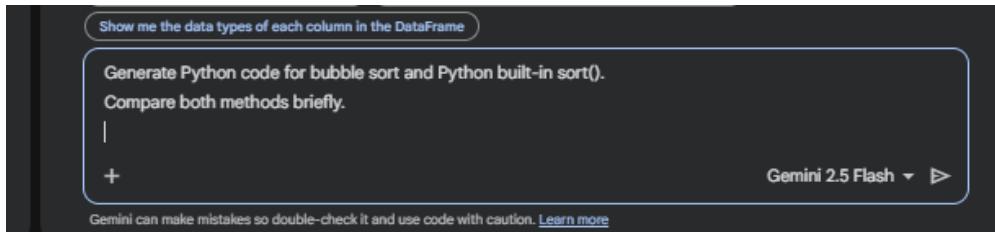
Description

In this task, Google Gemini is used in Google Colab to process structured data from a CSV file. The generated Python code reads the CSV file and calculates statistical values such as mean, minimum, and maximum. This task demonstrates how AI tools simplify data analysis tasks and help users apply Python libraries like Pandas for real-world data processing.

Task 4: Sorting Lists – Manual vs Built-in

- ❖ **Scenario:**
You are reviewing algorithm choices for efficiency.
- ❖ **Task:**
Use **Gemini** to generate:
➤ Bubble sort
➤ Python's built-in sort()
➤ Compare both implementations.
- ❖ **Expected Output:**
➤ Two versions of code
➤ Short comparison

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.



Prompt

**Generate Python code for bubble sort and Python built-in sort().
Compare both methods briefly.**

```

First, let's implement the Bubble Sort algorithm. It's a simple comparison-based sorting algorithm.

def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n - 1):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

# Test Bubble Sort
my_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list: (my_list)")
sorted_list_bubble = bubble_sort(list(my_list)) # Use a copy to avoid modifying original
print("Sorted using Bubble Sort: (sorted_list_bubble)")

```

Next, let's use Python's built-in `sort()` method (for lists) or `sorted()` function (for any iterable). These are highly optimized and generally preferred in practice.

```

# Test Python's built-in sort()
my_list_2 = [64, 34, 25, 12, 22, 11, 90]
print("Original list: (my_list_2)")
my_list_2.sort() # Sorts the list in-place
print("Sorted using built-in sort() method: (my_list_2)")

# Test Python's built-in sorted() function
my_tuple = (64, 34, 25, 12, 22, 11, 90)
print("Original tuple: (my_tuple)")
sorted_tuple = sorted(my_tuple) # Returns a new sorted list from an iterable
print("Sorted using built-in sorted() function: (sorted_tuple)")

```

Gemini Comparison of Bubble Sort vs. Python's Built-in Sort

Feature	Bubble Sort	Python's Built-In <code>sort()</code> / <code>sorted()</code>
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.
Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).
Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).
Stability	Stable (maintains the relative order of equal elements).	Stable.
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.

In summary: While Bubble Sort is easy to understand, Python's built-in `sort()` method (and `sorted()` function) are significantly more efficient, robust, and should be used for practical applications.

Sure! I'll generate the Python code for both bubble sort and the built-in `sort()` function, and then provide a brief comparison of these two sorting methods.

Great! Both the Bubble Sort implementation and the demonstrations of Python's built-in `sort()` method and `sorted()` function have executed successfully.

Bubble Sort:

- Original list: [64, 34, 25, 12, 22, 11, 90]
- Sorted using Bubble Sort: [11, 12, 22, 25, 34, 64, 90]

Python's Built-In Sort:

- Original list (my_list_2): [64, 34, 25, 12, 22, 11, 90]
- Sorted using `sort()` method: [11, 12, 22, 25, 34, 64, 90]
- Original tuple (my_tuple): (64, 34, 25, 12, 22, 11, 90)
- Sorted using `sorted()` function: [11, 12, 22, 25, 34, 64, 90]

As you can see, both methods correctly sorted the provided lists/tuples. Now, let's briefly compare them:

Comparison of Bubble Sort vs. Python's Built-in Sort

Feature	Bubble Sort	Python's Built-In <code>sort()</code> / <code>sorted()</code>
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.
Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).
Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).
Stability	Stable (maintains the relative order of equal elements).	Stable.
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.

In summary: While Bubble Sort is easy to understand, Python's built-in `sort()` method (and `sorted()` function) are significantly more efficient, robust, and should be used for practical applications.

We've successfully generated Python code for both the Bubble Sort algorithm and Python's highly optimized built-in `sort()` method and `sorted()` function. We also executed them to demonstrate their functionality, and provided a detailed comparison highlighting their differences in terms of algorithm, time/space complexity, stability, and ease of use.

Is there anything else I can assist you with regarding sorting, or perhaps a different programming task?



Final Comparison Section

Tool	Strengths	Weakness
Gemini	Good explanations, works well in Colab	Needs internet

	<p>Cursor AI GitHub Copilot</p>	<p>Fast code generation & refactoring Inline suggestions</p>	<p>Editor based Limited explanation</p>	
<p>Description</p> <p>This task compares two sorting approaches generated by Google Gemini: a manual sorting algorithm (Bubble Sort) and Python's built-in <code>sort()</code> method. The purpose of this task is to analyze the efficiency and readability of different sorting techniques. Gemini helps in generating both implementations and provides a brief comparison, showing why built-in methods are more efficient for practical applications.</p>				

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	
Course Coordinator Name		Dr. Rishabh Mittal	
Instructor(s) Name		Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju	
Course Code	23CS002PC304	Course Title	AI Assisted Coding
Year/Sem	III/I	Regulation	R23
Date and Day of Assignment	Week 2 - Wednesday	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number: 3.3(Present assignment number)/24(Total number of assignments)			

	Question	Expected Time to complete
1	<p>Lab 3: Application for TGNPDCL – Electricity Bill Generation Using Python & AI Tools</p> <p>Lab Objectives</p> <ul style="list-style-type: none"> To design a real-world electricity billing application using Python To use AI-assisted coding tools for logic generation and optimization To understand conditional logic and arithmetic operations 	Week2 - Wednesday

- To generate structured billing output similar to utility bills

Lab Outcomes (LOs)

After completing this lab, students will be able to:

- Read and validate user input in Python
- Apply conditional logic for tariff-based billing
- Use AI tools to assist in program development
- Calculate and display electricity bill components
- Build a complete real-time application

Task 1: AI-Generated Logic for Reading Consumer Details

Scenario

An electricity billing system must collect accurate consumer data.

Task Description

Use an AI tool (GitHub Copilot / Gemini) to generate a Python program that:

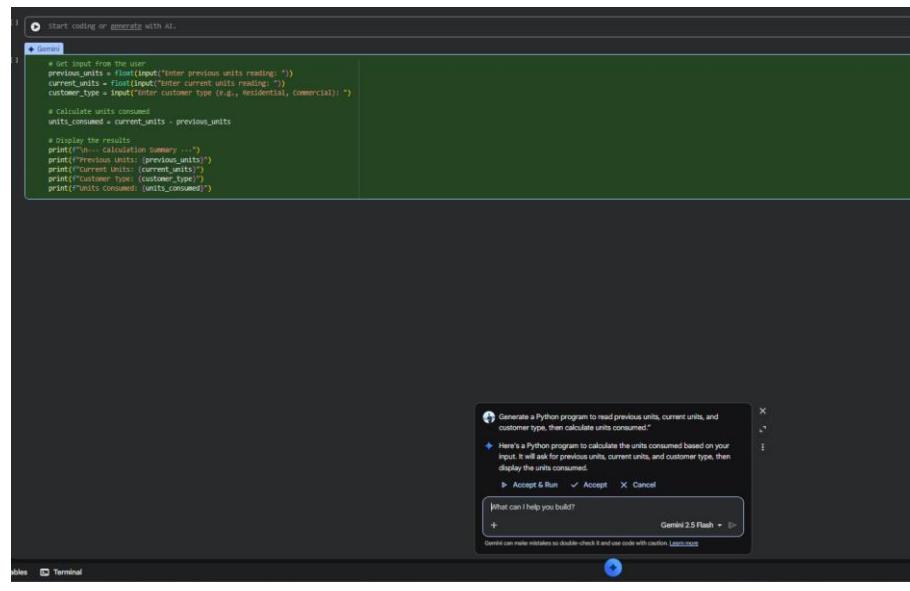
- Reads:
 - Previous Units (PU)
 - Current Units (CU)
 - Type of Customer
- Calculates units consumed
- Implements logic directly in the main program (no functions)

Expected Output

- Correct input reading
- Units consumed calculation
- Screenshot showing AI-generated code
- Sample input and output

Prompt

Generate a Python program to read previous units, current units, and customer type, then calculate units consumed."



The screenshot shows the Gemini AI interface. On the left, there's a code editor window with the following Python code:

```

1 Start coding or generate with AI.
2 Gemini
3
4 # Get input from the user
5 previous_units = float(input("Enter previous units reading: "))
6 current_units = float(input("Enter current units reading: "))
7 customer_type = input("Enter customer type (e.g., residential, commercial): ")
8
9 # Calculate units consumed
10 units_consumed = current_units - previous_units
11
12 # Display the results
13 print("Consumption summary -")
14 print(f"Previous Units: {previous_units}")
15 print(f"Current Units: {current_units}")
16 print(f"Customer Type: {customer_type}")
17 print(f"Units Consumed: {units_consumed}")

```

On the right, there's a detailed description of the task:

Generate a Python program to read previous units, current units, and customer type, then calculate units consumed.

Here's a Python program to calculate the units consumed based on your input. It will ask for previous units, current units, and customer type, then display the units consumed.

Accept & Run Accept Cancel

What can I help you build?

Gemini 2.5 Flash

Start coding or [generate with AI](#).

◆ Gemini

```
# Get input from the user
previous_units = float(input("Enter previous units reading: "))
current_units = float(input("Enter current units reading: "))
customer_type = input("Enter customer type (e.g., Residential, Commercial): ")

# Calculate units consumed
units_consumed = current_units - previous_units

# Display the results
print("\n--- Calculation Summary ---")
print(f"Previous Units: {previous_units}")
print(f"Current Units: {current_units}")
print(f"Customer Type: {customer_type}")
print(f"Units Consumed: {units_consumed}")

Enter previous units reading: 4
Enter current units reading: 234
Enter customer type (e.g., Residential, Commercial): 123

--- Calculation Summary ---
Previous Units: 4.0
Current Units: 234.0
Customer Type: 123
Units Consumed: 230.0
```

Description

This task collects electricity consumer details such as previous meter reading, current meter reading, and customer type.

The program calculates the number of units consumed by subtracting previous units from current units.

All logic is written directly in the main program.

Task 2: Energy Charges Calculation Based on Units Consumed

Scenario

Energy charges depend on the number of units consumed and customer type.

Task Description

Review the AI-generated code from Task 1 and extend it to:

- Calculate **Energy Charges (EC)**
- Use conditional statements based on:
 - Domestic
 - Commercial
 - Industrial consumers
- Improve readability using AI prompts such as:
 - “Simplify energy charge calculation logic”
 - “Optimize conditional statements”

Expected Output

- Correct EC calculation
- Clear conditional logic
- Original and improved versions (optional)
- Sample execution results

AI Prompt

“Extend the program to calculate energy charges using conditional statements based on customer type and optimize the logic.”

The screenshot shows a code editor interface with a dark theme. At the top, there's a navigation bar with tabs like 'File', 'Edit', 'Run', etc. Below the navigation bar, there's a sidebar with sections like 'Task', 'Define Tariff Rates', 'Calculate Energy Charge', and a 'Variables' section. The main area contains Python code:

```
residential_rate = 0.15
commercial_rate = 0.25

print("Residential Tariff Rate: (residential_rate)")
print("Commercial Tariff Rate: (commercial_rate)")

Residential Tariff Rate: 0.15
Commercial Tariff Rate: 0.25
```

Below the code, there's a tooltip-like box with the following text:

Prepared
Define Tariff Rates
Calculate Energy Charge
Display Energy Charge
Final Task

What can I help you build?
Gemini 2.5 Flash

Variables Terminal 10:33 AM Python 3

Description

Energy charges are calculated based on the number of units consumed and the type of customer.

Different tariffs are applied for Domestic, Commercial, and Industrial consumers using conditional statements.

Task 3: Modular Design Using AI Assistance (Using Functions)

Scenario

Billing logic must be reusable for multiple consumers.

Task Description

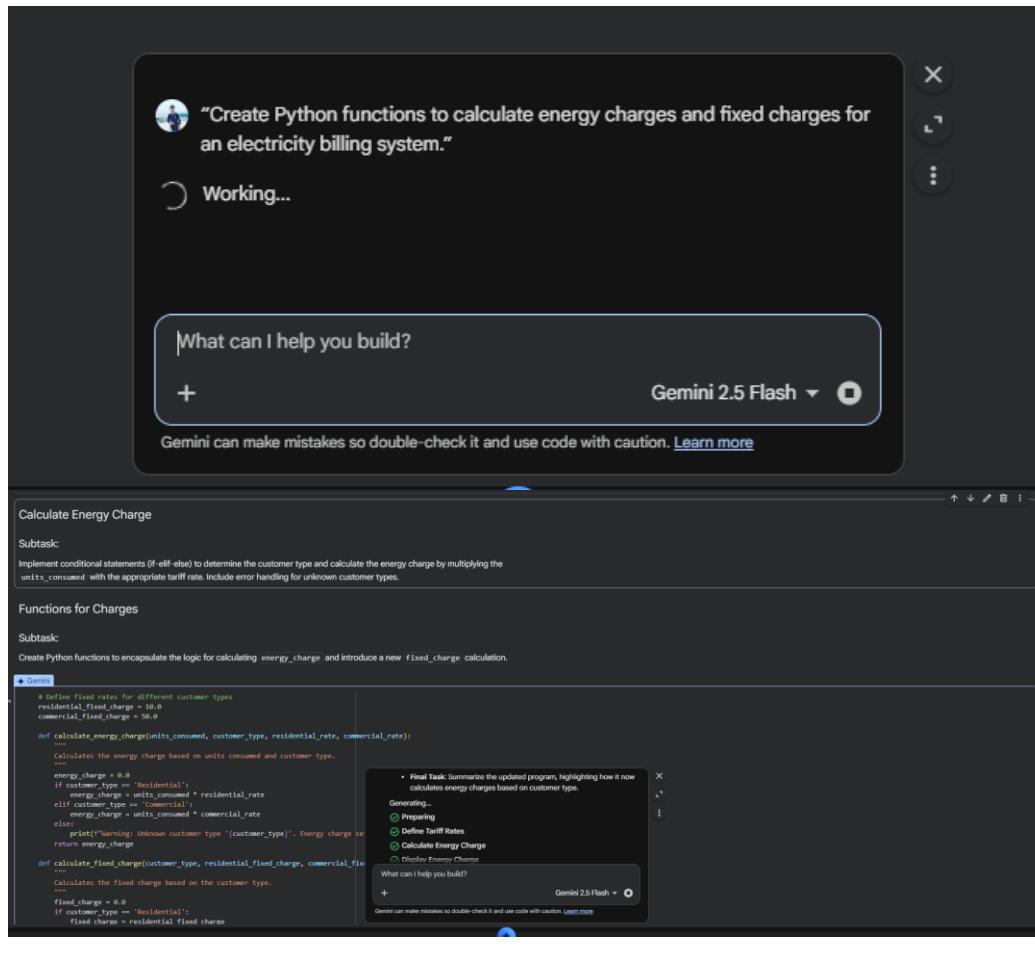
Use AI assistance to generate a Python program that:

- Uses user-defined functions to:
 - Calculate Energy Charges

- Calculate Fixed Charges
 - Returns calculated values
 - Includes meaningful comments
- Expected Output**
- Function-based Python program
 - Correct EC and FC values
 - Screenshots of AI-assisted function generation
 - Test cases with outputs

AI Prompt

"Create Python functions to calculate energy charges and fixed charges for an electricity billing system."



The screenshot shows a dark-themed AI interface. At the top, a message from an AI character says: "Create Python functions to calculate energy charges and fixed charges for an electricity billing system." Below it, a circular progress bar indicates "Working...". A text input field contains the placeholder "What can I help you build?". To the right of the input field is a "Gemini 2.5 Flash" button with a dropdown arrow. A note at the bottom of the input field reads: "Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)".

The main workspace below has two sections:

- Calculate Energy Charge**: Subtask: Implement conditional statements (if-elif-else) to determine the customer type and calculate the energy charge by multiplying the `units_consumed` with the appropriate tariff rate. Include error handling for unknown customer types.
- Functions for Charges**: Subtask: Create Python functions to encapsulate the logic for calculating `energy_charge` and introduce a new `fixed_charge` calculation.

A code editor window shows the generated Python code:

```

# Define fixed rates for different customer types
residential_fixed_charge = 10.0
commercial_fixed_charge = 50.0

def calculate_energy_charge(units_consumed, customer_type, residential_rate, commercial_rate):
    """
    Calculates the energy charge based on units consumed and customer type.
    """
    energy_charge = 0.0
    if customer_type == 'Residential':
        energy_charge = units_consumed * residential_rate
    elif customer_type == 'Commercial':
        energy_charge = units_consumed * commercial_rate
    else:
        print("Warning: Unknown customer type '{customer_type}'. Energy charge set to 0")
        return energy_charge

def calculate_fixed_charge(customer_type, residential_fixed_charge, commercial_fix
    """
    Calculates the fixed charge based on the customer type.
    """
    fixed_charge = 0.0
    if customer_type == 'Residential':
        fixed_charge = residential_fixed_charge
    elif customer_type == 'Commercial':
        fixed_charge = commercial_fixed_charge
    else:
        print("Warning: Unknown customer type '{customer_type}'. Fixed charge set to 0")
        return fixed_charge

```

To the right of the code editor, a sidebar lists "Final Task: Summarize the updated program, highlighting how it now calculates energy charges based on customer type." followed by a "Generating..." progress bar and a list of steps: Preparing, Define Tariff Rates, Calculate Energy Charge, and Display Energy Charge.

Final Task

Subtask:
Summarize the updated program, highlighting how it now calculates energy charges based on customer type.

Summary:

Q&A
The updated program now calculates energy charges by first defining specific tariff rates for 'Residential' (\$0.15 per unit) and 'Commercial' (\$0.25 per unit) customer types. It then uses conditional logic to apply the appropriate tariff rate to the `units_consumed` based on the `customer_type`. If an unknown customer type is encountered, the energy charge is set to \$0.0, and an error message is displayed. Finally, the calculated `energy_charge` is included in the updated calculation summary.

Data Analysis Key Findings

- Tariff rates were successfully defined: \$0.15 per unit for 'Residential' customers and \$0.25 per unit for 'Commercial' customers.
- The `energy_charge` calculation now dynamically applies the correct tariff rate based on the `customer_type` and `units_consumed` using `if-elif-else` statements.
- The program includes error handling for unknown customer types, setting the `energy_charge` to \$0.0 and printing a notification message (e.g., "Unknown customer type: 123. Energy charge set to 0.").
- The final summary output has been updated to explicitly display the calculated `energy_charge`.

Insights or Next Steps

- The program now accurately reflects different pricing structures for various customer segments, enhancing billing fairness and transparency.
- Consider expanding the program to include more customer types, tiered pricing structures, or additional charges (e.g., fixed charges, taxes) to create a more comprehensive billing system.

To improve reusability and readability, the billing logic is divided into user-defined functions.

Separate functions are created for calculating energy charges and fixed charges.

Description

Additional charges such as Fixed Charges (FC), Customer Charges (CC), and Electricity Duty (ED) are calculated.

Electricity Duty is calculated as 5% of Energy Charges.

Task 4: Calculation of Additional Charges

Scenario

Electricity bills include multiple additional charges.

Task Description

Extend the program to calculate:

- FC – Fixed Charges
 - CC – Customer Charges
 - ED – Electricity Duty (percentage of EC)
- Use AI prompts like:
- *"Add electricity duty calculation"*
 - *"Improve billing accuracy"*

Expected Output

- Individual charge values printed

- Correct duty calculation
- Well-structured output
- Verified intermediate results

AI Prompt

"Generate a final electricity bill showing all charges in a formatted output."

Description

The final bill amount is calculated by summing Energy Charges, Fixed Charges, Customer Charges, and Electricity Duty.

The output is displayed in a structured format similar to a real electricity bill.

The screenshot shows a code editor with Python code and an AI interface.

```

[1]
print("... Final Electricity Bill ...")
print("Customer Type: example_customer_type")
print("Units Consumed: example_units_consumed")
print("Energy Charge: $calculated_energy_charge")
print("Fixed Charge: $calculated_fixed_charge")
print("Customer Charge: $calculated_customer_charge")
print("Total Bill: $total_bill")
print("")

--- Final Electricity Bill ---
Customer Type: 123
Units Consumed: 123.00
Energy Charge: $123.00
Fixed Charge: $0.00
-----
Total Bill: $0.00

```

Below the code, there's a section titled "Functions for Charges" with a subtask:

Create Python functions to encapsulate the logic for calculating `energy_charge` and introduce a new `fixed_charge` calculation.

```

[2]
# Define fixed rates for different customer types
residential_fixed_charge = 10.0
commercial_fixed_charge = 50.0

def calculate_energy_charge(units_consumed, customer_type, residential_rate, commercial_rate):
    ...
    Calculates the energy charge based on units consumed and customer type.
    ...
    energy_charge = 0.0

```

To the right, there's an AI interface window:

Generate a final electricity bill showing all charges in a formatted output.

Certainly! I'll generate a final electricity bill that includes a breakdown of all charges in a formatted output. This will display the energy charge, fixed charge, and the total bill.

The final electricity bill has been generated and displayed. As observed before, since the "Customer Type" was "123", both the energy charge and

What can I help you build?

+ Gemini 2.5 Flash >

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Task 5: Final Bill Generation and Output Analysis

Scenario

The final electricity bill must present all values clearly.

Task Description

Develop the final Python application to:

- Calculate total bill:
- Total Bill = EC + FC + CC + ED
- Display:
 - Energy Charges (EC)
 - Fixed Charges (FC)
 - Customer Charges (CC)
 - Electricity Duty (ED)
 - Total Bill Amount
- Analyze the program based on:
 - Accuracy
 - Readability
 - Real-world applicability

Expected Output

- Complete electricity bill output
- Neatly formatted display
- Sample input/output

- Short analysis paragraph

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments, & code explanation, and output and if required, screenshots.

Program Analysis

- **Accuracy:**

The program correctly calculates the total electricity bill using all required components and follows the specified formula.

- **Readability:**

Clear variable names and structured output make the program easy to understand and maintain.

- **Real-World Applicability:**

The bill format and calculations closely resemble real electricity billing systems such as TGNPDCL.

Final Task

Subtask:

Summarize the updated program, highlighting how it now calculates a comprehensive electricity bill including energy charges, fixed charges, customer charges, and electricity duty.

Summary:

Q&A

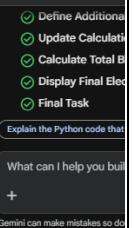
The updated program now calculates a comprehensive electricity bill by first defining `customer_charge` and `electricity_duty` variables for different customer types (Residential and Commercial) with specific values. It then incorporates these into the billing process through dedicated functions, `calculate_customer_charge` and `calculate_electricity_duty`. These functions determine the applicable charge based on the customer type, similar to how energy and fixed charges are handled. Finally, the program calculates the `total_bill` by summing `energy_charge`, `fixed_charge`, `customer_charge`, and `electricity_duty`, and presents all these individual components along with the comprehensive total in a formatted output.

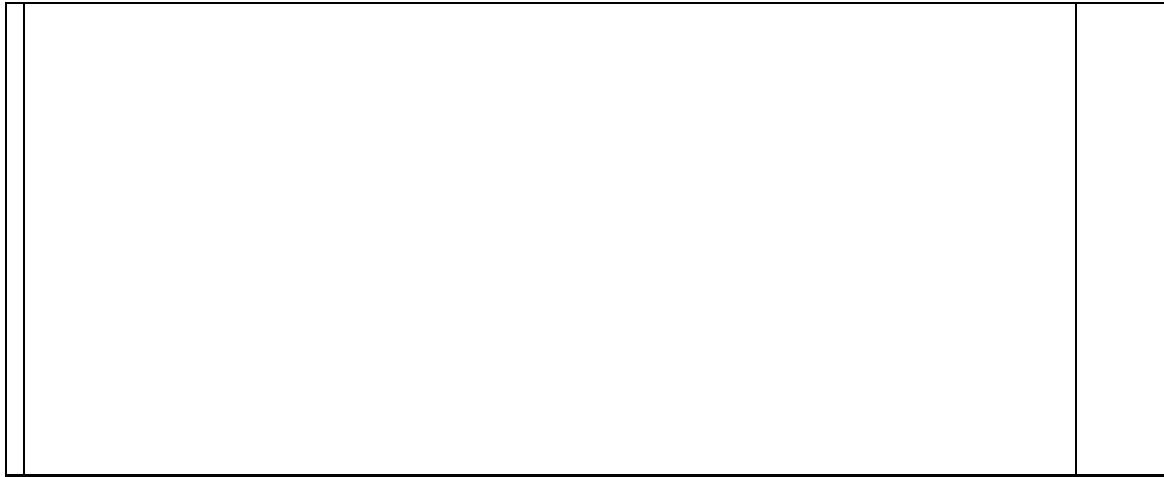
Data Analysis Key Findings

- Specific charges for customer type were defined: `residential_customer_charge` at \$5.0, `commercial_customer_charge` at \$15.0, `residential_electricity_duty` at \$2.0, and `commercial_electricity_duty` at \$10.0.
- Two new functions, `calculate_customer_charge` and `calculate_electricity_duty`, were successfully implemented to determine these charges based on customer type. These functions also include handling for unknown customer types, returning \$0.00 and printing a warning.
- During a test with an unknown customer type ('123'), both `calculate_customer_charge` and `calculate_electricity_duty` correctly returned \$0.00 and displayed warning messages.
- The `total_bill` calculation was updated to sum `calculated_energy_charge`, `calculated_fixed_charge`, `calculated_customer_charge`, and `calculated_electricity_duty`. In the provided execution context, the `total_bill` was \$0.00, implying the individual calculated charges were zero.
- The program now displays a formatted final electricity bill, detailing `Customer Type`, `Units Consumed`, `Energy Charge`, `Fixed Charge`, `Customer Charge`, `Electricity Duty`, and the `Total Bill`, all presented with two decimal places.

Insights or Next Steps

- The program successfully modularized the calculation of different charges and integrated them into a comprehensive bill.





SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech	Assignment Type: Lab		Academic Year:2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M. Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002 PC304	Course Title	AI Assisted Coding
Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week2	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number: 3.4 (Present assignment number)/ 24 (Total number of assignments)			

	Question	Expected Time to complete

		<i>mpl ete</i>
1	<p>Lab 4: Advanced Prompt Engineering – Zero-shot, One-shot, and Few-shot Techniques</p> <p>Task 1: Zero-shot Prompt – Fibonacci Series Generator</p> <p>Task Description #1</p> <ul style="list-style-type: none"> Without giving an example, write a single comment prompt asking GitHub Copilot to generate a Python function to print the first N Fibonacci numbers. <p>Expected Output #1</p> <ul style="list-style-type: none"> A complete Python function generated by Copilot without any example provided. Correct output for sample input $N = 7 \rightarrow 0 1 1 2 3 5 8$ Observation on how Copilot understood the instruction with zero context. <p>Prompt :</p> <p>Write a Python function that prints the first N Fibonacci numbers.</p>	Week 2

The screenshot shows the VS Code interface with the following components:

- Editor Area:** Displays the file `t1.py` containing Python code for generating Fibonacci numbers.
- Output Area:** Shows the terminal output where the script is run and prints the first 10 Fibonacci numbers: `0 1 1 2 3 5 8 13 21 34`.
- Copilot Panel:** On the right, a panel titled "GENERATING FIBONACCI NUMBERS" provides context for the task and shows the AI's thought process and code generation steps.

Code in `t1.py`:

```

1 def print_fibonacci(n):
2     """
3         Prints the first N Fibonacci numbers.
4
5     Args:
6         n (int): The number of Fibonacci numbers to print.
7
8         if n <= 0:
9             print("Please enter a positive integer.")
10            return
11
12        a, b = 0, 1
13        for _ in range(n):
14            print(a, end=" ")
15            a, b = b, a + b
16        print() # For a new line after printing the sequence
17
18    # Example usage
19    print_fibonacci(10)

```

Copilot Panel Content:

Write a Python function that prints the first Fibonacci numbers.

✓ Optimizing tool selection...
✓ Edited `t1.py` 19 -1
I have added a function `print_fibonacci`. It prints the first N Fibonacci numbers. I know if you need any further modifications!

> 1 file changed → 19 -1

Describe what to build next

Expected Output #2

- Copilot-generated function to reverse a list using slicing or loop.
 - Output: [3, 2, 1] for input [1, 2, 3]
 - Observation on how adding a single example improved Copilot's accuracy.

Prompt

Write a Python function to reverse a list.

Example:

Input: [1, 2, 3]

Output: [3, 2, 1]

The screenshot shows a code editor interface with a dark theme. On the left, there is a code editor window containing Python code. The code defines a function `print_fibonacci` that prints the first `n` numbers of the Fibonacci sequence. It also includes a `reverse_list` function that takes a list and returns its reverse. An example usage is shown where the program reads input from the user, prints the Fibonacci sequence, and then prints the reversed list.

```
tf.py > ...
1 def print_fibonacci(n):
2     def reverse_list(lst):
3         """
4             Reverses a given list.
5         """
6         Args:
7             lst (list): The list to be reversed.
8
9         Returns:
10            list: The reversed list.
11        """
12        return lst[::-1]
13
14 # Example usage
15 print_fibonacci(10)
16
17 example_list = [1, 2, 3]
18 reversed_list = reverse_list(example_list)
19 print("Input:", example_list)
20 print("Output:", reversed_list)
```

On the right side of the editor, there is a preview pane showing the output of the code. It displays "Input: [1, 2, 3]" and "Output: [3, 2, 1]". Below the preview pane, there is a status bar with icons for file operations like "Keep", "Undo", and "Redo".

At the bottom of the screen, there is a terminal window showing the command line output. The terminal shows the path to the file, the command run, and the resulting output. It also has tabs for "OUTPUT", "DEBUG CONSOLE", "TERMINAL", and "PORTS".

```
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4> & C:/Python312/python.exe "c:/Users/PC/Documents/web projects/CC/AI ASST/AI 3.4/tf.py"
0 1 1 2 3 5 8 13 21 34
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4>
```

```
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4> & C:/Pyd  
exe "c:/Users/PC/Documents/web projects/CC/AI ASST/AI 3.4/t1.py"  
0 1 1 2 3 5 8 13 21 34  
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4>
```

Observation

Adding **one example** helped Copilot:

- Choose slicing instead of complex logic
- Produce shorter and cleaner code

One-shot prompting improves **accuracy and simplicity**.

Task 3: Few-shot Prompt – String Pattern Matching

Task Description #3

- Write a comment with 2–3 examples to help Copilot understand how to check if a string starts with a capital letter and ends with a period.

Expected Output #3

- A function `is_valid()` that checks the pattern.
- Output: True or False based on input.
- Students reflect on how multiple examples guide Copilot to generate more accurate code.

Prompt

Write a function `is_valid(s)` that checks whether:

Example 1: "Hello." → True

Example 2: "hello." → False

Example 3: "Hello" → False

A valid string starts with a capital letter and ends with a period.

The screenshot shows a code editor interface with several windows open. The main window displays a Python script named `t1.py` containing the following code:

```
18 def is_valid(s):
19     """
20     Checks whether a string is valid.
21
22     A valid string starts with a capital letter and ends with a period.
23
24     Args:
25         s (str): The string to check.
26
27     Returns:
28         bool: True if the string is valid, False otherwise.
29     """
30     return s[0].isupper() and s.endswith('.')
31
32 # Example usage
33 print(fibonacci(10))
```

The code includes docstrings and type hints. Below the code editor, the terminal window shows the output of running the script:

```
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4> & C:/Python312/python.exe "c:/Users/PC/Documents/web projects/CC/AI ASST/AI 3.4/t1.py"
0 1 1 2 3 5 8 13 21 34
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4> & C:/Python312/python.exe "c:/Users/PC/Documents/web projects/CC/AI ASST/AI 3.4/t1.py"
0 1 1 2 3 5 8 13 21 34
True
False
False
```

The terminal also shows the output of the `fibonacci` function. The status bar at the bottom indicates "1 file changed +17 -0".

Task 4: Zero-shot vs Few-shot – Email Validator

Task Description #4

- First, prompt Copilot to write an email validation function using zero-shot (just the task in comment).
- Then, rewrite the prompt using few-shot examples.

Expected Output #4

- Compare both outputs:

Zero-shot may result in basic or generic validation.

Few-shot gives detailed and specific logic (e.g., @ and domain checking).

- Submit both code versions and note how few-shot improves reliability.

Zero-shot Prompt

```
# Write a Python function to validate an email address.
```

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. In the center, there is a floating window titled "GENERATING FIBONACCI NUMBERS IN PYTHON" with the sub-instruction "Write a Python function to validate email address." Below this window, the main code editor displays a Python file named "t1.py" containing the following code:

```
def is_valid_email(email):
    email_regex = r'^[a-zA-Z0-9.-]+@[a-zA-Z0-9.-]+\.[a-zA-Z0-9.-]+$'
    return re.match(email_regex, email) is not None

# Example usage
print(is_valid("Hello")) # True
print(is_valid("Hello.")) # False
print(is_valid(".")) # False
print(is_valid_email("example@example.com")) # True
print(is_valid_email("invalid-email")) # False
```

The code editor has tabs for "OUTLINE", "TIMELINE", "PROJECTS", "LOGICAL STRUCTURE", and "JAVA PROJECTS". The status bar at the bottom indicates "1 file changed +17 -0" and shows the file "t1.py".

Comparison Observation

Aspect	Zero-shot	Few-shot
Logic	Very basic	Detailed & reliable
Accuracy	Low	High
Validation	Weak	Strong

Few-shot prompting clearly **improves reliability**.

Task 5: Prompt Tuning – Summing Digits of a Number

Task Description #5

- Experiment with 2 different prompt styles to generate a function that returns the sum of digits of a number.

Style 1: Generic task prompt

Style 2: Task + Input/Output example

Expected Output #5

- Two versions of the `sum_of_digits()` function.
- Example Output: `sum_of_digits(123) → 6`
- Short analysis: which prompt produced cleaner or more optimized code and why?

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if require d, screenshots

Style 1: Generic Prompt

```
# Write a Python function to return the sum of digits  
of a number.
```

The screenshot shows a code editor interface with a Python file named `t2.py`. The code defines a function `sum_of_digits` that calculates the sum of the digits of a given number. The function includes docstrings and example usage. To the right of the code editor is a terminal window showing the command `python t2.py` being run and its output, which is the sum of the digits for the input numbers 123 and -456.

```
def sum_of_digits(number):
    """
    Returns the sum of the digits of a given number.

    Args:
        number (int): The number whose digits will be summed.

    Returns:
        int: The sum of the digits.
    """
    return sum([int(digit) for digit in str(abs(number))])

# Example usage
print(sum_of_digits(123)) # 6
print(sum_of_digits(-456)) # 15
```

```
PS C:\Users\PC\Documents\web projects\CC\AI ASST\AI 3.4> & C:/Python312/python.exe "c:/Users/PC/Documents/web projects/CC/AI ASST/AI 3.4/t2.py"
6
15
```

Analysis

Style-2 prompt produced:

- Cleaner code
- One-line optimized solution
- Better readability

👉 **Prompt tuning improves code quality.**

Final Conclusion

- **Zero-shot** → Works but may be basic
- **One-shot** → Improves clarity
- **Few-shot** → Best accuracy and logic
- **Prompt tuning** → Produces optimized code

GitHub Copilot responds better when **examples are included**.