AI ASSISTED CODING

Lab Assignment - 2.3

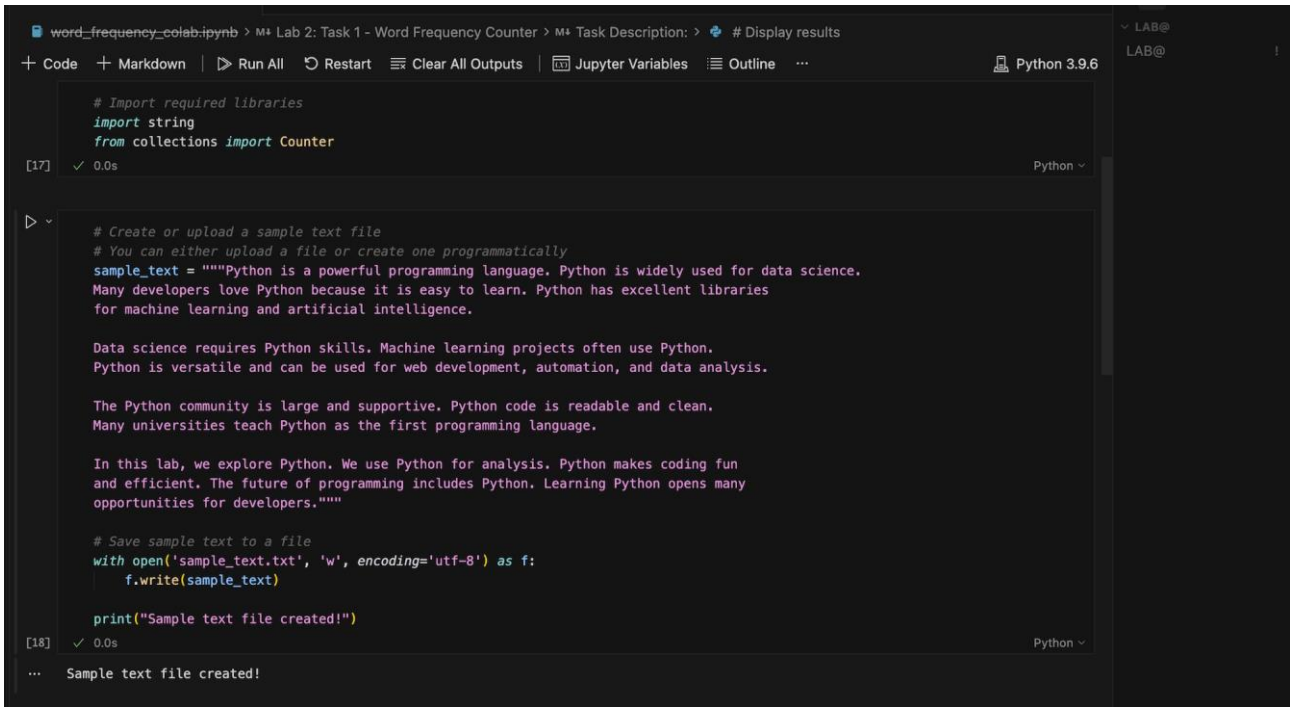
Name: Aman Sarkar
Hall Ticket: 2303A51273
Batch: 05

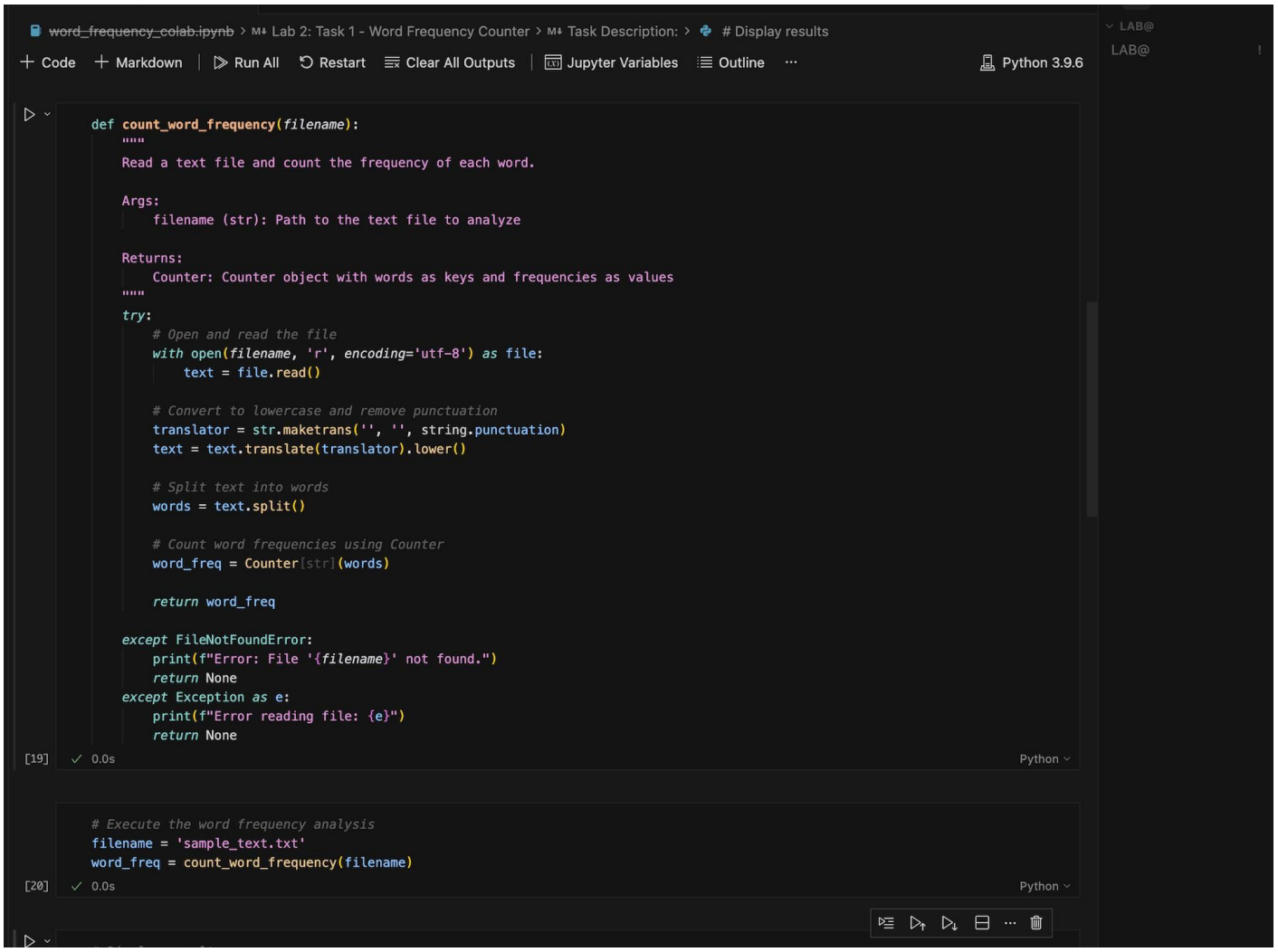
Q) Task 1: Word Frequency from Text File
❖ Scenario:
You are analyzing log files for keyword frequency.
❖ Task:
Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.
❖ Expected Output:
➢ Working code
➢ Explanation
➢ Screenshot

Solution:

PROMPT
Generate a Python program in Google Colab that reads a text file and counts the frequency of each word.

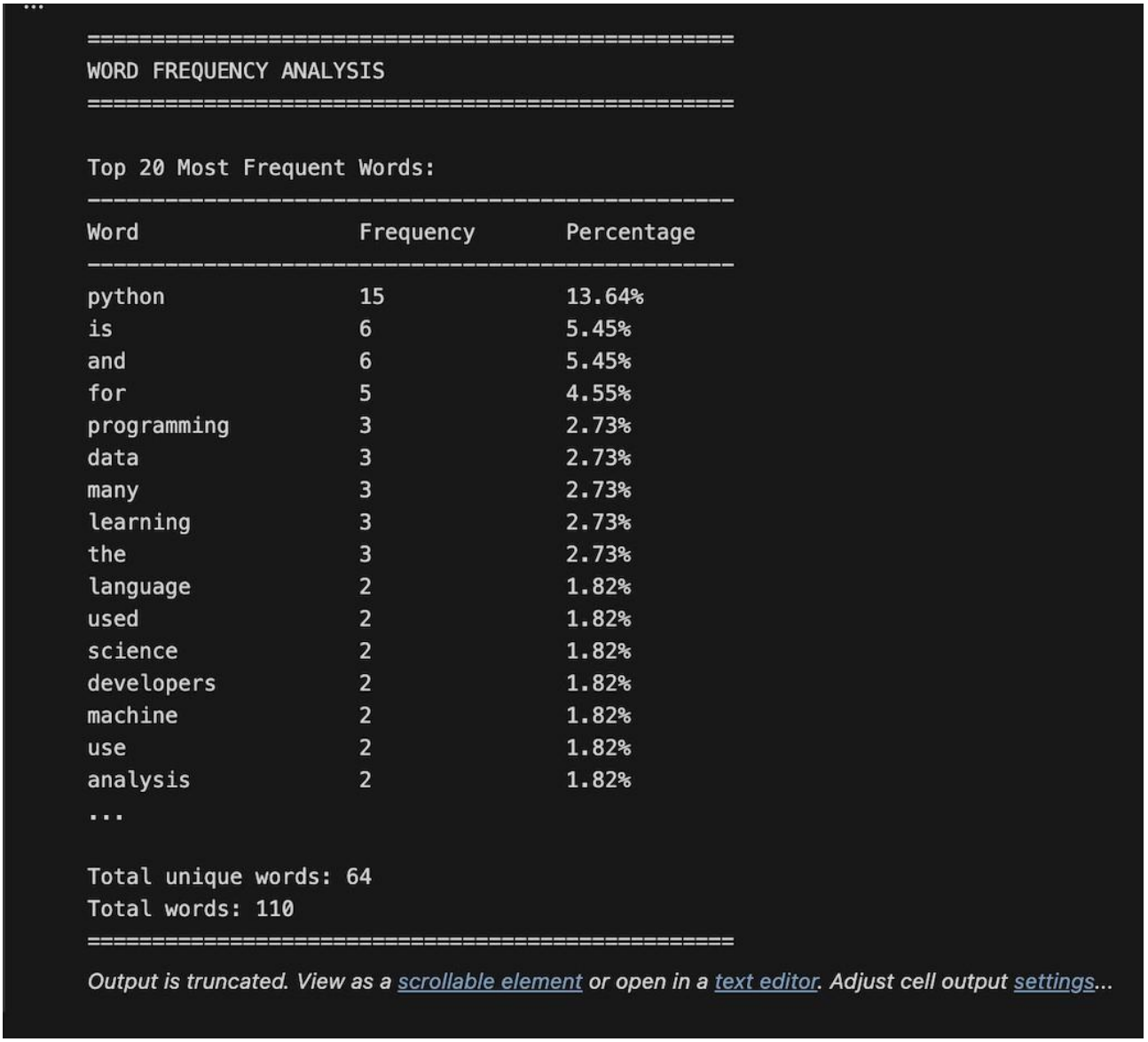
CODE:




word_frequency_colab.ipynb > Lab 2: Task 1 - Word Frequency Counter > Task Description: > # Display results
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
Python 3.9.6
def count_word_frequency(filename):
"""
Read a text file and count the frequency of each word.
Args:
filename (str): Path to the text file to analyze
Returns:
Counter: Counter object with words as keys and frequencies as values
"""
try:
# Open and read the file
with open(filename, 'r', encoding='utf-8') as file:
text = file.read()
# Convert to lowercase and remove punctuation
translator = str.maketrans('', '', string.punctuation)
text = text.translate(translator).lower()
# Split text into words
words = text.split()
# Count word frequencies using Counter
word_freq = Counter[str](words)
return word_freq
except FileNotFoundError:
print(f"Error: File '{filename}' not found.")
return None
except Exception as e:
print(f"Error reading file: {e}")
return None
[19] 0.0s
Python
# Execute the word frequency analysis
filename = 'sample_text.txt'
word_freq = count_word_frequency(filename)
[20] 0.0s
Python


# Display results
if word_freq:
print("\n" + "="*50)
print("WORD FREQUENCY ANALYSIS")
print("="*50)
# Display top 20 most common words
print("\nTop 20 Most Frequent Words:")
print("-"*50)
print(f"{'Word':<20} {'Frequency':<15} {'Percentage':<15}")
print("-"*50)
total_words = sum(word_freq.values())
for word, count in word_freq.most_common(20):
percentage = (count / total_words) * 100
print(f"{word:<20} {count:<15} {percentage:.2f}%")
print("-"*50)
print(f"\nTotal unique words: {len(word_freq)}")
print(f"Total words: {total_words}")
print("="*50)

OUTPUT:



```
...
    ================================================
    WORD FREQUENCY ANALYSIS
    ================================================

    Top 20 Most Frequent Words:
    ------------------------------------------------
    Word                 Frequency       Percentage
    ------------------------------------------------
    python               15              13.64%
    is                   6               5.45%
    and                  6               5.45%
    for                  5               4.55%
    programming          3               2.73%
    data                 3               2.73%
    many                 3               2.73%
    learning             3               2.73%
    the                  3               2.73%
    language             2               1.82%
    used                 2               1.82%
    science              2               1.82%
    developers           2               1.82%
    machine              2               1.82%
    use                  2               1.82%
    analysis             2               1.82%
    ...

    Total unique words: 64
    Total words: 110
    ================================================
    Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

CODE Explanation:

This Python program works by first importing the required modules to handle punctuation removal and word counting. The text file is opened in read mode and its content is read completely. Then, all punctuation marks are removed and the text is converted to lowercase so that words are counted correctly without case differences. After that, the text is split into individual words. The Counter function is used to count the number of times each word appears in the file. The program also includes error handling to display a message if the file is not found or if any other error occurs. Finally, the word frequencies are displayed in an organized format, making the output easy to understand

Q) 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:

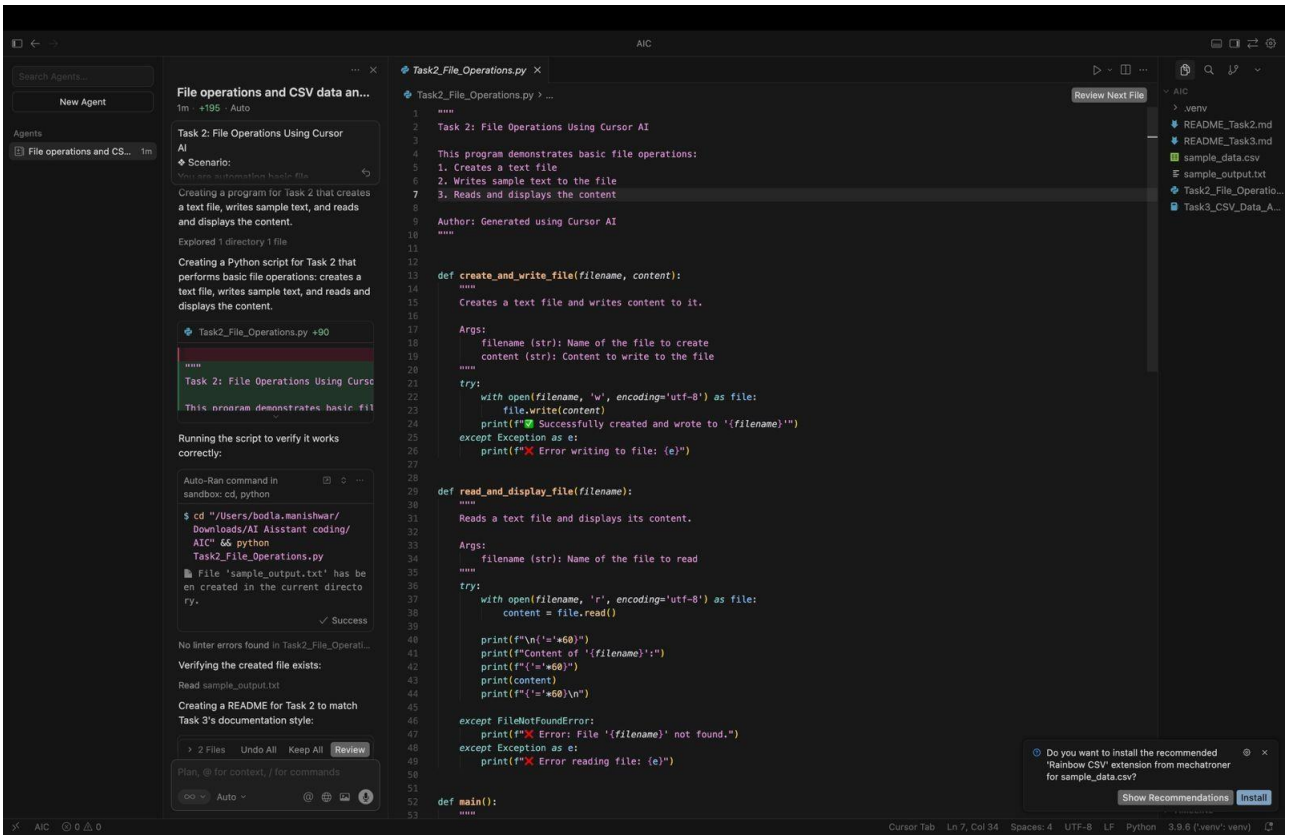
➢ Functional code

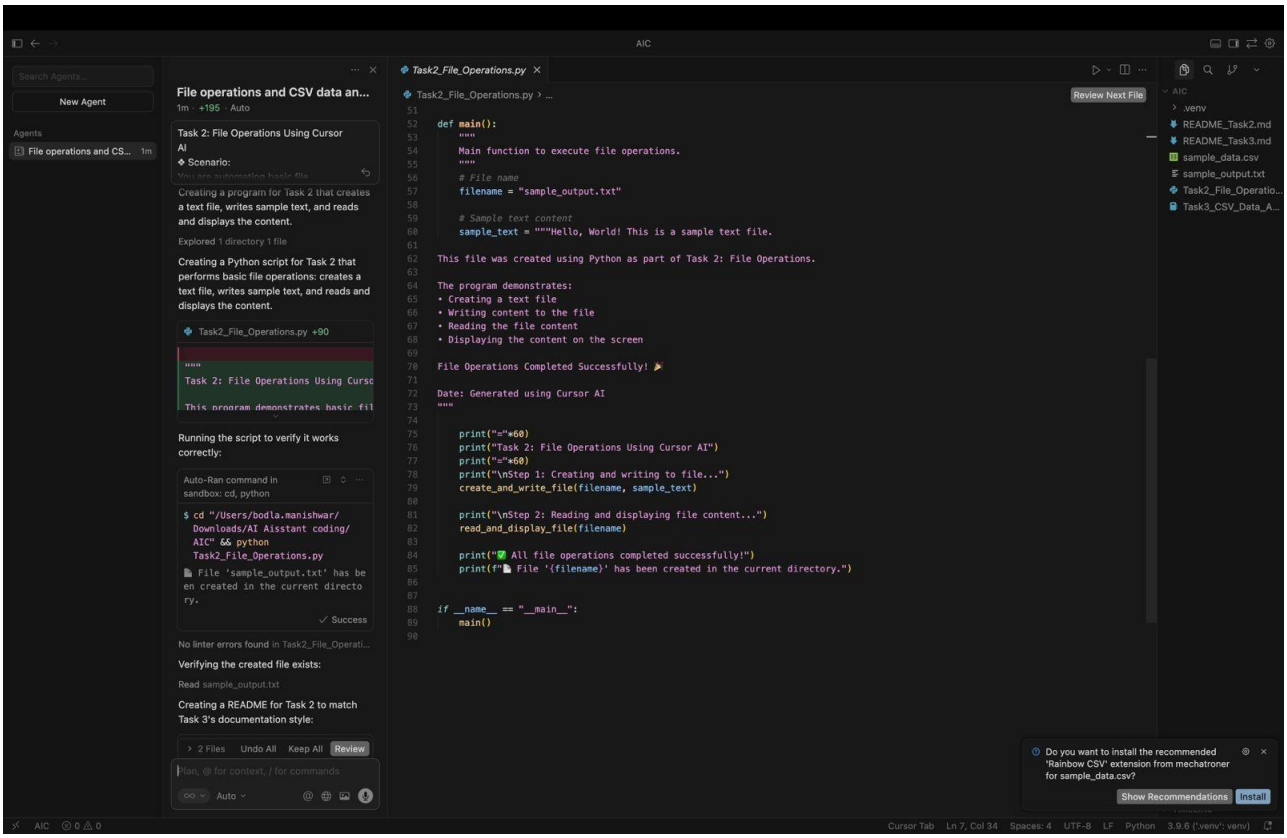
➢ Cursor AI screenshots

PROMPT:

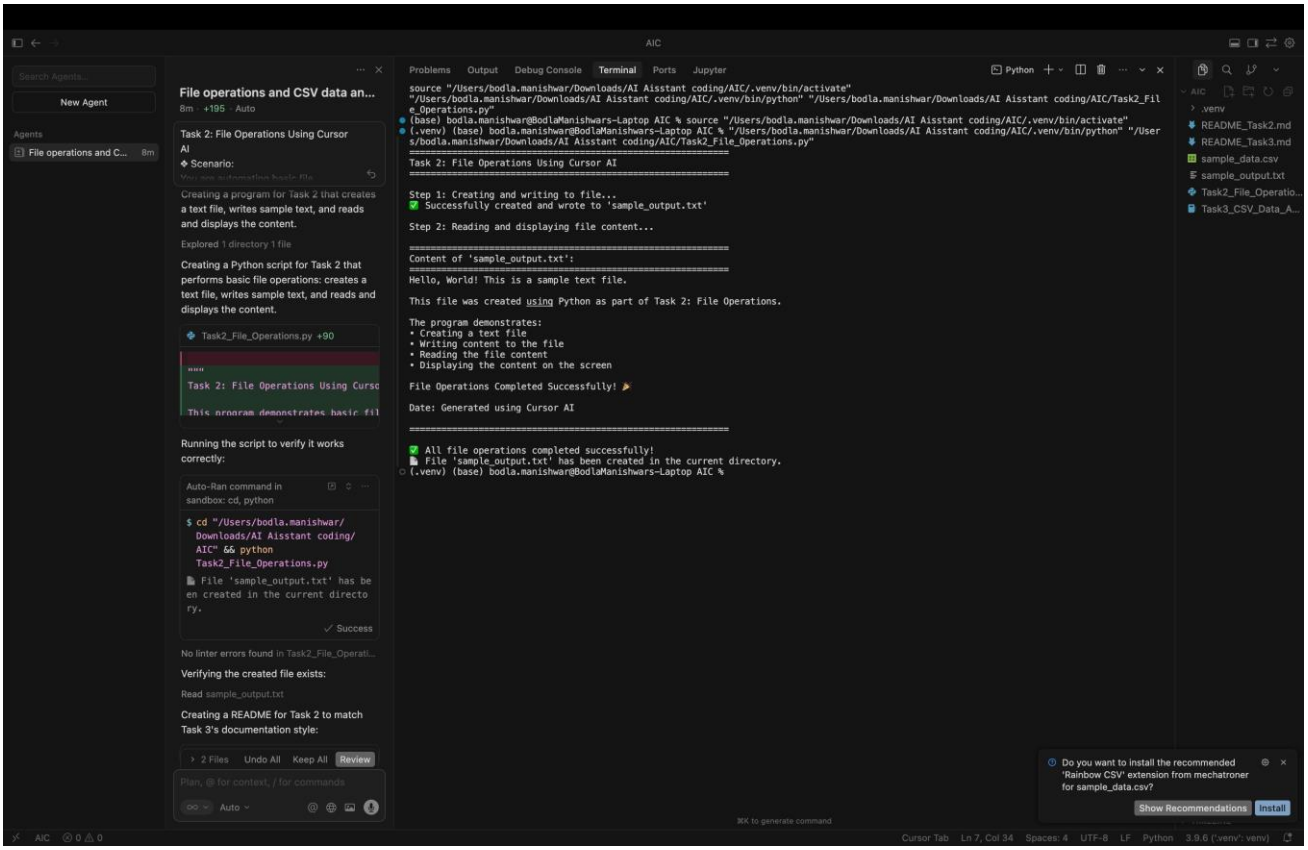
Generate a simple Python program that demonstrates basic file operations. The program should create a text file, write some sample text into it, then read the content from the file and display it on the screen.

CODE:





OUTPUT:



CODE EXPLANATION:

This Python program demonstrates basic file operations by creating a text file, writing sample content to it, and then reading and displaying that content on the screen. It uses separate functions for writing and reading files to keep the code organized and clear. The program also includes exception handling to manage errors such as file access issues, ensuring smooth execution. The main() function controls the overall flow, and the program runs only when executed directly, making it a simple and effective example of file handling in Python.

Q)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:

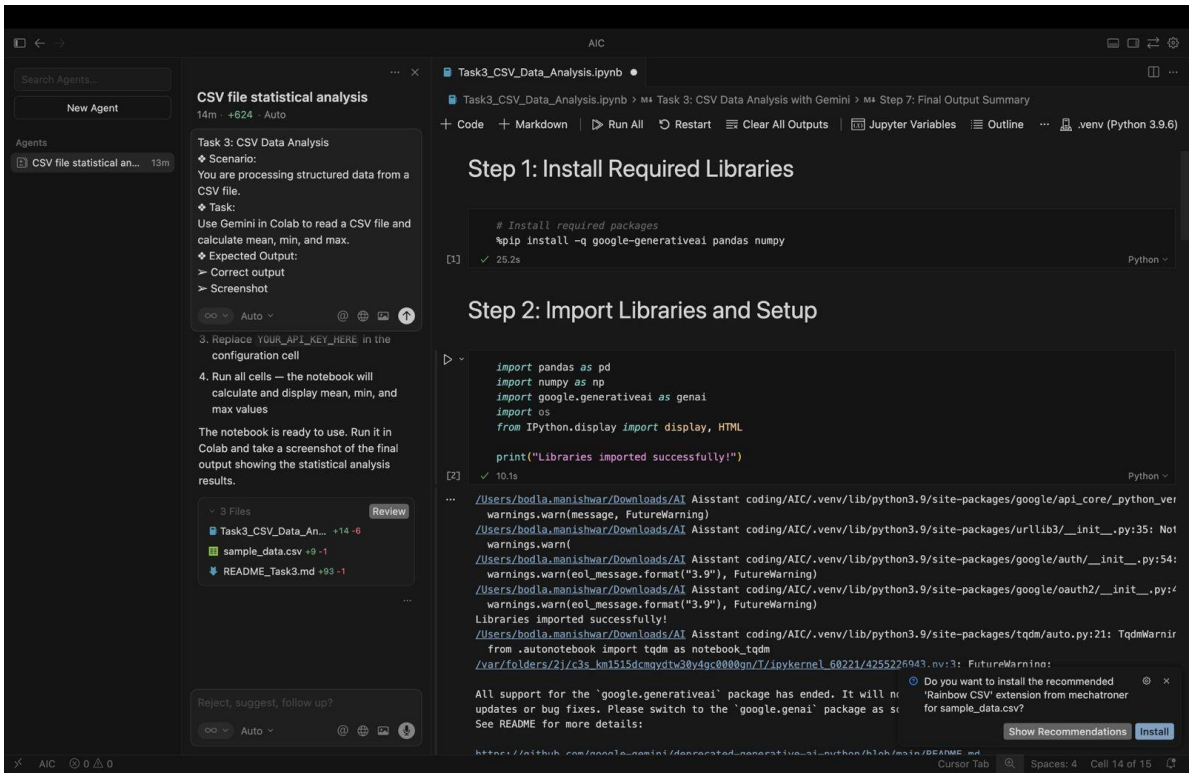
➢ Correct output

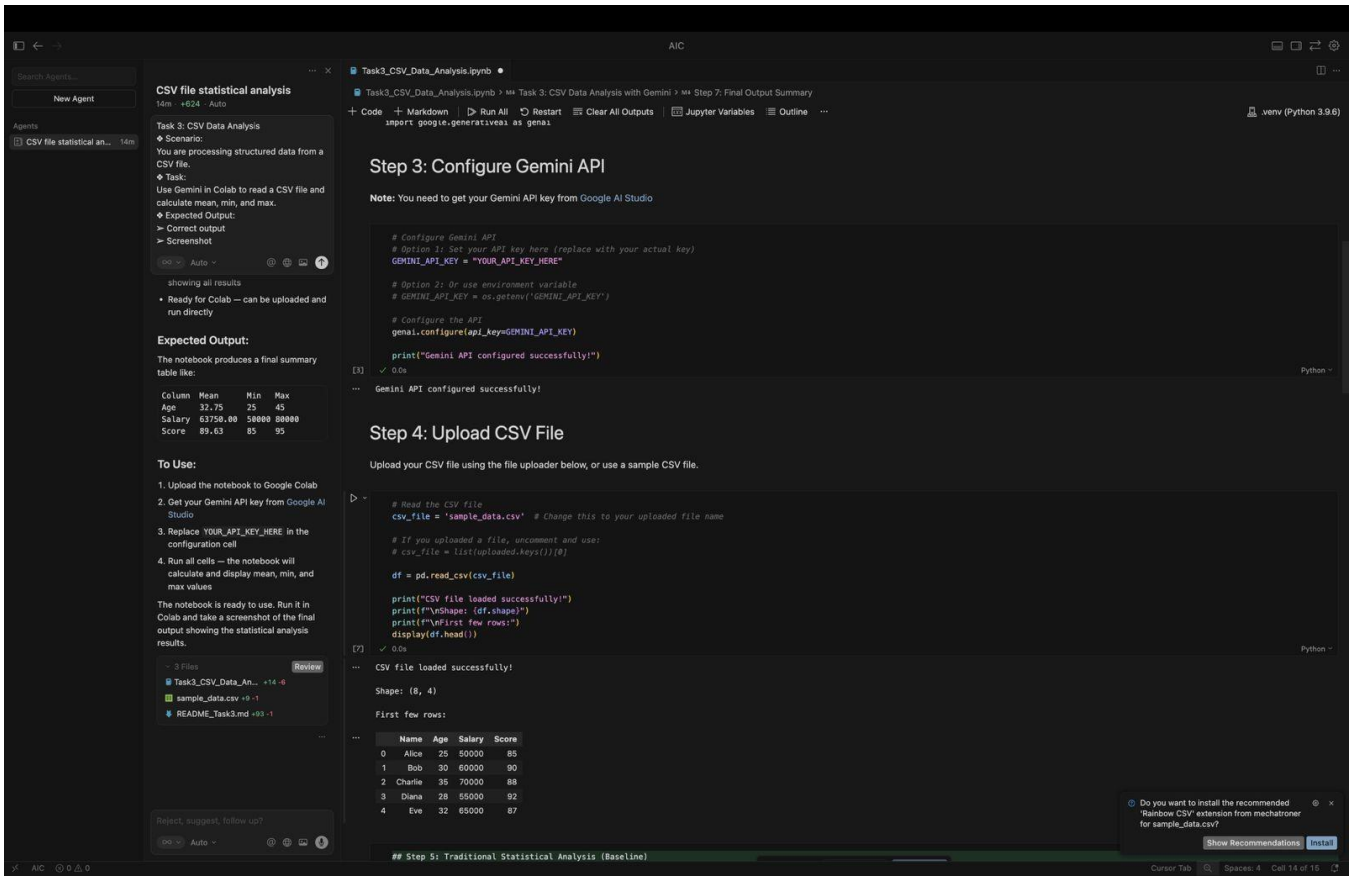
➢ Screenshot

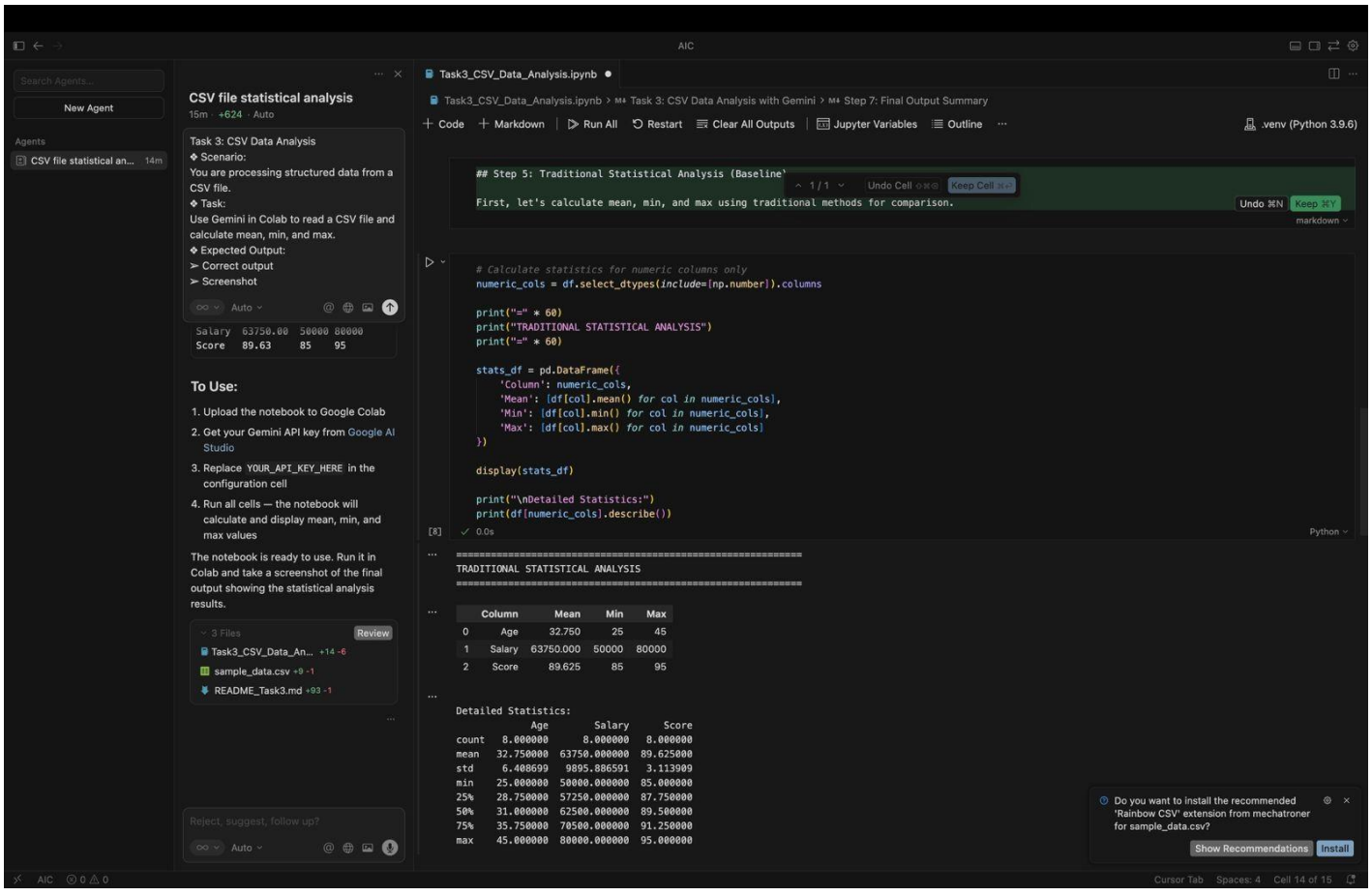
PROMPT:

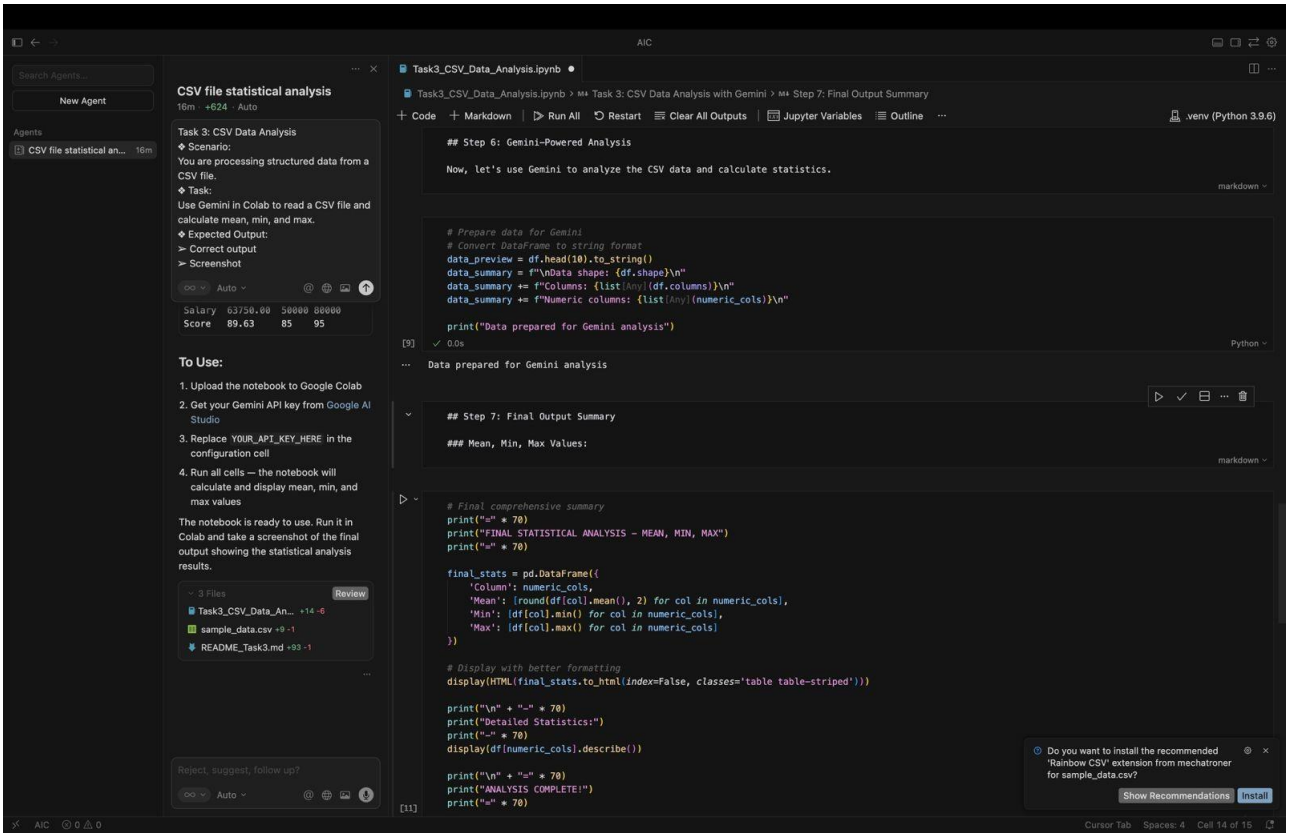
Write Python code in Google Colab to read a CSV file and calculate mean, minimum, and maximum values using pandas.

CODE:

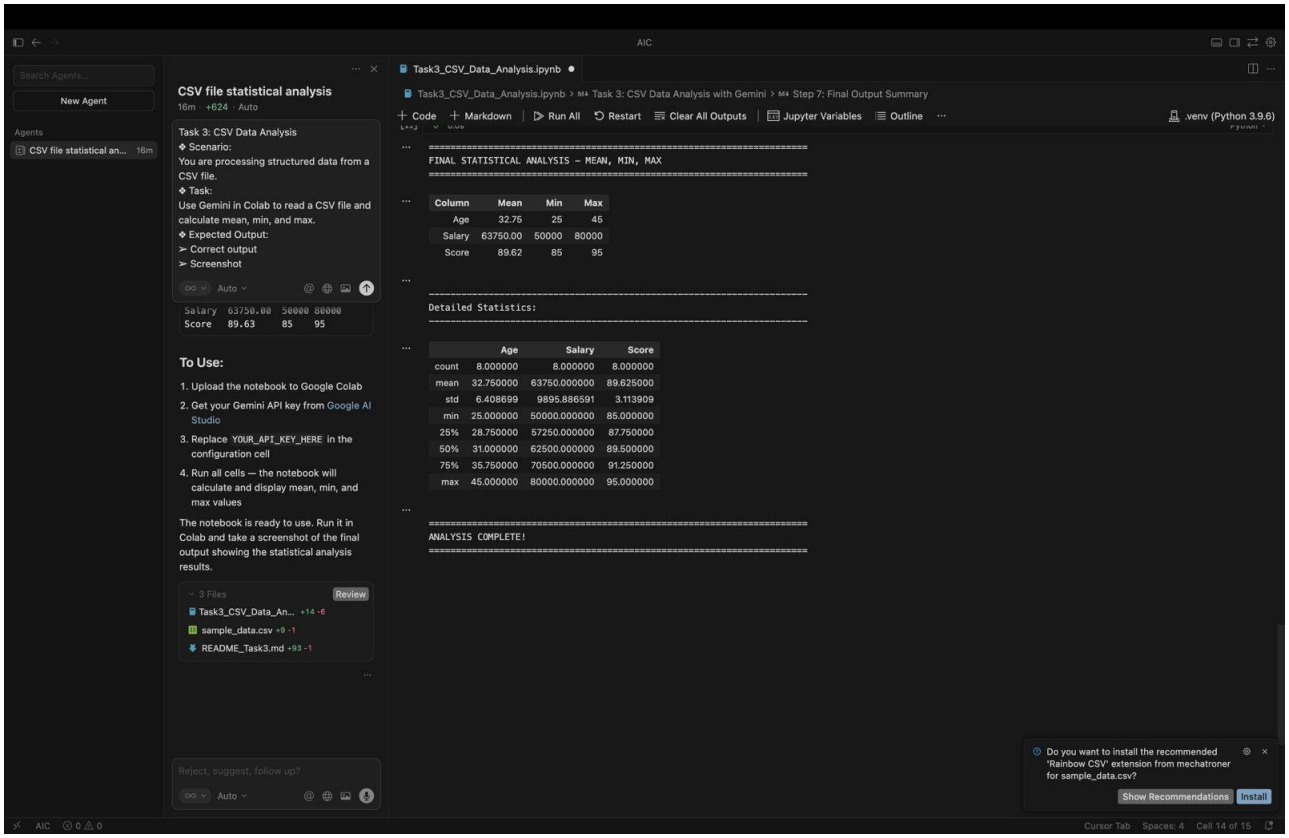



AIC
Task3_CSV_Data_Analysis.ipynb
CSV file statistical analysis
Step 3: Configure Gemini API
Step 4: Upload CSV File
Gemini API configured successfully!
CSV file loaded successfully!
Step 5: Traditional Statistical Analysis (Baseline)


AIC
CSV file statistical analysis
Task3_CSV_Data_Analysis.ipynb
Step 5: Traditional Statistical Analysis (Baseline)
First, let's calculate mean, min, and max using traditional methods for comparison.
TRADITIONAL STATISTICAL ANALYSIS
Detailed Statistics:
To Use:
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?


AIC
Search Agents...
New Agent
Agents
CSV file statistical an... 16m
CSV file statistical analysis
16m · +624 · Auto
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
Auto
Salary 63750.00 50000 80000
Score 89.63 85 95
To Use:
1. Upload the notebook to Google Colab
2. Get your Gemini API key from Google AI Studio
3. Replace YOUR_API_KEY_HERE in the configuration cell
4. Run all cells — the notebook will calculate and display mean, min, and max values
The notebook is ready to use. Run it in Colab and take a screenshot of the final output showing the statistical analysis results.
3 Files
Review
Task3_CSV_Data_An... +14 -6
sample_data.csv +9 -1
README_Task3.md +93 -1
Reject, suggest, follow up?
Auto
Task3_CSV_Data_Analysis.ipynb
Task3_CSV_Data_Analysis.ipynb > Task 3: CSV Data Analysis with Gemini > Step 7: Final Output Summary
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
.venv (Python 3.9.6)
Step 6: Gemini-Powered Analysis
Now, let's use Gemini to analyze the CSV data and calculate statistics.
markdown
# Prepare data for Gemini
# Convert DataFrame to string format
data_preview = df.head(10).to_string()
data_summary = f"\nData shape: {df.shape}\n"
data_summary += f"Columns: {list[Any](df.columns)}\n"
data_summary += f"Numeric columns: {list[Any](numeric_cols)}\n"
print("Data prepared for Gemini analysis")
[9] 0.0s
Python
Data prepared for Gemini analysis
Step 7: Final Output Summary
Mean, Min, Max Values:
markdown
# Final comprehensive summary
print("=" * 70)
print("FINAL STATISTICAL ANALYSIS - MEAN, MIN, MAX")
print("=" * 70)
final_stats = pd.DataFrame({
'Column': numeric_cols,
'Mean': [round(df[col].mean(), 2) for col in numeric_cols],
'Min': [df[col].min() for col in numeric_cols],
'Max': [df[col].max() for col in numeric_cols]
})
# Display with better formatting
display(HTML(final_stats.to_html(index=False, classes='table table-striped')))
print("\n" + "-" * 70)
print("Detailed Statistics:")
print("-" * 70)
display(df[numeric_cols].describe())
print("\n" + "=" * 70)
print("ANALYSIS COMPLETE!")
print("=" * 70)
[11]
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install
AIC 0 0
Cursor Tab
Spaces: 4
Cell 14 of 15

OUTPUT:


AIC
CSV file statistical analysis
16m · +624 · Auto
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
Salary 63750.00 50000 80000
Score 89.63 85 95
To Use:
1. Upload the notebook to Google Colab
2. Get your Gemini API key from Google AI Studio
3. Replace YOUR_API_KEY_HERE in the configuration cell
4. Run all cells — the notebook will calculate and display mean, min, and max values
The notebook is ready to use. Run it in Colab and take a screenshot of the final output showing the statistical analysis results.
Task3_CSV_Data_Analysis.ipynb
Task3_CSV_Data_Analysis.ipynb > Task 3: CSV Data Analysis with Gemini > Step 7: Final Output Summary
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
.venv (Python 3.9.6)
FINAL STATISTICAL ANALYSIS - MEAN, MIN, MAX
Column Mean Min Max
Age 32.75 25 45
Salary 63750.00 50000 80000
Score 89.62 85 95
Detailed Statistics:
Age Salary Score
count 8.000000 8.000000 8.000000
mean 32.750000 63750.000000 89.625000
std 6.408699 9895.886591 3.113909
min 25.000000 50000.000000 85.000000
25% 28.750000 57250.000000 87.750000
50% 31.000000 62500.000000 89.500000
75% 35.750000 70500.000000 91.250000
max 45.000000 80000.000000 95.000000
ANALYSIS COMPLETE!
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install
AIC 0 0
Cursor Tab
Spaces: 4
Cell 14 of 15

CODE EXPLANATION:

This code performs statistical analysis on numeric columns of a DataFrame (df). First, it identifies all columns that contain numerical data using select_dtypes(include=[np.number]). Then, for each numeric column, it calculates the mean, minimum, and maximum values and stores them in a new DataFrame called stats_df. This DataFrame is displayed to show a clean summary of basic statistics.

Q)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:

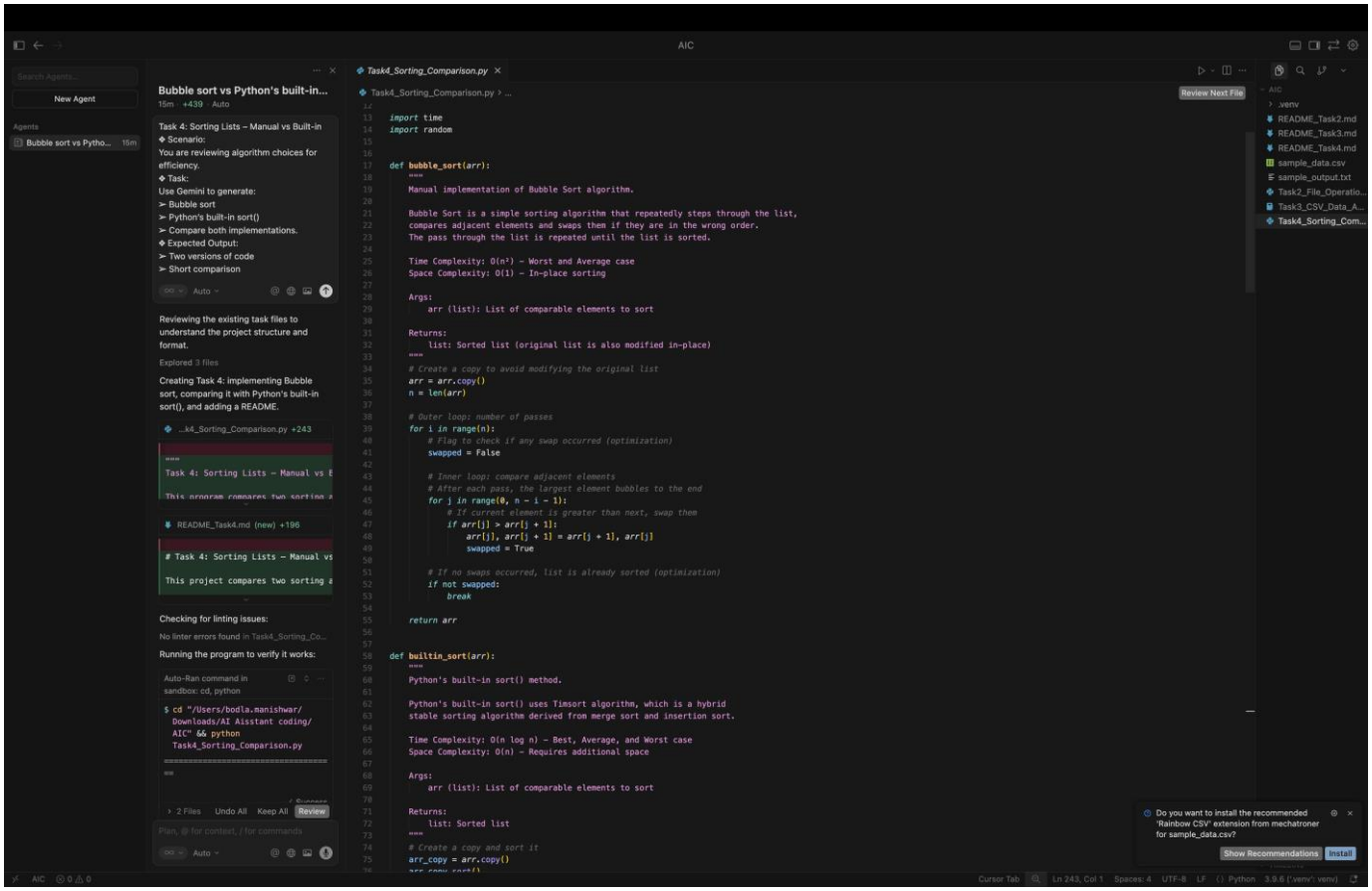
➢ Two versions of code

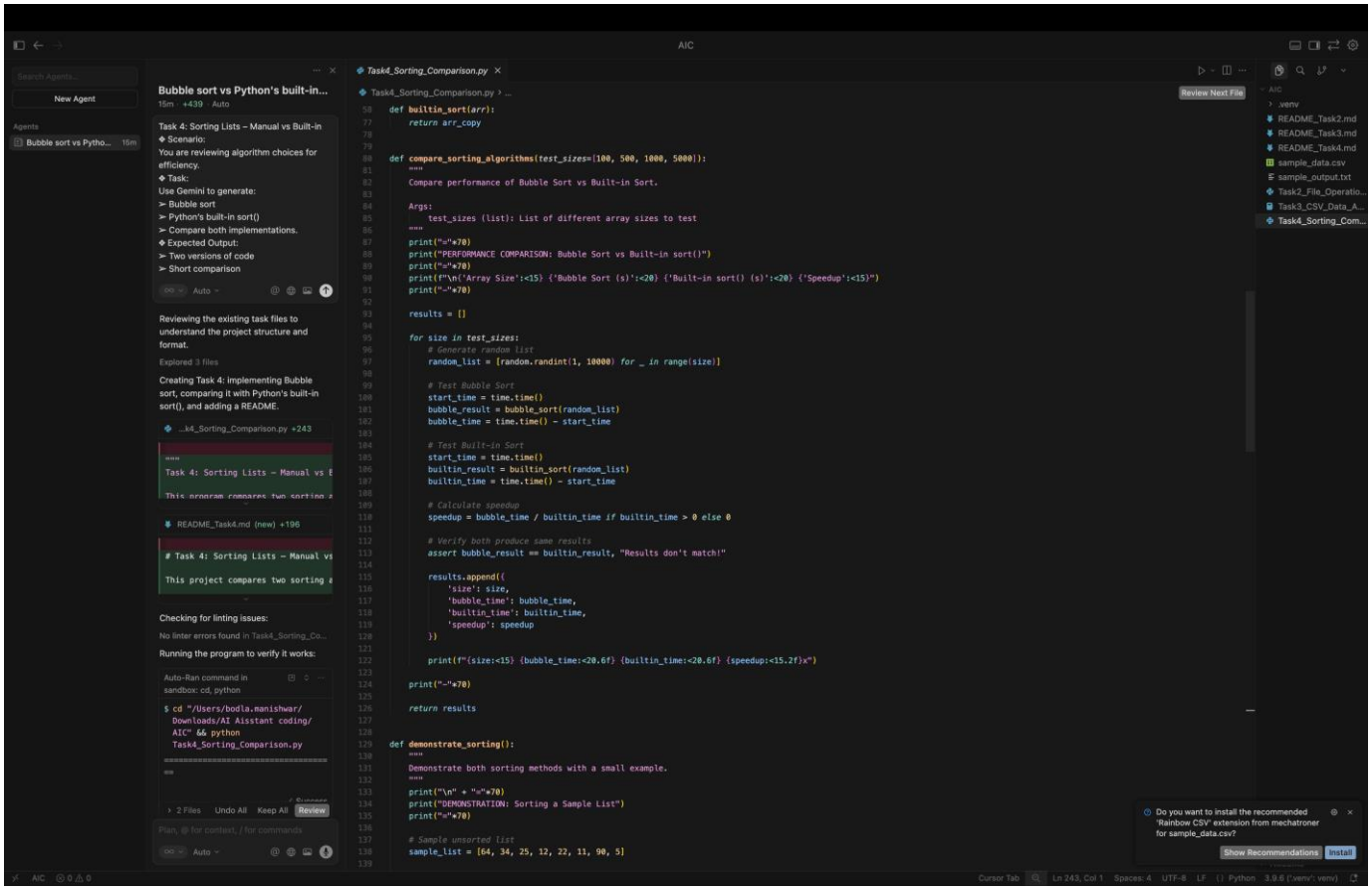
➢ Short comparison

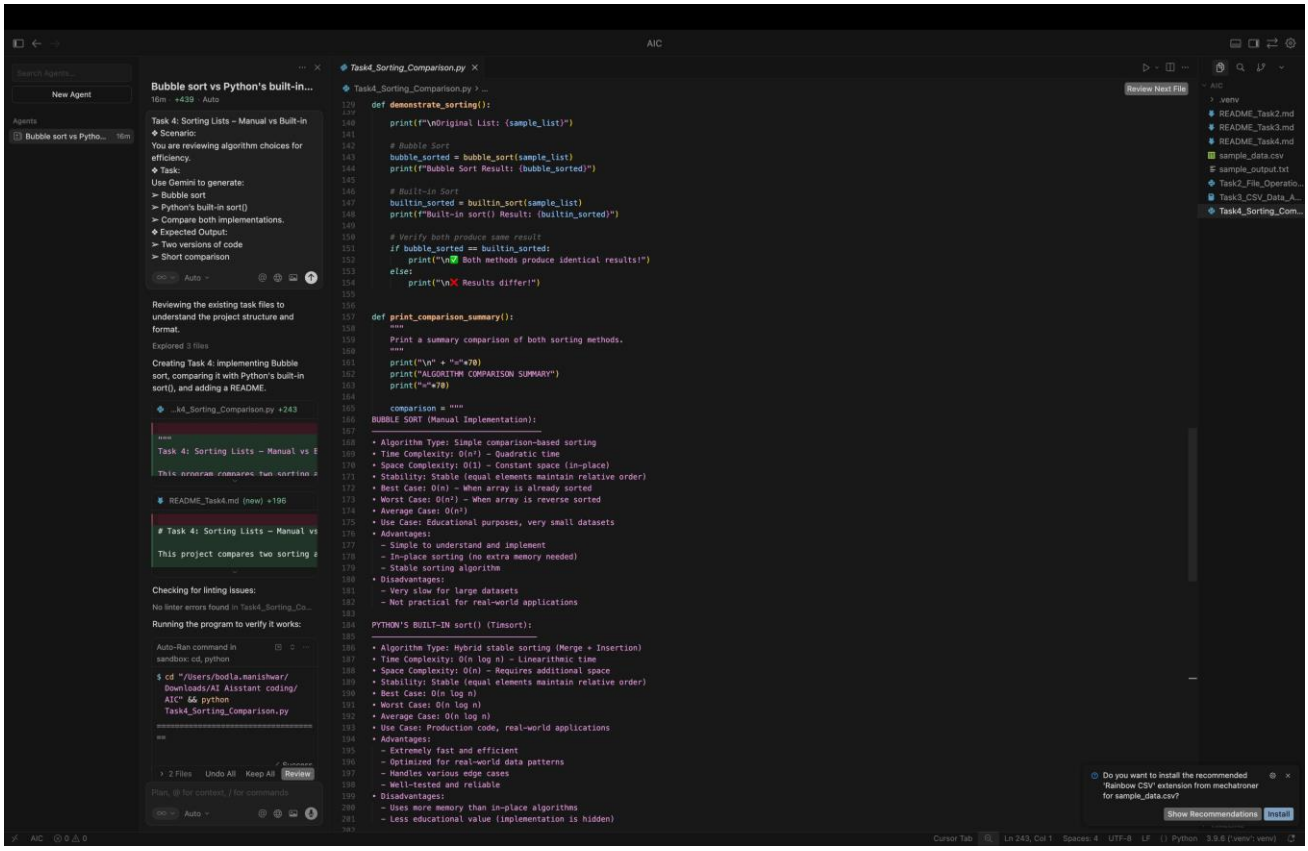
PROMPT:

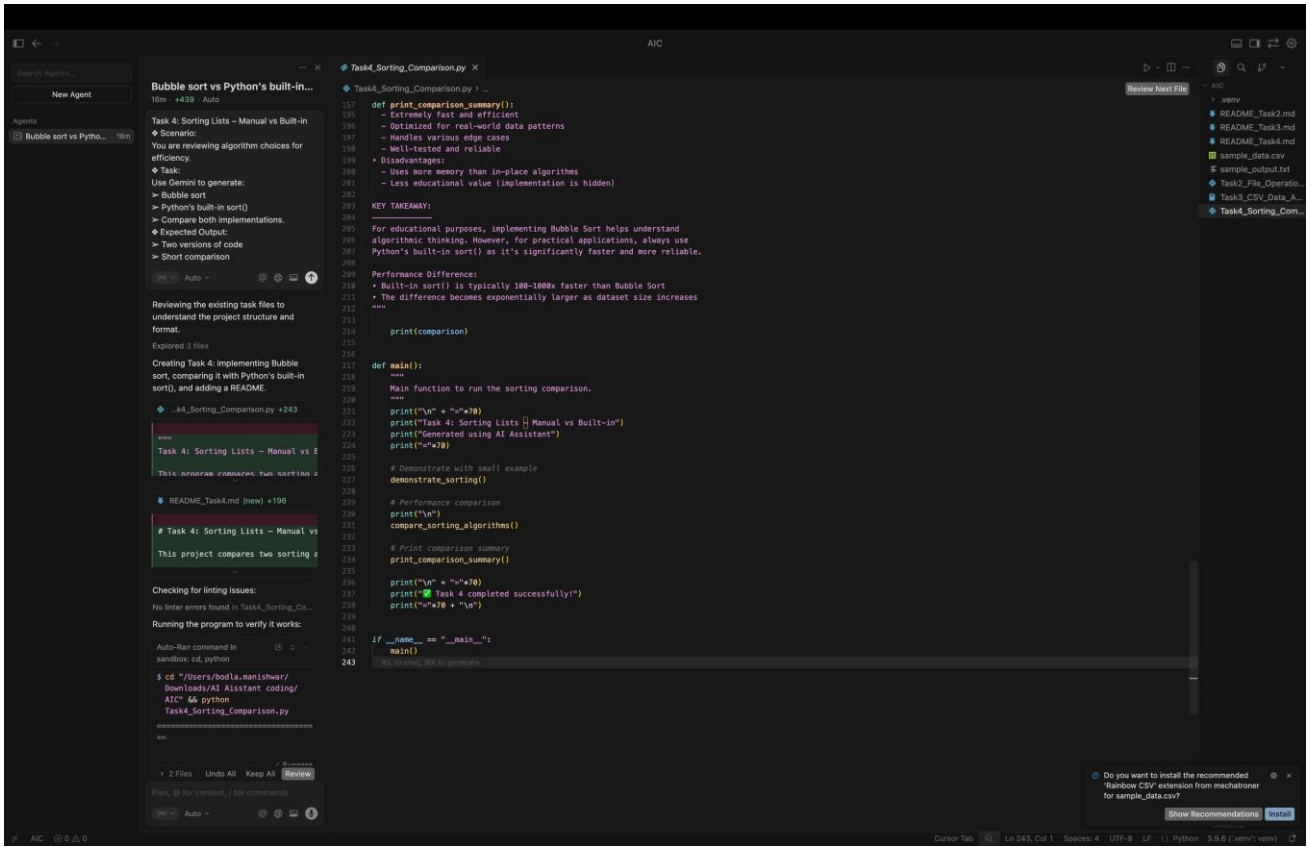
Generate Python code to sort a list using bubble sort and Python's built-in sort() method. Show both implementations clearly and provide a short comparison explaining their efficiency and usage.

CODE:

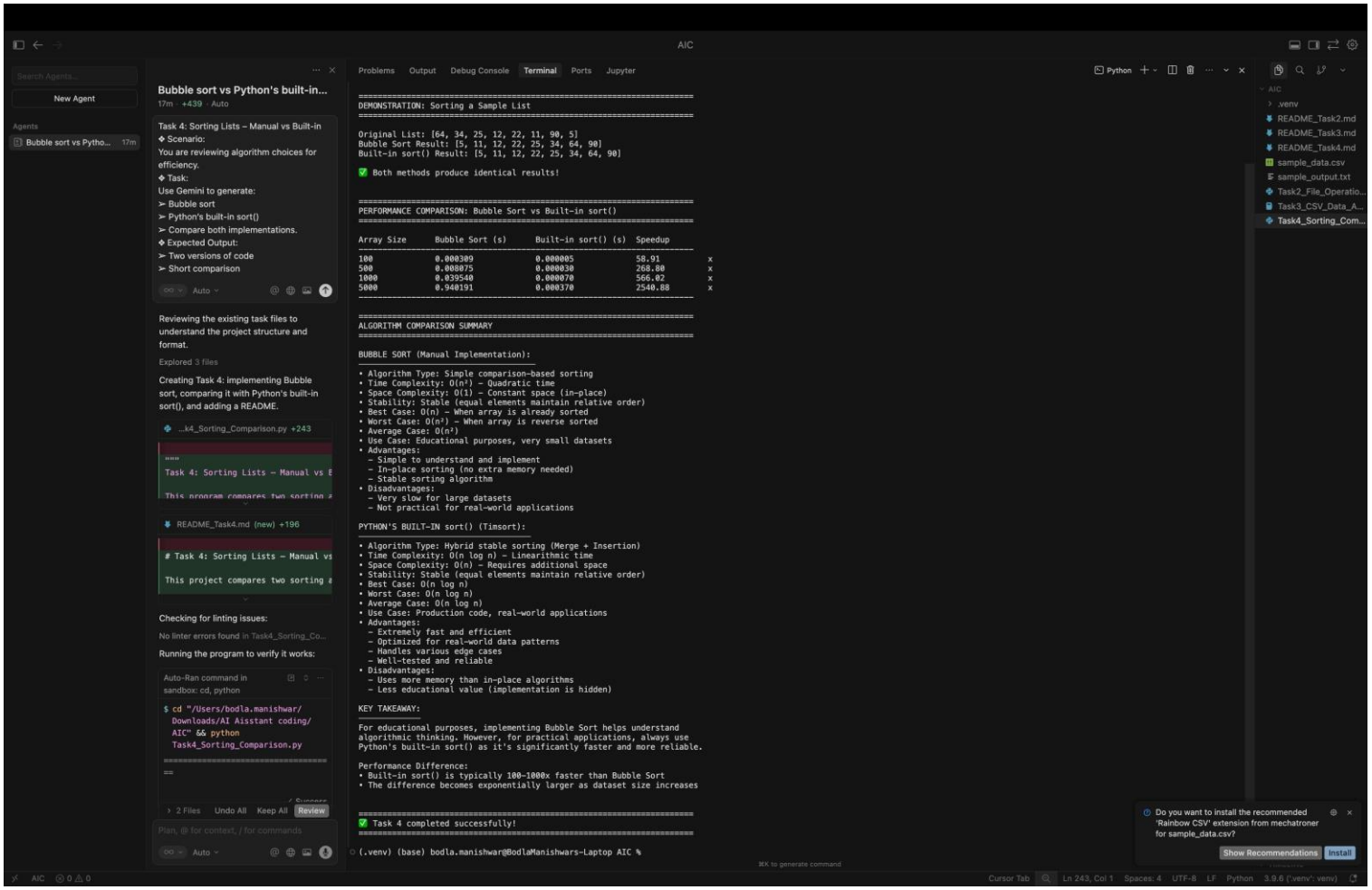



AIC
Bubble sort vs Python's built-in...
New Agent
Task4_Sorting_Comparison.py
Review Next File
def builtin_sort(arr):
return arr_copy
def compare_sorting_algorithms(test_sizes=[100, 500, 1000, 5000]):
"""
Compare performance of Bubble Sort vs Built-in Sort.
Args:
test_sizes (list): List of different array sizes to test
"""
print("="*70)
print("PERFORMANCE COMPARISON: Bubble Sort vs Built-in sort()")
print("="*70)
print(f"\n{'Array Size':<15} {'Bubble Sort (s)':<20} {'Built-in sort() (s)':<20} {'Speedup':<15}")
print("-"*70)
results = []
for size in test_sizes:
# Generate random list
random_list = [random.randint(1, 10000) for _ in range(size)]
# Test Bubble Sort
start_time = time.time()
bubble_result = bubble_sort(random_list)
bubble_time = time.time() - start_time
# Test Built-in Sort
start_time = time.time()
builtin_result = builtin_sort(random_list)
builtin_time = time.time() - start_time
# Calculate speedup
speedup = bubble_time / builtin_time if builtin_time > 0 else 0
# Verify both produce same results
assert bubble_result == builtin_result, "Results don't match!"
results.append({
'size': size,
'bubble_time': bubble_time,
'builtin_time': builtin_time,
'speedup': speedup
})
print(f"{size:<15} {bubble_time:<20.6f} {builtin_time:<20.6f} {speedup:<15.2f}x")
print("-"*70)
return results
def demonstrate_sorting():
"""
Demonstrate both sorting methods with a small example.
"""
print("\n" + "="*70)
print("DEMONSTRATION: Sorting a Sample List")
print("="*70)
# Sample unsorted list
sample_list = [64, 34, 25, 12, 22, 11, 90, 5]
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install


AIC
Bubble sort vs Python's built-in...
New Agent
Task4_Sorting_Comparison.py
Review Next File
def demonstrate_sorting():
print(f"\nOriginal List: {sample_list}")
# Bubble Sort
bubble_sorted = bubble_sort(sample_list)
print(f"Bubble Sort Result: {bubble_sorted}")
# Built-in Sort
builtin_sorted = builtin_sort(sample_list)
print(f"Built-in sort() Result: {builtin_sorted}")
# Verify both produce same result
if bubble_sorted == builtin_sorted:
print("\n✅ Both methods produce identical results!")
else:
print("\n❌ Results differ!")
def print_comparison_summary():
"""
Print a summary comparison of both sorting methods.
"""
print("\n" + "="*70)
print("ALGORITHM COMPARISON SUMMARY")
print("="*70)
comparison = """
BUBBLE SORT (Manual Implementation):
• Algorithm Type: Simple comparison-based sorting
• Time Complexity: O(n²) - Quadratic time
• Space Complexity: O(1) - Constant space (in-place)
• Stability: Stable (equal elements maintain relative order)
• Best Case: O(n) - When array is already sorted
• Worst Case: O(n²) - When array is reverse sorted
• Average Case: O(n²)
• Use Case: Educational purposes, very small datasets
• Advantages:
- Simple to understand and implement
- In-place sorting (no extra memory needed)
- Stable sorting algorithm
• Disadvantages:
- Very slow for large datasets
- Not practical for real-world applications
PYTHON'S BUILT-IN sort() (Timsort):
• Algorithm Type: Hybrid stable sorting (Merge + Insertion)
• Time Complexity: O(n log n) - Linearithmic time
• Space Complexity: O(n) - Requires additional space
• Stability: Stable (equal elements maintain relative order)
• Best Case: O(n log n)
• Worst Case: O(n log n)
• Average Case: O(n log n)
• Use Case: Production code, real-world applications
• Advantages:
- Extremely fast and efficient
- Optimized for real-world data patterns
- Handles various edge cases
- Well-tested and reliable
• Disadvantages:
- Uses more memory than in-place algorithms
- Less educational value (implementation is hidden)
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install


AIC
Bubble sort vs Python's built-in...
Task4_Sorting_Comparison.py
def print_comparison_summary():
- Extremely fast and efficient
- Optimized for real-world data patterns
- Handles various edge cases
- Well-tested and reliable
• Disadvantages:
- Uses more memory than in-place algorithms
- Less educational value (implementation is hidden)
KEY TAKEAWAY:
For educational purposes, implementing Bubble Sort helps understand
algorithmic thinking. However, for practical applications, always use
Python's built-in sort() as it's significantly faster and more reliable.
Performance Difference:
• Built-in sort() is typically 100-1000x faster than Bubble Sort
• The difference becomes exponentially larger as dataset size increases
print(comparison)
def main():
Main function to run the sorting comparison.
print("Task 4: Sorting Lists – Manual vs Built-in")
print("Generated using AI Assistant")
demonstrate_sorting()
compare_sorting_algorithms()
print_comparison_summary()
print("✅ Task 4 completed successfully!")
if __name__ == "__main__":
main()
Reviewing the existing task files to understand the project structure and format.
Creating Task 4: implementing Bubble sort, comparing it with Python's built-in sort(), and adding a README.
Checking for linting issues:
No linter errors found
Running the program to verify it works:
$ cd "/Users/bodla.manishwar/Downloads/AI Aisstant coding/AIC" && python Task4_Sorting_Comparison.py
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install

OUTPUT:


AIC
Problems
Output
Debug Console
Terminal
Ports
Jupyter
DEMONSTRATION: Sorting a Sample List
Original List: [64, 34, 25, 12, 22, 11, 90, 5]
Bubble Sort Result: [5, 11, 12, 22, 25, 34, 64, 90]
Built-in sort() Result: [5, 11, 12, 22, 25, 34, 64, 90]
✅ Both methods produce identical results!
PERFORMANCE COMPARISON: Bubble Sort vs Built-in sort()
Array Size Bubble Sort (s) Built-in sort() (s) Speedup
100 0.000309 0.000005 58.91 x
500 0.008075 0.000030 268.80 x
1000 0.039540 0.000070 566.02 x
5000 0.940191 0.000370 2540.88 x
ALGORITHM COMPARISON SUMMARY
BUBBLE SORT (Manual Implementation):
• Algorithm Type: Simple comparison-based sorting
• Time Complexity: O(n²) - Quadratic time
• Space Complexity: O(1) - Constant space (in-place)
• Stability: Stable (equal elements maintain relative order)
• Best Case: O(n) - When array is already sorted
• Worst Case: O(n²) - When array is reverse sorted
• Average Case: O(n²)
• Use Case: Educational purposes, very small datasets
• Advantages:
- Simple to understand and implement
- In-place sorting (no extra memory needed)
- Stable sorting algorithm
• Disadvantages:
- Very slow for large datasets
- Not practical for real-world applications
PYTHON'S BUILT-IN sort() (Timsort):
• Algorithm Type: Hybrid stable sorting (Merge + Insertion)
• Time Complexity: O(n log n) - Linearithmic time
• Space Complexity: O(n) - Requires additional space
• Stability: Stable (equal elements maintain relative order)
• Best Case: O(n log n)
• Worst Case: O(n log n)
• Average Case: O(n log n)
• Use Case: Production code, real-world applications
• Advantages:
- Extremely fast and efficient
- Optimized for real-world data patterns
- Handles various edge cases
- Well-tested and reliable
• Disadvantages:
- Uses more memory than in-place algorithms
- Less educational value (implementation is hidden)
KEY TAKEAWAY:
For educational purposes, implementing Bubble Sort helps understand
algorithmic thinking. However, for practical applications, always use
Python's built-in sort() as it's significantly faster and more reliable.
Performance Difference:
• Built-in sort() is typically 100-1000x faster than Bubble Sort
• The difference becomes exponentially larger as dataset size increases
✅ Task 4 completed successfully!
(.venv) (base) bodla.manishwar@BodlaManishwars-Laptop AIC %
Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for sample_data.csv?
Show Recommendations
Install

CODE EXPLANATION:

This program compares Bubble Sort and Python's built-in sort(). Bubble Sort manually compares and swaps elements to arrange them in order, but it is slow for large lists because it has $O(n^2)$ time complexity. Python's built-in sort() uses an efficient algorithm and sorts data much faster with O(n log n) time complexity. The program measures execution time for both methods and shows that the built-in sort is much faster and more suitable for real-world use.