

TASK-1

Task 1: Word Frequency from Text File

❖ Scenario:

You are analyzing log files for keyword frequency.

❖ Task:

Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.

❖ Expected Output:

➤ Working code

➤ Explanation

➤ Screenshot

The screenshot shows a Jupyter Notebook interface with the Gemini AI integration. The top right corner features the Gemini interface with a greeting "Hello, Nukala" and several buttons for interacting with AI, such as "How can I install Python libraries?", "Load data from Google Drive", and "Show an example of training a simple ML model". On the left side, there's a sidebar with icons for file operations like copy, paste, and search. The main workspace contains two code cells. The first cell is a text cell with the placeholder "Start coding or generate with AI...". The second cell is a code cell containing the following Python code:

```
sample_text = "This is a sample text file. This file contains sample text."
with open('sample.txt', 'w') as f:
    f.write(sample_text)

print("Created 'sample.txt' with sample content.")
```

Below the code cell, a note says: "Now let's write the Python code to read the text file, count word frequencies, and display the results. We'll use the collections.Counter module for efficient counting."

The bottom part of the screenshot shows the generated Python code for reading a file and counting word frequencies:

```
from collections import Counter
import re

def count_word_frequency(filepath):
    """Reads a text file and returns a Counter object with word frequencies.
    Args:
        filepath (str): The path to the text file.
    Returns:
        Counter: A Counter object where keys are words and values are their frequencies.
    """
    words = []
    with open(filepath, 'r') as file:
        text = file.read()
        # Convert to lowercase and remove non-alphanumeric characters
        words = re.findall(r'\b\w+\b', text.lower())
```

The bottom right corner shows the Gemini interface again, with a message "Gemini 2.5 Flash" and a "Accept and run" button.

```

# Convert to lowercase and remove non-alphanumeric characters
words = re.findall(r'\b\w+\b', text.lower())
return Counter(words)

# Specify the path to your text file
file_path = 'sample.txt' # Change this to your actual file path
word_counts = count_word_frequency(file_path)

print("Word Frequencies in '%s'" % file_path)
for word, count in word_counts.most_common():
    print("%s: %d" % (word, count))

*** Word Frequencies in 'sample.txt'
this : 2
sample : 2
text : 2
the : 2
is : 2
'a' : 1
'contains': 1

```

The code defines a function to read a text file, sample.txt, for processing. It converts the entire text to lowercase and extracts all alphanumeric words using regular expressions. The collections.Counter module is then used to efficiently count the frequency of each extracted word. The function is called with the sample.txt file, which was pre-populated with example text. Finally, the code iterates through the word counts and prints each word along with its frequency.

TASK-2

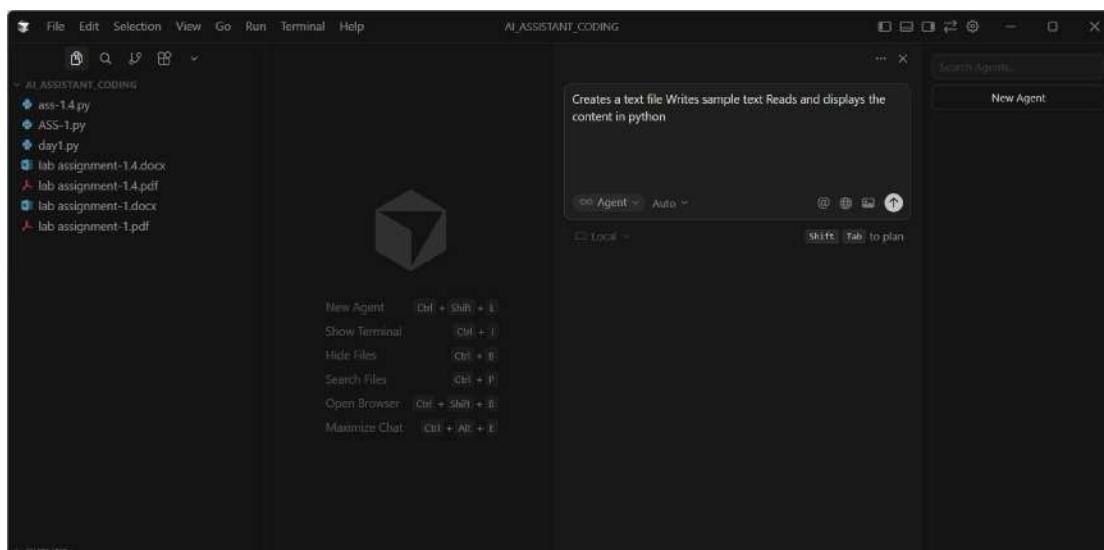
❖ Scenario:

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



```
File Edit Selection View Go Run Terminal Help AI_ASSISTANT_CODING 23.py X AI_ASSISTANT_CODING Review Next File New Agent Agents Python file operations... 3m 23.py ② Step 1: Create and write sample text to a file 3 filename = "sample.txt" 4 sample_text = """Hello, World! 5 This is a sample text file. 6 Python file operations are simple and powerful. 7 We can write, read, and manipulate files easily.*** 8 9 # Write the text to the file 10 with open(filename, 'w') as file: 11     file.write(sample_text) 12 13 print(f"\nCreated and wrote sample text to '{filename}'") 14 print() 15 16 # Step 2: Read and display the content 17 with open(filename, 'r') as file: 18     content = file.read() 19 20 print(f"Content of '{filename}':") 21 print("-" * 50) 22 print(content) 23 print("-" * 50) 24 print("-" * 50)
```

Content of 'sample.txt':

```
Hello, World!
This is a sample text file.
Python file operations are simple and powerful.
We can write, read, and manipulate files easily.
```

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

o=x8lEKrop2ZcU

Gemini

◆ Certainly! Here's a 5-line explanation of the code:

1. The code defines a function to read a text file, `sample.txt`, for processing.
2. It converts the entire text to lowercase and extracts all alphanumeric words using regular expressions.
3. The `collections.Counter` module is then used to efficiently count the frequency of each extracted word.
4. The function is called with the `sample.txt` file, which was pre-populated with example text.
5. Finally, the code iterates through the word counts and prints each word along with its frequency.

Show me the Python code to count word frequency

Run the word frequency counting code

How can I sort the word counts by frequency

read a CSV file and calculate mean, min, and max.

+ Gemini 2.5 Flash ▶

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

✓ 10:11 Python 3

10:34 AM 1/21/2026

The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains Python code to create a sample CSV file named `sample_data.csv` and calculate mean, min, and max values for a numerical column.

```
# Create a sample DataFrame
data = {'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C'],
        'Value': [10, 15, 15, 20, 10, 12, 25],
        'Quantity': [1, 5, 2, 8, 3, 2, 7]}
df_sample = pd.DataFrame(data)

# Save the DataFrame to a CSV file
df_sample.to_csv('sample_data.csv', index=False)

print("Created 'sample_data.csv' with sample content.")
print(df_sample)
```

The output cell shows the created CSV file with sample content:

	Category	Value	Quantity
0	A	10	1
1	B	15	5
2	A	15	2
3	C	20	8
4	B	10	3
5	A	12	2
6	C	25	7

The next code cell reads the `sample_data.csv` file into a pandas DataFrame and calculates the mean, min, and max for the `'Value'` column.

```
Now, let's read the 'sample_data.csv' file into a pandas DataFrame and calculate the mean, min, and max for a numerical column, for example, the 'Value' column.
```

The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains Python code to read a CSV file and calculate mean, min, and max values for a numerical column.

```
import pandas as pd

# Specify the path to your CSV file
csv_file_path = 'sample_data.csv' # Change this to your actual CSV file path

try:
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv(csv_file_path)

    print(f"Successfully loaded '{csv_file_path}'. Here's the first few rows:")
    display(df.head())

    # Select a numerical column to calculate statistics (e.g., 'Value')
    # You can change 'Value' to any numerical column in your CSV.
    numerical_column = 'Value'

    if numerical_column in df.columns and pd.api.types.is_numeric_dtype(df[numerical_column]):
        # Calculate mean, min, and max
        mean_value = df[numerical_column].mean()
        min_value = df[numerical_column].min()
        max_value = df[numerical_column].max()

        print("\nStatistics for column: '{}'\n".format(numerical_column))
        print("Mean: {:.2f}\n".format(mean_value))
        print("Min: {:.2f}\n".format(min_value))
        print("Max: {:.2f}\n".format(max_value))
    else:
        print("Unknown column: '{}' not found or is not numeric in the CSV file.\n".format(numerical_column))
        print("Available columns: ", df.columns.tolist())

```

```

print("Name: max_value=27")
print("Min: min_value=27")
print("Max: max_value=27")
else:
    print("Error: column '(numerical_column)' not found or is not numeric in the CSV file.")
    print("Available columns: ", df.columns.tolist())
except FileNotFoundError:
    print("Error: The file '{csv_file_path}' was not found. Please check the path.")
except Exception as e:
    print(f"An error occurred: {e}")

```

--> Successfully loaded 'sample_data.csv'. Here's the first few rows:

	Category	Value	Quantity
0	A	10	1
1	B	25	5
2	A	15	2
3	C	30	8
4	B	20	3

Statistics for column 'Value':

Mean: 21.00
Std: 10.00
Max: 35.00

Gemini

read a CSV file and calculate mean, min, and max.

Okay, I can help with that! Here's the Python code to read a CSV file, then calculate and display the mean, minimum, and maximum values for a numerical column. I'll include a step to create a sample CSV first.

Accept and run Accept Cancel

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 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

The screenshot shows a web browser interface with two tabs open: "New Tab" and "CSV files". The main content area displays a conversation with Gemini AI:

Gemini

N read a CSV file and calculate mean, min, and max.

◆ Okay, I can help with that! Here's the Python code to read a CSV file, then calculate and display the mean, minimum, and maximum values for a numerical column. I'll include a step to create a sample CSV first.

👉 🔍

▶ Accept and run ✓ Accept ✖ Cancel

give a python code for bubble sort

+ Gemini 2.5 Flash ▶

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

✓ 10:35 📱 Python 3

```

def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # 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] # Swap elements

    # Example usage:
    my_list = [64, 34, 25, 12, 22, 11, 90]
    print("Original list:", my_list)

    sorted_list = bubble_sort(my_list)
    print("Sorted list:", sorted_list)

    another_list = [5, 1, 4, 2, 8]
    print("Original list:", another_list)

    sorted_another_list = bubble_sort(another_list)
    print("Sorted list:", sorted_another_list)

```

... Original list: [64, 34, 25, 12, 22, 11, 90]
 Sorted list: [11, 12, 22, 25, 34, 64, 90]
 Original list: [5, 1, 4, 2, 8]
 Sorted list: [1, 2, 4, 5, 8]

N use python built in function for the same question sort()

◆ You're right! Python has very efficient built-in sorting mechanisms. Here's how you can sort lists using `list.sort()` (which sorts the list in-place) and `sorted()` (which returns a new sorted list).

built-in function returns a new sorted list without changing the original.

```

# Using list.sort() (sorts in-place)
list_to_sort_in_place = [64, 34, 25, 12, 22, 11, 90]
print("Original list (for in-place sort):", list_to_sort_in_place)
list_to_sort_in_place.sort()
print("Sorted list (in-place):", list_to_sort_in_place)

print("\n---")

# Using sorted() (returns a new sorted list)
original_list_for_sorted = [5, 1, 4, 2, 8]
print("Original list (for sorted() function):", original_list_for_sorted)
new_sorted_list = sorted(original_list_for_sorted)
print("New sorted list (from sorted() function):", new_sorted_list)
print("Original list remains unchanged:", original_list_for_sorted)

```

... Original list (for in-place sort): [64, 34, 25, 12, 22, 11, 90]
 Sorted list (in-place): [11, 12, 22, 25, 34, 64, 90]
 ... Original list (for sorted() function): [5, 1, 4, 2, 8]
 New sorted list (from sorted() function): [1, 2, 4, 5, 8]
 Original list remains unchanged: [5, 1, 4, 2, 8]

Short comparision

Bubble Sort: It's a simple sorting algorithm, but generally inefficient for large datasets ($O(n^2)$ complexity). It sorts by repeatedly stepping through the list, comparing adjacent elements and swapping them.

Python's `list.sort()` and `sorted()`: These are highly optimized and implemented in C, using Timsort (a hybrid stable sorting algorithm with $O(n \log n)$ average and worst-case complexity). They are significantly faster and more memory-efficient for most use cases. `list.sort()` sorts in-place, while `sorted()` returns a new sorted list.

Task 1: AI-Generated Logic for Reading Consumer Details

Scenario

An electricity billing system must collect accurate consumer data.

The screenshot shows a code editor with a dark theme. The file being edited is `electricity_billing.py`. The code reads previous and current unit values from user input, calculates the difference, and prints the total bill based on residential or commercial rates. A detailed code comment explains the logic and provides usage instructions.

```
# Read Previous Units (PU)
pu = float(input("Enter previous units: "))

# Read Current Units (CU)
cu = float(input("Enter current units: "))

# Calculate Units Consumed
units_consumed = cu - pu

# Display the results
print("Previous Units (PU):")
print(pu)
print("Current Units (CU):")
print(cu)
print("Units Consumed (Units Consumed):")
print(units_consumed)

# Basic Billing Calculations
if customer_type == "Residential":
    rate_per_unit = 5.0 # Example rate
elif customer_type == "Commercial":
    rate_per_unit = 10.0 # Example rate
else:
    print("Invalid customer type. Assuming residential rate.")
rate_per_unit = 5.0

total_bill = units_consumed * rate_per_unit
print("Total Bill: ", total_bill)
```

The code is saved as `electricity_billing.py`. In your workspace, when you run it, it will prompt for the inputs and display the results. As shown in the test run, it correctly calculates units consumed and a simple bill.

Task 2: Energy Charges Calculation Based on Units Consumed

Scenario

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

The screenshot shows a code editor with a dark theme. The file being edited is `electricity_billing.py`. The code reads previous and current unit values from user input, calculates the difference, and then calculates energy charges based on the customer type (Residential or Commercial). A detailed code comment explains the logic and provides usage instructions.

```
# Read Previous Units (PU)
pu = float(input("Enter previous units: "))

# Read Current Units (CU)
cu = float(input("Enter current units: "))

# Calculate Units Consumed
units_consumed = cu - pu

# Display the results
print("Previous Units (PU):")
print(pu)
print("Current Units (CU):")
print(cu)
print("Units Consumed (Units Consumed):")
print(units_consumed)

# Basic Billing Calculations
if customer_type == "Residential":
    energy_charge = units_consumed * 5.0 # Example rate
elif customer_type == "Commercial":
    energy_charge = units_consumed * 10.0 # Example rate
else:
    print("Invalid customer type. Assuming residential rate.")
energy_charge = units_consumed * 5.0

# Total Billing
total_bill = energy_charge + 40.0 # Fixed charge

print("Total Bill: ", total_bill)
```

The code is saved as `electricity_billing.py`. In your workspace, when you run it, it will prompt for the inputs and display the results. As shown in the test run, it correctly calculates units consumed and energy charges.

```

# Electricity Billing System Using Functions
# This program calculates electricity bills using modular functions for energy, charges and fixed charges.

def calculate_energy_charge(units_consumed, customer_type):
    """
    Calculate energy charges based on customer type and units consumed.
    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Calculated energy charge
    """
    if customer_type == "Residential":
        # Residential billing price: $0.0 for first 300, $0.0 for next 200, $0.0 thereafter
        if units_consumed <= 300:
            return 0.0
        elif units_consumed <= 500:
            return 0.0 + (units_consumed - 300) * 0.0
        else:
            return 100 * 0.0 + 200 * 0.0 + (units_consumed - 500) * 0.0
    elif customer_type == "Commercial":
        # Commercial billing: $0.0 for first 100, $0.0 thereafter
        if units_consumed <= 100:
            return 0.0
        else:
            return 100 * 0.0 + (units_consumed - 100) * 0.0
    elif customer_type == "Industrial":
        # Industrial billing: $0.0 for first 500, $0.0 thereafter
        if units_consumed <= 500:
            return 0.0
        else:
            return 500 * 0.0 + (units_consumed - 500) * 0.0
    else:
        raise ValueError("Customer type not specified")

def calculate_fixed_charge(customer_type):
    """
    Calculate fixed charge based on customer type.
    Args:
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Fixed charge
    """
    if customer_type == "Residential":
        return 10.0
    elif customer_type == "Commercial":
        return 20.0
    elif customer_type == "Industrial":
        return 50.0
    else:
        raise ValueError("Customer type not specified")

def calculate_total_charge(units_consumed, customer_type):
    """
    Calculate total bill amount.
    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Total bill amount
    """
    energy_charge = calculate_energy_charge(units_consumed, customer_type)
    fixed_charge = calculate_fixed_charge(customer_type)
    total_charge = energy_charge + fixed_charge
    return total_charge

```

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

Scenario

Billing logic must be reusable for multiple consumers.

```

# Electricity Billing System Using User-Defined Functions
# This program calculates electricity bills using modular functions for energy, charges and fixed charges.

def calculate_energy_charge(units_consumed, customer_type):
    """
    Calculate energy charges based on customer type and units consumed.
    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Calculated energy charge
    """
    if customer_type == "Residential":
        # Residential billing price: $0.0 for first 300, $0.0 for next 200, $0.0 thereafter
        if units_consumed <= 300:
            return 0.0
        elif units_consumed <= 500:
            return 0.0 + (units_consumed - 300) * 0.0
        else:
            return 100 * 0.0 + 200 * 0.0 + (units_consumed - 500) * 0.0
    elif customer_type == "Commercial":
        # Commercial billing: $0.0 for first 100, $0.0 thereafter
        if units_consumed <= 100:
            return 0.0
        else:
            return 100 * 0.0 + (units_consumed - 100) * 0.0
    elif customer_type == "Industrial":
        # Industrial billing: $0.0 for first 500, $0.0 thereafter
        if units_consumed <= 500:
            return 0.0
        else:
            return 500 * 0.0 + (units_consumed - 500) * 0.0
    else:
        raise ValueError("Customer type not specified")

def calculate_fixed_charge(customer_type):
    """
    Calculate fixed charge based on customer type.
    Args:
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Fixed charge
    """
    if customer_type == "Residential":
        return 10.0
    elif customer_type == "Commercial":
        return 20.0
    elif customer_type == "Industrial":
        return 50.0
    else:
        raise ValueError("Customer type not specified")

def calculate_total_charge(units_consumed, customer_type):
    """
    Calculate total bill amount.
    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('Residential', 'Commercial', 'Industrial')
    Returns:
        float: Total bill amount
    """
    energy_charge = calculate_energy_charge(units_consumed, customer_type)
    fixed_charge = calculate_fixed_charge(customer_type)
    total_charge = energy_charge + fixed_charge
    return total_charge

```

Task 4: Calculation of Additional Charges

Scenario

Electricity bills include multiple additional charges.

Task 5: Final Bill Generation and Output Analysis

Scenario

The final electricity bill must present all values clearly.

This program accurately calculates the electricity bill by using basic arithmetic formulas. The code is easy to read because of meaningful variable names and clear print statements. It is applicable in real-world situations as it follows the standard electricity billing structure used by power departments. The formatted output helps users understand each charge clearly.

Assignment-3.4

Task 1: Zero-shot Prompt – Fibonacci Series Generator

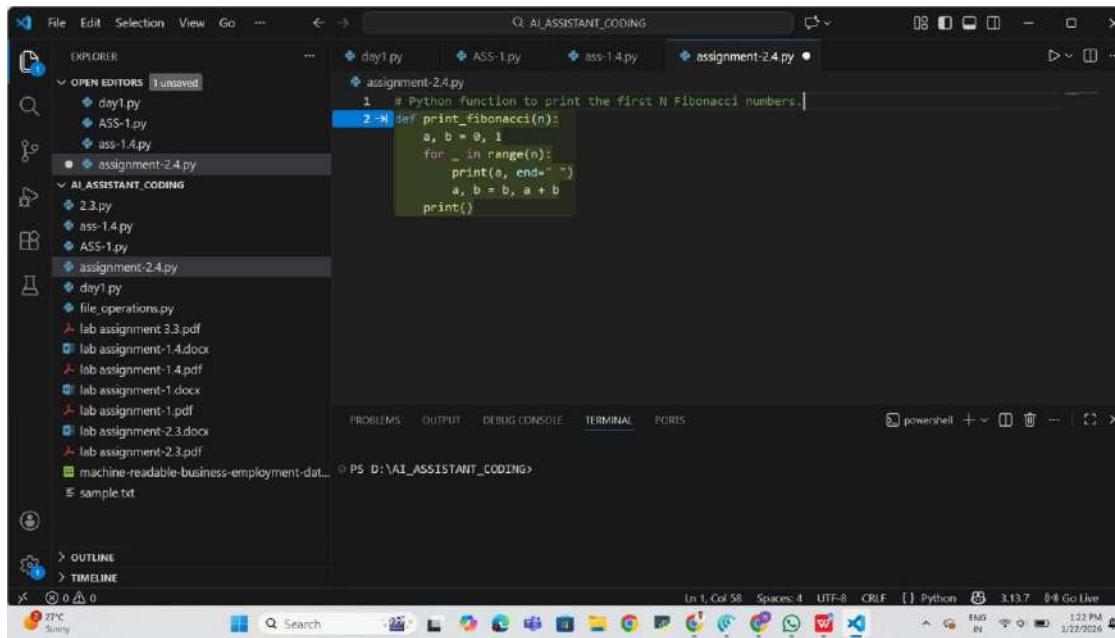
Task Description #1

- 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:



```
# Python function to print the first N Fibonacci numbers.
def print_fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b
    print()
```

CODE:

```
def fibonacci_sequence(n):
    """
    Generates the first N Fibonacci numbers.

    Args: n: Integer, number of Fibonacci numbers to generate

    Returns:
        List of first N Fibonacci numbers
    """
    fib_sequence
    = []
    a, b = 0, 1 for _ in range(n):
        fib_sequence.append(a) a, b = b, a +
        b return fib_sequence for i in
        range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
```

```

        return True
    return True # Test the function
num = int(input("Enter the number of Fibonacci numbers to generate: "))
fib_numbers = fibonacci_sequence(num)
print(f"The first {num} Fibonacci numbers are: {fib_numbers}")

```

OUTPUT :

The screenshot shows a terminal window with the following text:

```

PS D:\AT_ASSISTANT_CODING> & "C:/Users/Achanta/sai_teja/AppData/Local/Programs/Python/Python313/python.exe" ./AI_ASSISTANT_CODING/assignment-2.4.py
PS D:\AT_ASSISTANT_CODING> & "C:/Users/Achanta/sai_teja/AppData/Local/Programs/Python/Python313/python.exe" ./AI_ASSISTANT_CODING/assignment-2.4.py
PS D:\AT_ASSISTANT_CODING> & "C:/Users/Achanta/sai_teja/AppData/Local/Programs/Python/Python313/python.exe" ./AI_ASSISTANT_CODING/assignment-2.4.py
Enter the number of Fibonacci numbers to generate: 7
The first 7 Fibonacci numbers are: [0, 1, 1, 2, 3, 5, 8]
PS D:\AT_ASSISTANT_CODING>

```

Task2: 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.

PROMPT:

The screenshot shows a code editor with the following text:

```

23.py
ass-1.4.py
ASS-1.py
assignment-2.4.py
day1.py
file_operations.py
lab assignment 3.pdf

36 fib_numbers = fibonacci_sequence(num)
37 print(f"The first {num} Fibonacci numbers are: {fib}
38
39 # Reverse a list by using an example of your choice.

```

CODE:

```

# Reverse a list by using an example of your choice
def reverse_list(input_list): """
    Reverses the given list.

    Args: input_list: List of elements to be
          reversed

    Returns:
        List: Reversed list
    """
    return input_list[::-1] # Test
the reverse_list function sample_list =
[1, 2, 3] reversed_list =
reverse_list(sample_list)
print(f"Original list: {sample_list}")
print(f"Reversed list: {reversed_list}")

```

OUTPUT:

```

lab assignment-1.pdf
lab assignment-2.3.docx
lab assignment-2.3.pdf
machine-readable-business-employment.dat
sample.txt

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS + powershell
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti/sai_teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
Original list: [1, 2, 3, 4, 5]
Reversed list: [5, 4, 3, 2, 1]

PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti/sai_teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
Original list: [1, 2, 3]
Reversed list: [3, 2, 1]

PS D:\AI_ASSISTANT_CODING>

```

In 45, Col 41 [462 selected] Spaces: 4 UTF-8 CRLF {} Python 3.13.7 8:9 Go Live

Adding just one clear example helps Copilot quickly infer:

what the function should do the

expected input and output format

This improves **accuracy and reduces ambiguity**, even with a minimal prompt.

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:

```

45
46
47 # Write a function is_valid(s) that returns True or False.
48 # The string is valid if:
49 # 1) It starts with a capital letter (A-Z)
50 # 2) It ends with a period (.)
51 #
52 # Examples:
53 # is_valid("Hello.") -> True
54 # is_valid("Hello") -> False (does not start with a capital letter)
55 # is_valid("Hello") -> False (does not end with a period)

```

CODE:

```

def is_valid(s):
    """
    Checks if the string starts with a capital letter and ends with a period.

    Args: s: Input string to validate
    Returns:
        Boolean: True if valid, False otherwise
    """
    if len(s) == 0:
        return False
    return s[0].isupper() and s[-1] == '.'
# Test the is_valid function
test_strings = ["Hello.", "hello.", "Hello", ""]
for string in test_strings:
    result = is_valid(string)
    print(f"is_valid({string}) → {result}")

```

OUTPUT:

The screenshot shows the VS Code interface with the terminal tab active. The terminal window displays the following text:

```

PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
is_valid("Hello.") → True
is_valid("hello.") → False
is_valid("Hello") → False
is_valid("") → False
PS D:\AI_ASSISTANT_CODING>

```

Providing **multiple examples in comments** helps Copilot clearly understand:

what counts as valid

what should be rejected

This leads to **more accurate and relevant code generation.**

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.

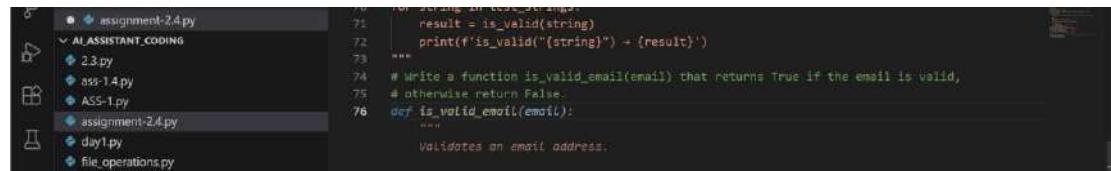
Few-shot gives detailed and specific logic (e.g., @ and domain

checking).

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

reliability.

PROMPT:



The screenshot shows a code editor interface with a sidebar containing a file tree. The tree includes files like assignment-2.3.py, 2.3.py, 855-14.py, ASS-1.py, assignment-2.4.py, day1.py, and file_operations.py. The main pane displays a Python script named assignment-2.4.py. The script contains a function definition for is_valid_email. The code is annotated with comments explaining its purpose and logic.

```
for string in test_emails:
    result = is_valid(string)
    print(f'is_valid("{string}") -> {result}')

# Write a function is_valid_email(email) that returns True if the email is valid,
# otherwise return False.
def is_valid_email(email):
    """
    Validates an email address.
    """

CODE:
```

CODE:

```
def is_valid_email(email):
    """
    Validates an email address based on basic criteria.

    Args: email: String, email address to validate

    Returns:
        Boolean: True if valid, False otherwise
    """
    # Basic checks for presence of '@' and '.'
    if "@" not in email or "." not in email:
        return False

    # Split the email into local and domain parts
    local_part, domain_part = email.split("@", 1)

    # Local part should not be empty
    if len(local_part) == 0:
        return False

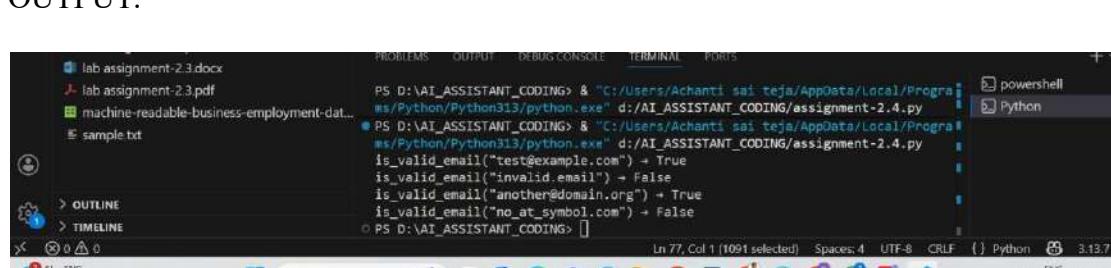
    # Domain part should contain at least one dot
    if "." not in domain_part:
        return False

    # Domain part should not start or end with a dot if
    if domain_part.startswith(".") or domain_part.endswith("."):
        return False

    return True

# Test the is_valid_email function
test_emails = ["test@example.com", "invalid.email", "another@domain.org",
    "no_at_symbol.com"] for
email in test_emails:
    result = is_valid_email(email)
    print(f'is_valid_email("{email}") -> {result}')


OUTPUT:
```



The screenshot shows a terminal window within a code editor. It displays the command PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py. Below the command, the script's output is shown, testing various email addresses and printing their validation results.

```
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
is_valid_email("test@example.com") -> True
is_valid_email("invalid.email") -> False
is_valid_email("another@domain.org") -> True
is_valid_email("no_at_symbol.com") -> False
```

PROMPT:



```
112     # Write a function is_valid_email(email) that checks whether an email is valid.
113     # Rules:
114     # - Must contain exactly one '@'
115     # - Must have characters before and after '@'
116     # - Must end with a valid domain containing '.'
117     #
118     # Examples:
119     # is_valid_email("user@example.com") -> True
120     # is_valid_email("user@example.com") -> False
121     # is_valid_email("user@com") -> False
122     # is_valid_email("user@pcom") -> False
```

CODE:

```
def is_valid_email(email):
    if email.count("@") != 1:
        return False

    local, domain = email.split("@")
    if not local or "." not in domain:
        return False
    return True

# Test the is_valid_email function test_emails =
["user@example.com", "userexample.com", "user@com"] for email in
test_emails:
    result = is_valid_email(email)
    print(f'is_valid_email("{email}") -> {result}')
```

OUTPUT:



```
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
is_valid_email("no_at_symbol.com") -> False
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
is_valid_email("user@example.com") -> True
is_valid_email("userexample.com") -> False
is_valid_email("user@com") -> False
PS D:\AI_ASSISTANT_CODING>
```

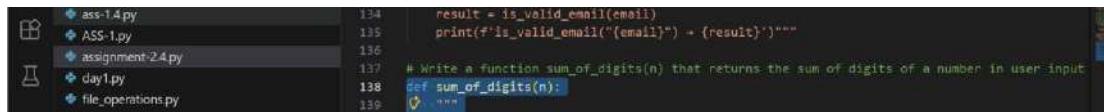
Few-shot prompting guides Copilot's reasoning by showing:

valid vs invalid patterns

edge cases

This results in **more reliable, accurate, and meaningful code** compared to zero-shot prompts.

PROMPT:



```
134     result = is_valid_email(email)
135     print(f'is_valid_email("{email}") -> {result}''')
136
137     # Write a function sum_of_digits(n) that returns the sum of digits of a number in user input
138     def sum_of_digits(n):
139         """
```

CODE:

```

def sum_of_digits(n):
    """
    Calculates the sum of digits of a given number.

    Args: n: Integer whose digits are to be summed

    Returns:
        Integer: Sum of the digits
    """
    return sum(int(digit) for digit in str(abs(n)))

# Test the sum_of_digits function
number = int(input("Enter a number to calculate the sum of its digits: "))
result = sum_of_digits(number)
print(f"The sum of the digits of {number} is: {result}")

```

OUTPUT:

```

is_valid_email("user@example.com") -> False
is_valid_email("user@com") -> False
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
Enter a number to calculate the sum of its digits: 123
The sum of the digits of 123 is: 6
PS D:\AI_ASSISTANT_CODING>

```

In 13B, Col 1 (471 selected) Spaces: 4 UTF-8 CRLF [] Python

PROMPT:

```

154 """
155 # Write a function sum_of_digits(n) that returns the sum of all digits in a number.
156 # Example:
157 # sum_of_digits(123) -> 6

```

CODE:

```

154 """
155 # Write a function sum_of_digits(n) that returns the sum of all digits in a number.
156 # Example:
157 # sum_of_digits(123) -> 6
158 def sum_of_digits(n):
159     return sum(int(digit) for digit in str(abs(n)))
160 # Test the sum_of_digits function
161 test_numbers = [123]
162 for number in test_numbers:
163     result = sum_of_digits(number)
164     print(f"sum_of_digits({number}) -> {result}")
165

```

OUTPUT:

```

is_valid_email("user@example.com") -> False
is_valid_email("user@com") -> False
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
Enter a number to calculate the sum of its digits: 123
The sum of the digits of 123 is: 6
PS D:\AI_ASSISTANT_CODING>

```

In 161, Col 21 Spaces: 4 UTF-8 CRLF [] Python 3.13.2 89 Go Live 150 MB

