

TASK-1

Task 1: Word Frequency from Text File

❖ Scenario:

You are analyzing log files for keyword frequency.

❖ Task:

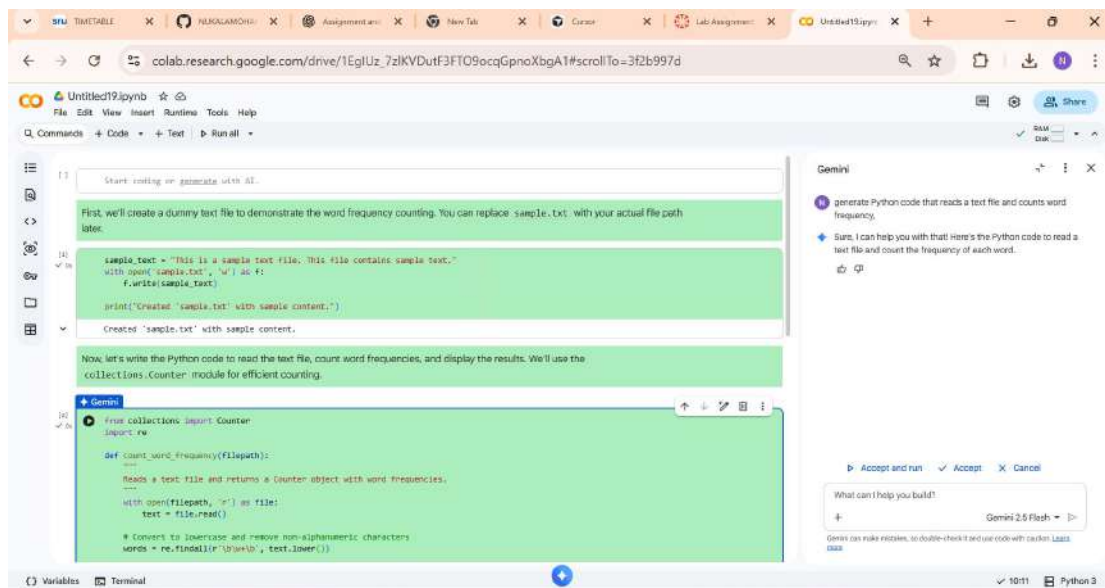
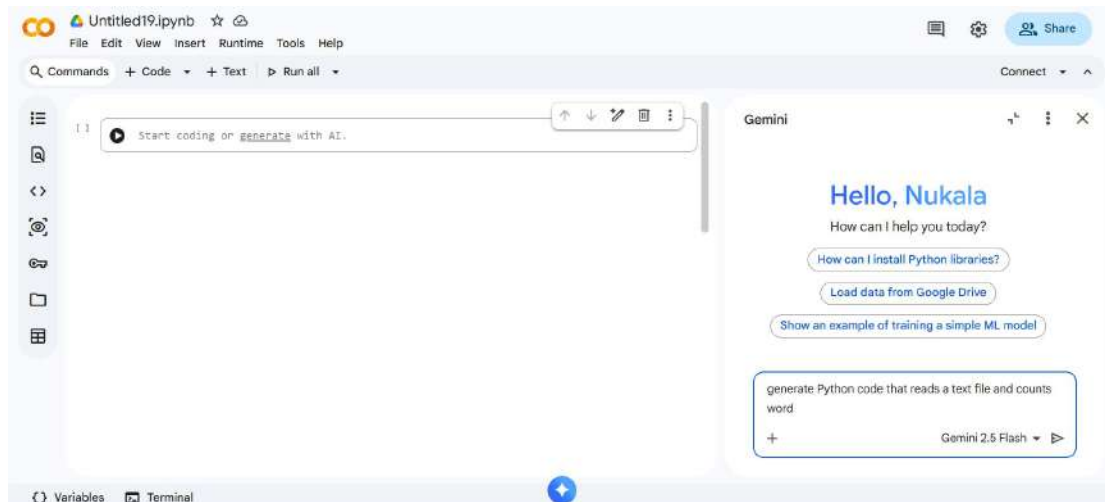
Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.

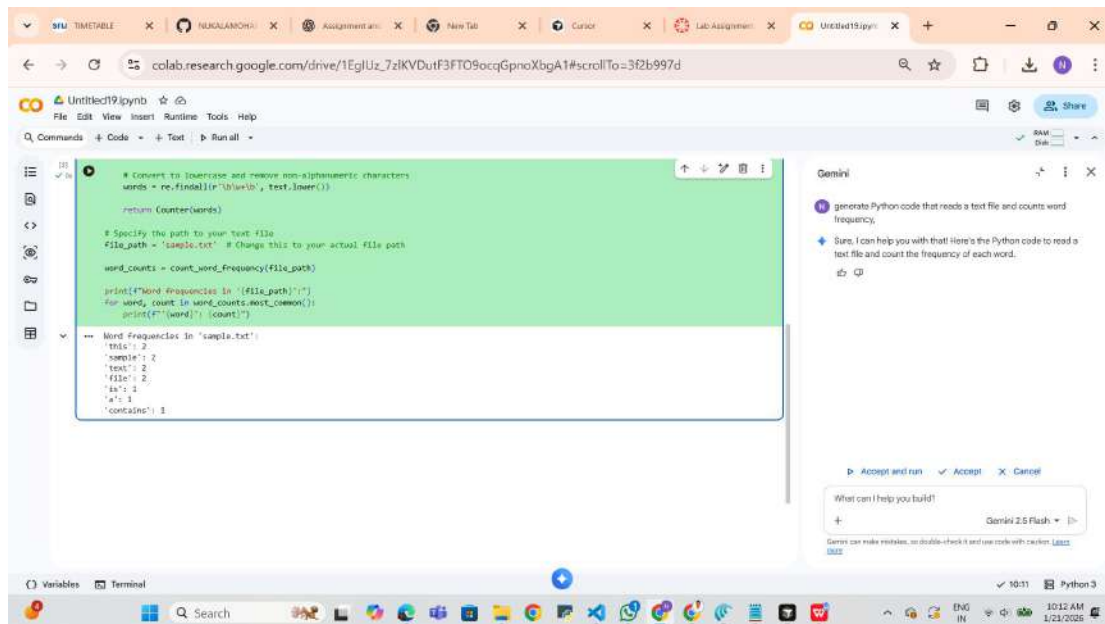
❖ Expected Output:

➤ Working code

➤ Explanation

➤ Screenshot





The 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.

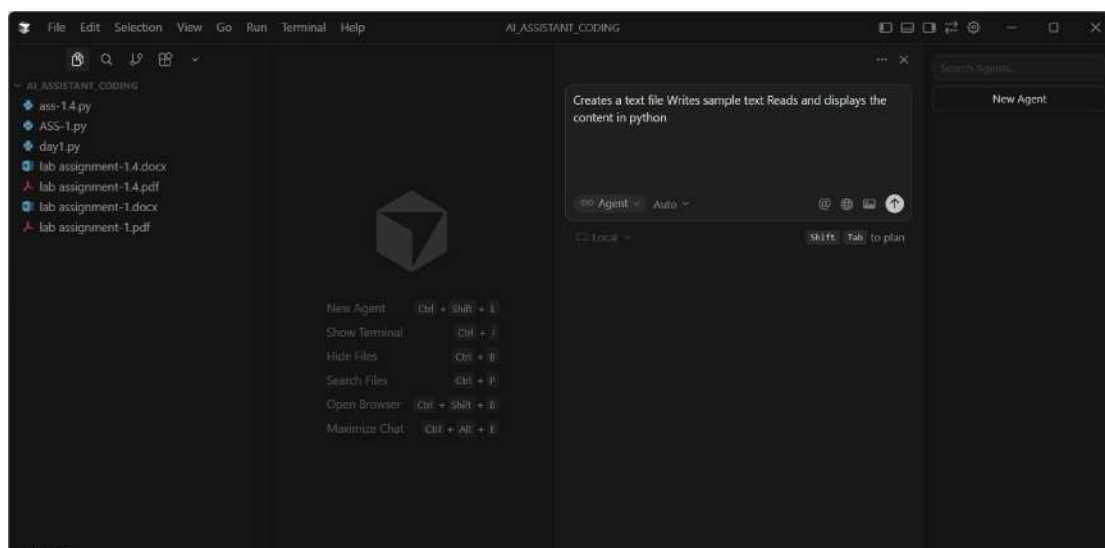
❖ Task:

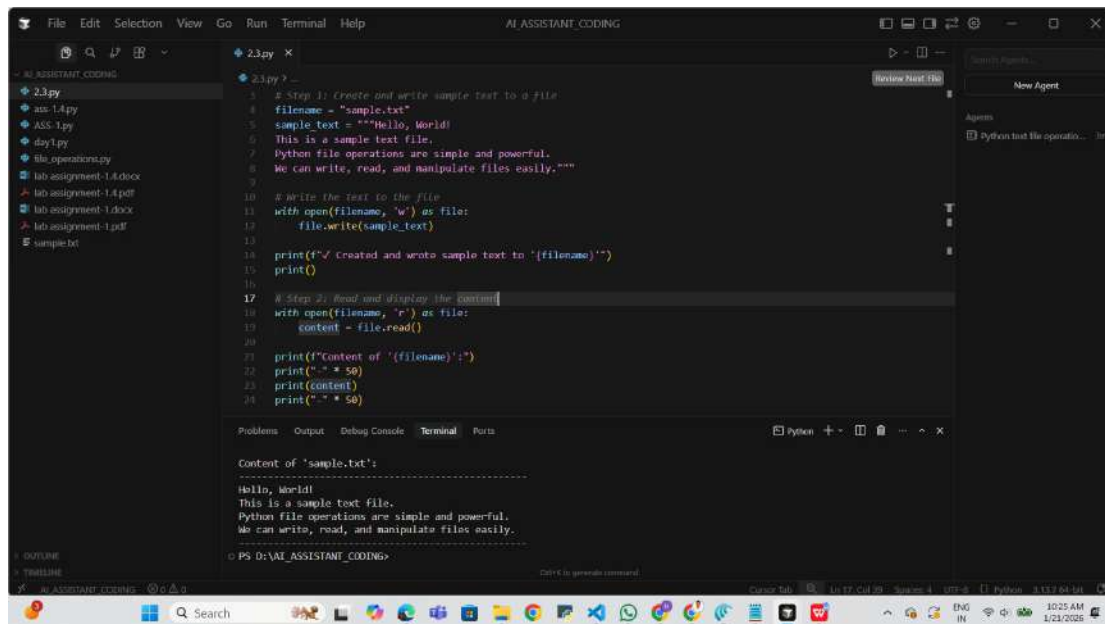
Use Cursor AI to generate a program that:

- Creates a text file
- Writes sample text
- Reads and displays the content

❖ Expected Output:

- Functional code
- Cursor AI screenshots





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

Share

RAM
Disk

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

ENG
IN

10:34 AM
1/21/2026

colab.research.google.com/drive/1EglUz_7zIKVDutF3FTO9ocqGpnoXbgA1#scrollTo=fcc15417

Untitled19.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text + Run all

Files

- .
- config
- sample_data
- machine-readable-business-em...
- sample.txt
- sample_data.csv

Start coding or generate with AI.

First, let's create a sample CSV file named `sample_data.csv` that we can use for demonstration. It will contain some numerical data.

```
import pandas as pd

# Create a sample DataFrame
data = {'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C'],
        'Value': [10, 25, 15, 30, 20, 12, 35],
        'Quantity': [1, 2, 3, 4, 2, 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)
```

Created 'sample_data.csv' with sample content.

Category	Value	Quantity
A	10	1
B	25	2
A	15	3
C	30	4
B	20	2
A	12	2
C	35	7

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.

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

Variables Terminal

10:35 AM 1/21/2025

colab.research.google.com/drive/1EglUz_7zIKVDutF3FTO9ocqGpnoXbgA1#scrollTo=fcc15417

Untitled19.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text + Run all

Files

- .
- config
- sample_data
- machine-readable-business-em...
- sample.txt
- sample_data.csv

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.

```
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(f"Statistics for column '{numerical_column}':")
        print(f"Mean: {mean_value:.2f}")
        print(f"Min: {min_value:.2f}")
        print(f"Max: {max_value:.2f}")
    else:
        print(f"Error: Column '{numerical_column}' not found or is not numeric in the CSV file.")
        print(f"Available columns: ", df.columns.tolist())
```

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.

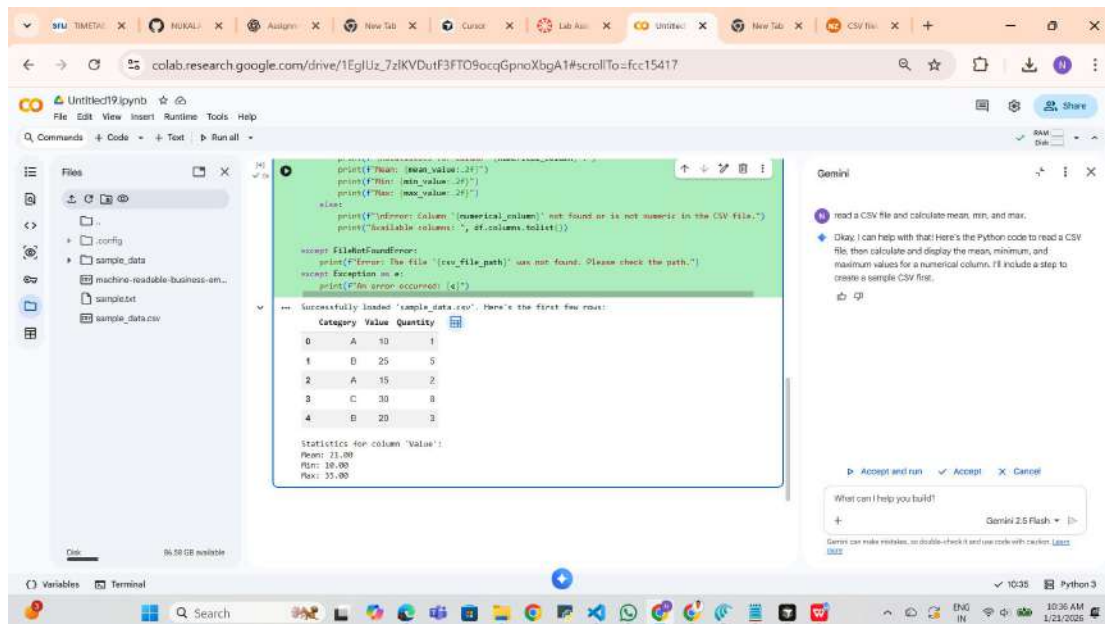
Accept and run Accept Cancel

What can I help you build?

Gemini 2.5 Flash

Variables Terminal

10:35 AM 1/21/2025



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

New Tab x

NZ CSV files x

+

—

x

🔍

☆

↓

N

⋮

💬

⚙️

👤 Share

✓

RAM

Disk

▾

▲

🗑️

⋮

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

...

ENC

10:38 AM

The screenshot shows a Google Colab notebook titled 'Untitled19.ipynb'. The code defines a custom bubble sort function named `bubble_sort(arr)`. The function takes a list `arr` and returns a new sorted list. It includes comments explaining the logic: traversing through all array elements, checking if the current element is greater than the next, and swapping them if necessary. The function is tested with two example lists: `my_list = [64, 34, 25, 12, 22, 11, 90]` and `another_list = [5, 1, 4, 2, 8]`. The output shows the original and sorted versions of both lists.

```
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
    return arr

# 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)
```

Output:

```
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]
```

- 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).

The screenshot shows a Google Colab notebook comparing two built-in sorting methods: `list.sort()` (in-place sorting) and `sorted()` (returns a new sorted list). The code demonstrates how `list.sort()` modifies the original list, while `sorted()` creates a new sorted list without changing the original. The output shows the original and sorted versions of two lists, confirming that the original list remains unchanged when using `sorted()`.

```
# 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)
```

Output:

```
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 comparison

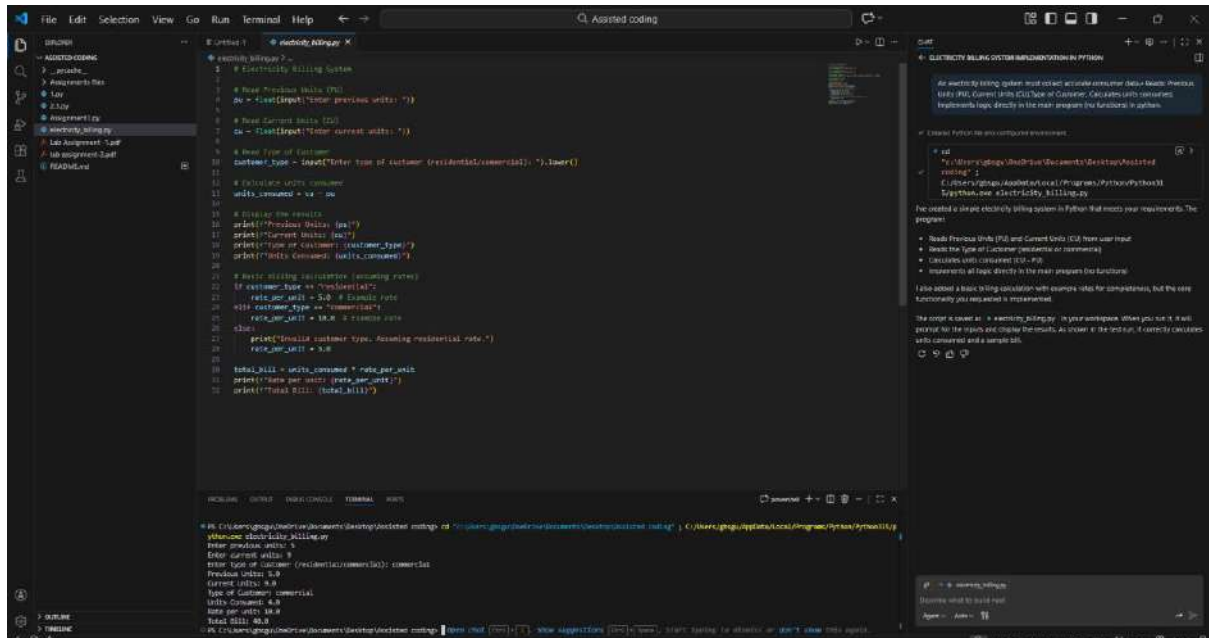
Bubble Sort: It's a simple sorting algorithm, but generally inefficient for large datasets ($O(n^2)$ complexity). It sorts by repeatedly stepping through the list, comparing adjacent elements and swapping them.

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.

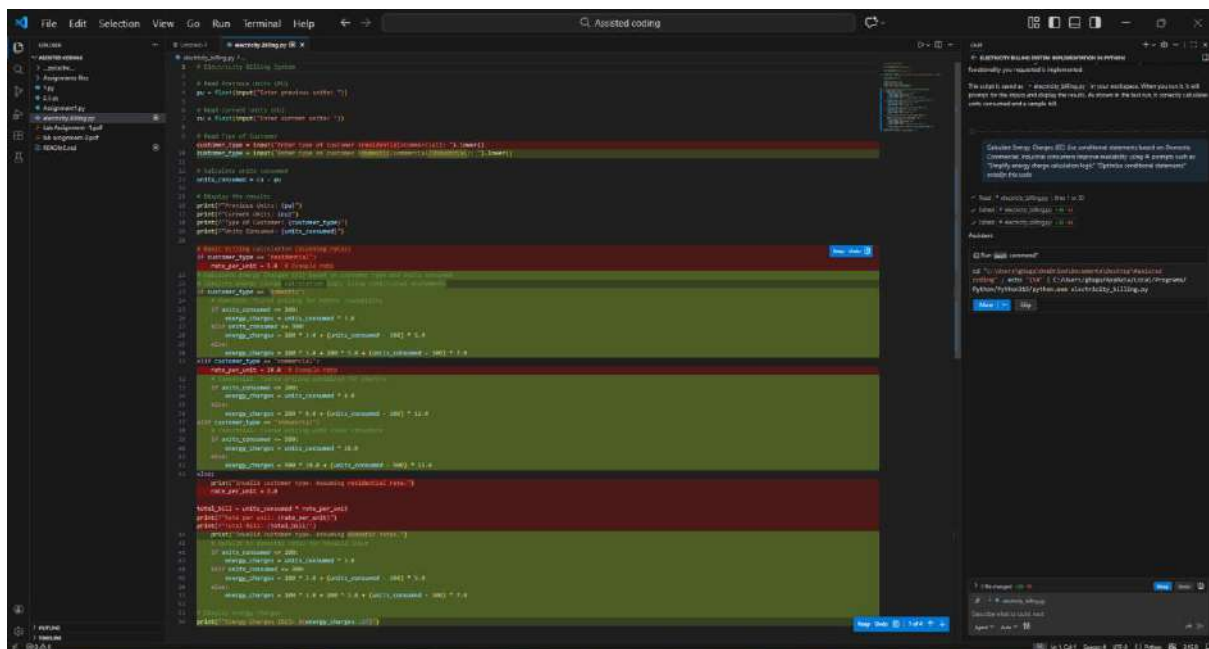


```
1 # Electricity Billing System
2
3 # Read Previous Units (PU)
4 pu = float(input("Enter previous units: "))
5
6 # Read Current Units (CU)
7 cu = float(input("Enter current units: "))
8
9 # Read Type of Customer
10 customer_type = input("Enter type of customer (residential/commercial): ").lower()
11
12 # Calculate units consumed
13 units_consumed = cu - pu
14
15 # Display the results
16 print(f"Previous Units: {pu}")
17 print(f"Current Units: {cu}")
18 print(f"Type of customer: {customer_type}")
19 print(f"Units Consumed: {units_consumed}")
20
21 # Basic billing calculation (assuming rates)
22 if customer_type == "residential":
23     rate_per_unit = 5.0 # Residential rate
24 elif customer_type == "commercial":
25     rate_per_unit = 10.0 # Commercial rate
26 else:
27     print("Invalid customer type. Assuming residential rate.")
28     rate_per_unit = 5.0
29
30 total_bill = units_consumed * rate_per_unit
31 print(f"Total per unit: {rate_per_unit}")
32 print(f"Total Bill: {total_bill}")
```

Task 2: Energy Charges Calculation Based on Units Consumed

Scenario

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



```
1 # Electricity Billing System
2
3 # Read Previous Units (PU)
4 pu = float(input("Enter previous units: "))
5
6 # Read Current Units (CU)
7 cu = float(input("Enter current units: "))
8
9 # Read Type of Customer
10 customer_type = input("Enter type of customer (residential/commercial): ").lower()
11
12 # Calculate units consumed
13 units_consumed = cu - pu
14
15 # Display the results
16 print(f"Previous Units: {pu}")
17 print(f"Current Units: {cu}")
18 print(f"Type of customer: {customer_type}")
19 print(f"Units Consumed: {units_consumed}")
20
21 # Basic billing calculation (assuming rates)
22 if customer_type == "residential":
23     rate_per_unit = 5.0 # Residential rate
24 elif customer_type == "commercial":
25     rate_per_unit = 10.0 # Commercial rate
26 else:
27     print("Invalid customer type. Assuming residential rate.")
28     rate_per_unit = 5.0
29
30 total_bill = units_consumed * rate_per_unit
31 print(f"Total per unit: {rate_per_unit}")
32 print(f"Total Bill: {total_bill}")
```

```
# electricity_billing.py
# Electricity Billing System
# This program calculates electricity bills using modular functions for energy charges and fixed charges.

def calculate_energy_charges(units_consumed, customer_type):
    """Calculate energy charges based on customer type and units consumed.
    Uses tiered pricing structure for different consumption levels.

    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('domestic', 'commercial', 'industrial')

    Returns:
        float: Calculated energy charges
    """
    # Domestic tiered pricing: 3.0 for first 100, 5.0 for next 200, 7.0 thereafter
    if customer_type == "domestic":
        if units_consumed <= 100:
            return units_consumed * 3.0
        elif units_consumed <= 300:
            return 100 * 3.0 + (units_consumed - 100) * 5.0
        else:
            return 100 * 3.0 + 200 * 5.0 + (units_consumed - 300) * 7.0
    elif customer_type == "commercial":
        # Commercial tiered pricing: 6.0 for first 200, 12.0 thereafter
        if units_consumed <= 200:
            return units_consumed * 6.0
        else:
            return 200 * 6.0 + (units_consumed - 200) * 12.0
    elif customer_type == "industrial":
        # Industrial tiered pricing: 10.0 for first 500, 15.0 thereafter
        if units_consumed <= 500:
            return units_consumed * 10.0
        else:
            return 500 * 10.0 + (units_consumed - 500) * 15.0
    else:
        # Default to commercial rates for invalid customer types
        print(f"Invalid customer type: {customer_type}. Using default rates.")
        if units_consumed <= 200:
            return units_consumed * 6.0
        else:
            return 200 * 6.0 + (units_consumed - 200) * 12.0

# Main program logic
units_consumed = 350
customer_type = "domestic"

# Calculate energy charges
energy_charges = calculate_energy_charges(units_consumed, customer_type)

# Calculate fixed charges (e.g., meter rental, distribution charges)
fixed_charges = 10.0

# Total bill
total_bill = energy_charges + fixed_charges

# Display results
print(f"Customer Type: {customer_type}")
print(f"Units Consumed: {units_consumed}")
print(f"Energy Charges: {energy_charges}")
print(f"Fixed Charges: {fixed_charges}")
print(f"Total Bill: {total_bill}")
```

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

Scenario

Billing logic must be reusable for multiple consumers.

```
# electricity_billing.py
# Electricity Billing System with User-Defined Functions
# This program calculates electricity bills using modular functions for energy charges and fixed charges.

def calculate_energy_charges(units_consumed, customer_type):
    """Calculate energy charges based on customer type and units consumed.
    Uses tiered pricing structure for different consumption levels.

    Args:
        units_consumed (float): Number of units consumed
        customer_type (str): Type of customer ('domestic', 'commercial', 'industrial')

    Returns:
        float: Calculated energy charges
    """
    # Domestic tiered pricing: 3.0 for first 100, 5.0 for next 200, 7.0 thereafter
    if customer_type == "domestic":
        if units_consumed <= 100:
            return units_consumed * 3.0
        elif units_consumed <= 300:
            return 100 * 3.0 + (units_consumed - 100) * 5.0
        else:
            return 100 * 3.0 + 200 * 5.0 + (units_consumed - 300) * 7.0
    elif customer_type == "commercial":
        # Commercial tiered pricing: 6.0 for first 200, 12.0 thereafter
        if units_consumed <= 200:
            return units_consumed * 6.0
        else:
            return 200 * 6.0 + (units_consumed - 200) * 12.0
    elif customer_type == "industrial":
        # Industrial tiered pricing: 10.0 for first 500, 15.0 thereafter
        if units_consumed <= 500:
            return units_consumed * 10.0
        else:
            return 500 * 10.0 + (units_consumed - 500) * 15.0
    else:
        # Default to commercial rates for invalid customer types
        print(f"Invalid customer type: {customer_type}. Using default rates.")
        if units_consumed <= 200:
            return units_consumed * 6.0
        else:
            return 200 * 6.0 + (units_consumed - 200) * 12.0

# Main program logic
units_consumed = 350
customer_type = "domestic"

# Calculate energy charges
energy_charges = calculate_energy_charges(units_consumed, customer_type)

# Calculate fixed charges (e.g., meter rental, distribution charges)
fixed_charges = 10.0

# Total bill
total_bill = energy_charges + fixed_charges

# Display results
print(f"Customer Type: {customer_type}")
print(f"Units Consumed: {units_consumed}")
print(f"Energy Charges: {energy_charges}")
print(f"Fixed Charges: {fixed_charges}")
print(f"Total Bill: {total_bill}")
```

```
File Edit Selection View Go Run Terminal Help
electricity_bill.py
def calculate_energy_charges(units_consumed, customer_type):
    """Calculate energy charges based on units consumed and customer type.
    """
    # Base rate per unit
    base_rate = 10.0

    # Calculate energy charges
    energy_charges = units_consumed * base_rate

    # Add customer-specific charges
    if customer_type == 'domestic':
        # Domestic customers get a 5% discount
        energy_charges *= 0.95
    elif customer_type == 'commercial':
        # Commercial customers get a 10% surcharge
        energy_charges *= 1.10
    elif customer_type == 'industrial':
        # Industrial customers get a 15% surcharge
        energy_charges *= 1.15

    return energy_charges

def calculate_fixed_charges(customer_type):
    """Calculate fixed charges based on customer type.
    """
    # Fixed charges are monthly service fees
    if customer_type == 'domestic':
        # Domestic customers pay a fixed charge of $5.00
        fixed_charges = 5.00
    elif customer_type == 'commercial':
        # Commercial customers pay a fixed charge of $10.00
        fixed_charges = 10.00
    elif customer_type == 'industrial':
        # Industrial customers pay a fixed charge of $20.00
        fixed_charges = 20.00

    return fixed_charges

def main():
    """Main function to handle user input and display billing information.
    """
    # Get user input
    units_consumed = float(input("Enter units consumed: "))
    customer_type = input("Enter customer type (domestic/commercial/industrial): ")

    # Calculate charges
    energy_charges = calculate_energy_charges(units_consumed, customer_type)
    fixed_charges = calculate_fixed_charges(customer_type)

    # Calculate total bill
    total_bill = energy_charges + fixed_charges

    # Display results
    print(f"Electricity Bill: {total_bill}")
    print(f"Units Consumed: {units_consumed}")
    print(f"Customer Type: {customer_type}")
    print(f"Energy Charges: {energy_charges}")
    print(f"Fixed Charges: {fixed_charges}")
    print(f"Total Bill: {total_bill}")

    # Return calculated values as a tuple
    return energy_charges, fixed_charges, total_bill

if __name__ == "__main__":
    main()
```

1. In calculate_energy_charges, I've added a customer_type parameter. A function that calculates energy charges based on units consumed and customer type (domestic, commercial, industrial).

2. In calculate_fixed_charges, I've added a customer_type parameter. A function that returns fixed monthly service fees based on customer type.

3. In main, I've added a customer_type input. Call the calculate functions, display results, and return the calculated values as a tuple.

Code Structure:

- Modular Design: Separated calculations into reusable functions.
- Meaningful Comments: Added docstrings and inline comments explaining each function's purpose and logic.
- Error Handling: Included feedback to domestic rates for invalid customer types.
- Return Values: Functions return calculated amounts as specified.
- Clear Output: Formatted all display with charges and total.

The code runs successfully without errors and produces accurate billing calculations. The functions are well-documented and follow Python best practices for readability and maintainability.

```
File Edit Selection View Go Run Terminal Help
electricity_bill.py
def main():
    """Main function to handle user input and display billing information.
    """
    # Get user input
    units_consumed = float(input("Enter units consumed: "))
    customer_type = input("Enter customer type (domestic/commercial/industrial): ")

    # Calculate charges
    energy_charges = calculate_energy_charges(units_consumed, customer_type)
    fixed_charges = calculate_fixed_charges(customer_type)

    # Calculate total bill
    total_bill = energy_charges + fixed_charges

    # Display results
    print(f"Electricity Bill: {total_bill}")
    print(f"Units Consumed: {units_consumed}")
    print(f"Customer Type: {customer_type}")
    print(f"Energy Charges: {energy_charges}")
    print(f"Fixed Charges: {fixed_charges}")
    print(f"Total Bill: {total_bill}")

    # Return calculated values as a tuple
    return energy_charges, fixed_charges, total_bill

if __name__ == "__main__":
    main()
```

1. In calculate_energy_charges, I've added a customer_type parameter. A function that calculates energy charges based on units consumed and customer type (domestic, commercial, industrial).

2. In calculate_fixed_charges, I've added a customer_type parameter. A function that returns fixed monthly service fees based on customer type.

3. In main, I've added a customer_type input. Call the calculate functions, display results, and return the calculated values as a tuple.

Code Structure:

- Modular Design: Separated calculations into reusable functions.
- Meaningful Comments: Added docstrings and inline comments explaining each function's purpose and logic.
- Error Handling: Included feedback to domestic rates for invalid customer types.
- Return Values: Functions return calculated amounts as specified.
- Clear Output: Formatted all display with charges and total.

The code runs successfully without errors and produces accurate billing calculations. The functions are well-documented and follow Python best practices for readability and maintainability.

Task 4: Calculation of Additional Charges

Scenario

Electricity bills include multiple additional charges.

```
File Edit Selection View Go Run Terminal Help
Assisted coding

# Electricity Billing System (Simplified Calculation)
# User Input
current_usage = float(input("Enter current usage (kWh): "))
tariff_rate = float(input("Enter tariff rate ($/kWh): "))
base_fee = float(input("Enter base fee ($): "))

# Calculate Energy Charges
energy_charges = current_usage * tariff_rate

# Calculate Demand Charges
demand_charges = 0
if current_usage > 100:
    demand_charges = (current_usage - 100) * 5.0

# Calculate Service Charges
service_charges = 0
if current_usage > 100:
    service_charges = (current_usage - 100) * 0.5

# Calculate Total Charges
total_charges = energy_charges + demand_charges + service_charges + base_fee

# Display Results
print(f"Current Usage: {current_usage} kWh")
print(f"Tariff Rate: {tariff_rate} $/kWh")
print(f"Base Fee: {base_fee} $")
print(f"Energy Charges: {energy_charges} $")
print(f"Demand Charges: {demand_charges} $")
print(f"Service Charges: {service_charges} $")
print(f"Total Charges: {total_charges} $")

# User Input
current_usage = float(input("Enter current usage (kWh): "))
tariff_rate = float(input("Enter tariff rate ($/kWh): "))
base_fee = float(input("Enter base fee ($): "))

# Calculate Energy Charges
energy_charges = current_usage * tariff_rate

# Calculate Demand Charges
demand_charges = 0
if current_usage > 100:
    demand_charges = (current_usage - 100) * 5.0

# Calculate Service Charges
service_charges = 0
if current_usage > 100:
    service_charges = (current_usage - 100) * 0.5

# Calculate Total Charges
total_charges = energy_charges + demand_charges + service_charges + base_fee

# Display Results
print(f"Current Usage: {current_usage} kWh")
print(f"Tariff Rate: {tariff_rate} $/kWh")
print(f"Base Fee: {base_fee} $")
print(f"Energy Charges: {energy_charges} $")
print(f"Demand Charges: {demand_charges} $")
print(f"Service Charges: {service_charges} $")
print(f"Total Charges: {total_charges} $")
```

```
File Edit Selection View Go Run Terminal Help
Assisted coding

# Electricity Billing System (Simplified Calculation)
# User Input
current_usage = float(input("Enter current usage (kWh): "))
tariff_rate = float(input("Enter tariff rate ($/kWh): "))
base_fee = float(input("Enter base fee ($): "))

# Calculate Energy Charges
energy_charges = current_usage * tariff_rate

# Calculate Demand Charges
demand_charges = 0
if current_usage > 100:
    demand_charges = (current_usage - 100) * 5.0

# Calculate Service Charges
service_charges = 0
if current_usage > 100:
    service_charges = (current_usage - 100) * 0.5

# Calculate Total Charges
total_charges = energy_charges + demand_charges + service_charges + base_fee

# Display Results
print(f"Current Usage: {current_usage} kWh")
print(f"Tariff Rate: {tariff_rate} $/kWh")
print(f"Base Fee: {base_fee} $")
print(f"Energy Charges: {energy_charges} $")
print(f"Demand Charges: {demand_charges} $")
print(f"Service Charges: {service_charges} $")
print(f"Total Charges: {total_charges} $")

# User Input
current_usage = float(input("Enter current usage (kWh): "))
tariff_rate = float(input("Enter tariff rate ($/kWh): "))
base_fee = float(input("Enter base fee ($): "))

# Calculate Energy Charges
energy_charges = current_usage * tariff_rate

# Calculate Demand Charges
demand_charges = 0
if current_usage > 100:
    demand_charges = (current_usage - 100) * 5.0

# Calculate Service Charges
service_charges = 0
if current_usage > 100:
    service_charges = (current_usage - 100) * 0.5

# Calculate Total Charges
total_charges = energy_charges + demand_charges + service_charges + base_fee

# Display Results
print(f"Current Usage: {current_usage} kWh")
print(f"Tariff Rate: {tariff_rate} $/kWh")
print(f"Base Fee: {base_fee} $")
print(f"Energy Charges: {energy_charges} $")
print(f"Demand Charges: {demand_charges} $")
print(f"Service Charges: {service_charges} $")
print(f"Total Charges: {total_charges} $")
```

Task 5: Final Bill Generation and Output Analysis

Scenario

The final electricity bill must present all values clearly.


```
File Edit Selection View Go Run Terminal Help
electric_bill.py
1 # Billing logic
2 def calculate_bill(customer_type, meter_reading, electricity_rate):
3     # Base charges
4     base_charges = 0.0
5     if customer_type == "residential":
6         base_charges = 25.0
7     elif customer_type == "commercial":
8         base_charges = 50.0
9     else:
10         base_charges = 0.0
11
12     # Usage charges
13     usage_charges = meter_reading * electricity_rate
14
15     # Total bill
16     total_bill = base_charges + usage_charges
17
18     # Display the bill details
19     print(f"--- Meter Reading Summary ---")
20     print(f"Meter Reading (kWh): {meter_reading}")
21     print(f"Customer Type: {customer_type}")
22     print(f"Electricity Rate: {electricity_rate}")
23     print(f"Base Charges: {base_charges}")
24     print(f"Usage Charges: {usage_charges}")
25     print(f"Total Bill: {total_bill}")
26
27     # Additional charges (e.g., late payment fee)
28     late_fee = 0.0
29     if total_bill > 100.0:
30         late_fee = 10.0
31
32     # Final bill amount
33     final_bill = total_bill + late_fee
34
35     # Print the final bill
36     print(f"--- Final Bill Summary ---")
37     print(f"Base Charges: {base_charges}")
38     print(f"Usage Charges: {usage_charges}")
39     print(f"Late Fee: {late_fee}")
40     print(f"Total Bill: {final_bill}")
41
42 # Main function
43 def main():
44     # Get user input
45     customer_type = input("Enter customer type (residential/commercial): ")
46     meter_reading = float(input("Enter meter reading (kWh): "))
47     electricity_rate = float(input("Enter electricity rate ($/kWh): "))
48
49     # Calculate the bill
50     calculate_bill(customer_type, meter_reading, electricity_rate)
51
52 # Run the program
53 if __name__ == "__main__":
54     main()
```

```
1 # Billing logic
2 def calculate_bill(customer_type, meter_reading, electricity_rate):
3     # Base charges
4     base_charges = 0.0
5     if customer_type == "residential":
6         base_charges = 25.0
7     elif customer_type == "commercial":
8         base_charges = 50.0
9     else:
10         base_charges = 0.0
11
12     # Usage charges
13     usage_charges = meter_reading * electricity_rate
14
15     # Total bill
16     total_bill = base_charges + usage_charges
17
18     # Display the bill details
19     print(f"--- Meter Reading Summary ---")
20     print(f"Meter Reading (kWh): {meter_reading}")
21     print(f"Customer Type: {customer_type}")
22     print(f"Electricity Rate: {electricity_rate}")
23     print(f"Base Charges: {base_charges}")
24     print(f"Usage Charges: {usage_charges}")
25     print(f"Total Bill: {total_bill}")
26
27     # Additional charges (e.g., late payment fee)
28     late_fee = 0.0
29     if total_bill > 100.0:
30         late_fee = 10.0
31
32     # Final bill amount
33     final_bill = total_bill + late_fee
34
35     # Print the final bill
36     print(f"--- Final Bill Summary ---")
37     print(f"Base Charges: {base_charges}")
38     print(f"Usage Charges: {usage_charges}")
39     print(f"Late Fee: {late_fee}")
40     print(f"Total Bill: {final_bill}")
41
42 # Main function
43 def main():
44     # Get user input
45     customer_type = input("Enter customer type (residential/commercial): ")
46     meter_reading = float(input("Enter meter reading (kWh): "))
47     electricity_rate = float(input("Enter electricity rate ($/kWh): "))
48
49     # Calculate the bill
50     calculate_bill(customer_type, meter_reading, electricity_rate)
51
52 # Run the program
53 if __name__ == "__main__":
54     main()
```

```
Enter customer type (residential/commercial): residential
Enter meter reading (kWh): 150
Enter electricity rate ($/kWh): 0.12

--- Meter Reading Summary ---
Meter Reading (kWh): 150
Customer Type: residential
Electricity Rate: 0.12
Base Charges: 25.0
Usage Charges: 18.0
Total Bill: 43.0

--- Final Bill Summary ---
Base Charges: 25.0
Usage Charges: 18.0
Late Fee: 0.0
Total Bill: 43.0
```

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

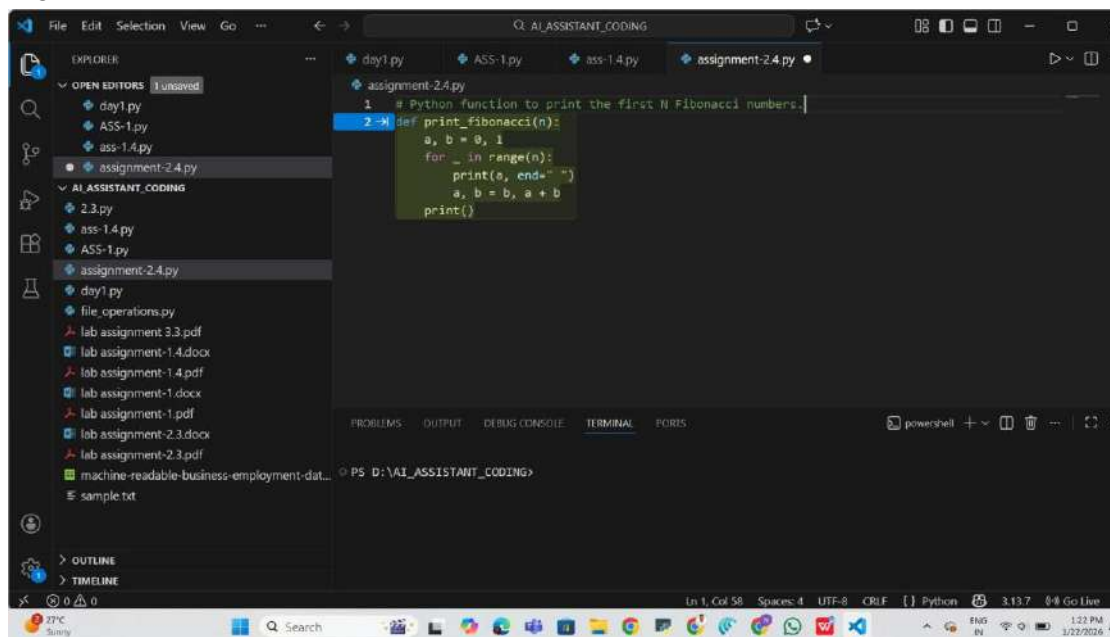
Task Description #1

- 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 → 0 1 1 2 3 5 8
- Observation on how Copilot understood the instruction with zero Context

PROMPT:



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 :

```

PS D:\AI_ASSISTANT_CODING> & "C:/Users/Ashanti sal teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Ashanti sal teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Ashanti sal teja/AppData/Local/Programs/Python/Python313/python.exe" d:/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:\AI_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:

```

26 fib_numbers = fibonacci_sequence(num)
27 print(f"The first {num} Fibonacci numbers are: {fib
28
29 # Reverse a list by using an example of your choice.

```

CODE:

```

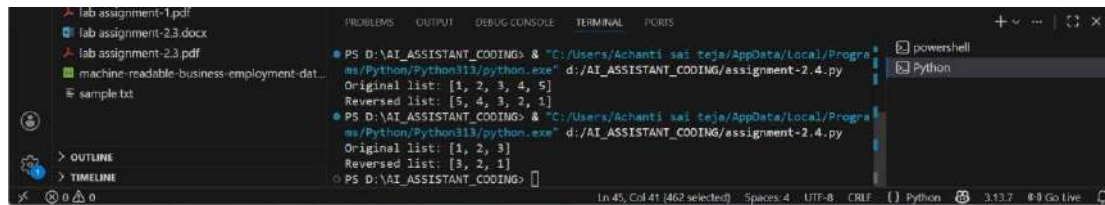
# Reverse a list by using an example of your choice
def reverse_list(input_list): """
    Reverses the given list.

    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:



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

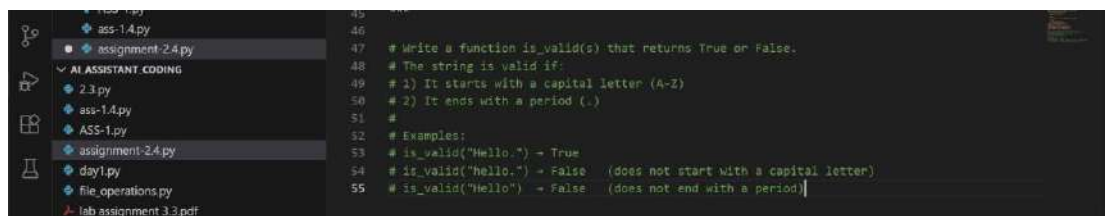
Task Description #3

- 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:



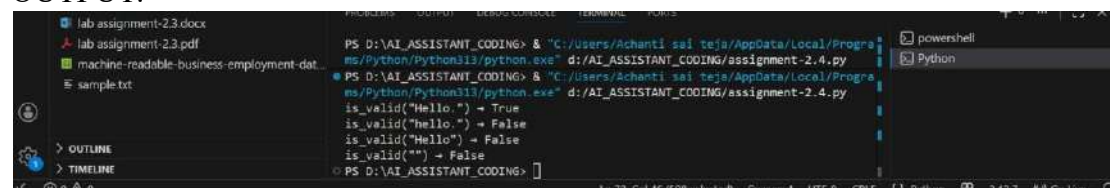
CODE:

```
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:



```
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:

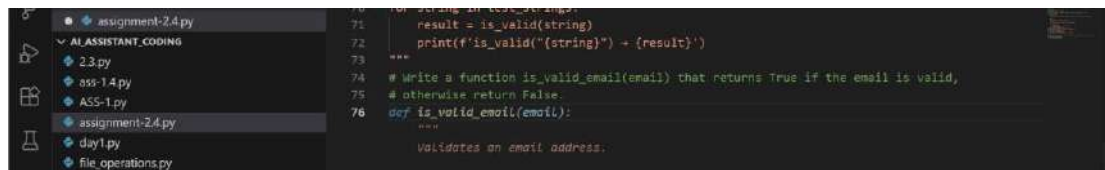
Zero-shot may result in basic or generic validation.

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

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

reliability.

PROMPT:



```
70 for string in test_strings:
71     result = is_valid(string)
72     print(f'is_valid("{string}") -> {result}')
73
74 # Write a function is_valid_email(email) that returns True if the email is valid,
75 # otherwise return False.
76 def is_valid_email(email):
77     """
78     Validates an email address.
79     """
```

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)

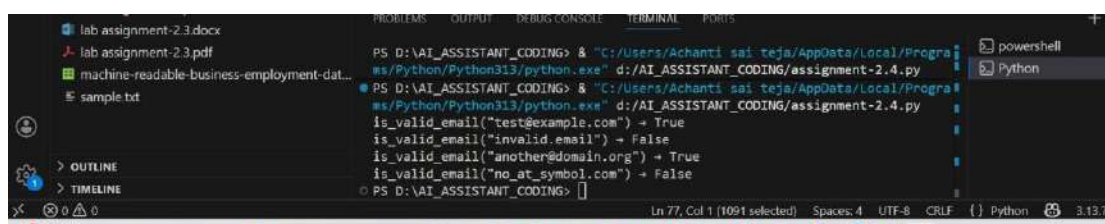
    # Local part should not be empty
    if len(local_part) == 0:
        return False

    # 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
    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:



```
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
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
PS D:\AI_ASSISTANT_CODING>
```


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 # Examples:
118 # is_valid_email("user@example.com") → True
119 # is_valid_email("userexample.com") → False
120 # is_valid_email("user@com") → False
121
122
```

CODE:

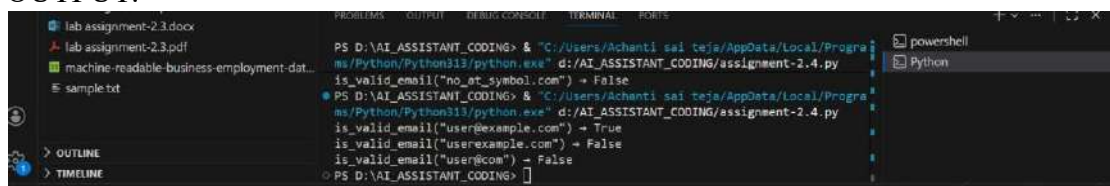
```
def is_valid_email(email):
    if email.count("@") != 1:
        return False

    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>
```

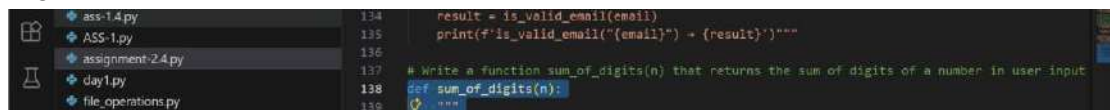
Few-shot prompting **guides Copilot's reasoning** by showing:

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}')
```

```
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>
```

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:

```
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:

```
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> "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/assignment-2.4.py
sum_of_digits(123) → 6
PS D:\AI_ASSISTANT_CODING>
```

