**LAB ASSIGNMENT – 2.2**

**Prompt-1:** write a function that filters out all negative numbers from a given list.

**Program:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Explanation:**

The code I provided defines a function called filter\_negative\_numbers that takes a list as input and returns a new list containing only the non-negative numbers.

Here's a breakdown:

* def filter\_negative\_numbers(input\_list):: This line defines the function named filter\_negative\_numbers and specifies that it accepts one argument, input\_list.
* """Docstring""": This is a docstring, which explains what the function does, its arguments, and what it returns.
* return [number for number in input\_list if number >= 0]: This is a list comprehension. It's a concise way to create a new list.
  + for number in input\_list: It iterates through each element in the input\_list.
  + if number >= 0: For each number, it checks if the number is greater than or equal to 0 (i.e., not negative).
  + [number ... ]: If the condition is true, the number is included in the new list being created.

So, the list comprehension effectively builds a new list by including only the numbers from the original list that are not negative.

* my\_list = [1, -2, 3, -4, 5, 0, -6]: This line creates a sample list to demonstrate the function.
* filtered\_list = filter\_negative\_numbers(my\_list): This line calls the filter\_negative\_numbers function with my\_list as input and stores the returned filtered list in the filtered\_list variable.
* print(...): These lines print the original and filtered lists to show the result.

In summary, the function efficiently creates a new list containing only the positive numbers and zero from the input list using a list comprehension.

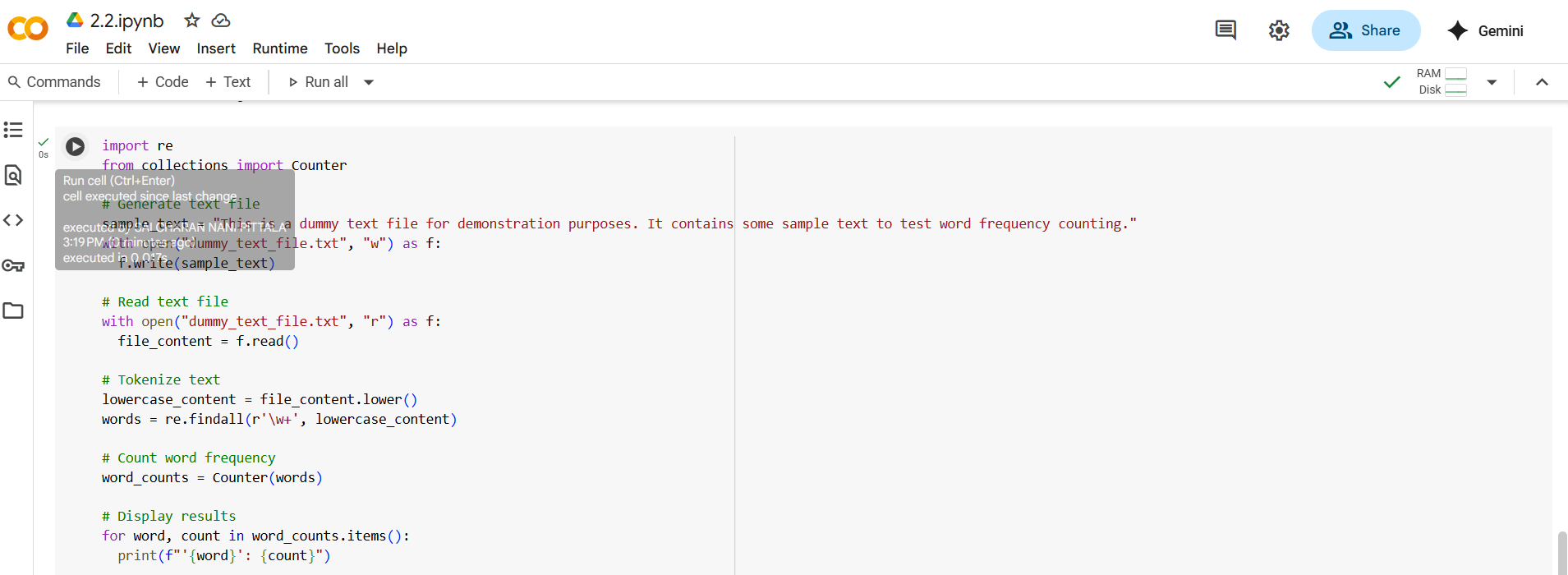
**Output:**

**A number with numbers on it

AI-generated content may be incorrect.**

**Prompt-2:** generate code that reads a text file and counts the frequency of each word.

**Program:**

****

**Explanation:**

This code snippet performs the task of reading a text file and counting the frequency of each word within it. Let's break it down:

1. import re and from collections import Counter: These lines import necessary libraries. re is used for regular expressions, which help in pattern matching (like finding words), and Counter is a specialized dictionary subclass from the collections module that's perfect for counting hashable objects.
2. sample\_text = "...": This line defines a string variable sample\_text containing the text that will be written to our dummy file.
3. with open("dummy\_text\_file.txt", "w") as f:: This opens a file named "dummy\_text\_file.txt" in write mode ("w"). The with statement ensures the file is automatically closed even if errors occur. The opened file object is assigned to the variable f.
   * f.write(sample\_text): This line writes the content of sample\_text into the "dummy\_text\_file.txt".
4. with open("dummy\_text\_file.txt", "r") as f:: This opens the same file, "dummy\_text\_file.txt", but this time in read mode ("r").
   * file\_content = f.read(): This reads the entire content of the file and stores it as a single string in the file\_content variable.
5. lowercase\_content = file\_content.lower(): This converts the entire file\_content string to lowercase. This is done to ensure that words like "The" and "the" are counted as the same word.
6. words = re.findall(r'\w+', lowercase\_content): This is where the tokenization happens.
   * re.findall(): This function finds all occurrences of a pattern in a string and returns them as a list.
   * r'\w+': This is the regular expression pattern.
     + \w: Matches any word character (alphanumeric characters and underscore).
     + +: Matches one or more occurrences of the preceding character.
     + r'': This indicates a raw string, which is helpful for regular expressions to avoid issues with backslashes.
   * So, re.findall(r'\w+', lowercase\_content) finds all sequences of one or more word characters in the lowercase text and returns them as a list of words.
7. word\_counts = Counter(words): This line uses the Counter class to count the frequency of each item in the words list. It returns a Counter object (which behaves like a dictionary) where keys are the words and values are their counts.
8. for word, count in word\_counts.items():: This loop iterates through each key-value pair in the word\_counts Counter object. In each iteration, word gets the word (key) and count gets its frequency (value).
   * print(f"'{word}': {count}"): This line prints each word and its corresponding count in a formatted string.

In essence, the code sets up a file, reads it, cleans the text by making it lowercase and extracting words, counts how many times each word appears, and then prints the results.

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Prompt-3:** Generate a Python class called Book with attributes title, author, and a method summary() that prints the details.

**Program:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Explanation:**

The code you provided defines a Python class called Book and demonstrates how to create and use objects of this class. Here's a breakdown:

1. class Book:: This line defines a new class named Book. Classes serve as blueprints for creating objects, encapsulating data (attributes) and functions (methods) that operate on that data.
2. def \_\_init\_\_(self, title: str, author: str) -> None:: This is the constructor method of the Book class. It's automatically called when you create a new instance (object) of the class.
   * self: This refers to the instance of the class being created.
   * title: str and author: str: These are parameters that the constructor accepts. The : str is a type hint, indicating that these parameters are expected to be strings.
   * -> None: This is a return type hint, indicating that this method does not return any value.
   * self.title: str = title: This line initializes an attribute of the Book object called title and assigns it the value passed to the title parameter. The : str here is also a type hint for the attribute.
   * self.author: str = author: Similarly, this initializes an attribute called author with the value from the author parameter.
3. def summary(self) -> None:: This defines a method named summary for the Book class.
   * self: This refers to the instance of the class on which the method is called.
   * -> None: This return type hint indicates the method does not return a value.
   * print(f"Title: {self.title}\nAuthor: {self.author}"): This line prints the title and author of the specific Book object. The f"" creates an f-string for easy formatting, and \n creates a newline.
4. if \_\_name\_\_ == "\_\_main\_\_":: This is a standard Python construct. The code inside this block will only run when the script is executed directly (not when it's imported as a module into another script). This is good practice for including example usage or testing code within a script.
5. book1 = Book("To Kill a Mockingbird", "Harper Lee"): This line creates the first instance of the Book class. It calls the \_\_init\_\_ constructor with "To Kill a Mockingbird" as the title and "Harper Lee" as the author. The resulting Book object is assigned to the variable book1.
6. print("Book Details:"): This prints a simple header.
7. book1.summary(): This calls the summary() method on the book1 object, which will print its title and author.
8. print("\nEnter book details:"): This prints a newline and a prompt for user input.
9. title = input("Enter book title: "): This line prompts the user to enter a book title and stores the input in the title variable.
10. author = input("Enter book author: "): This line prompts the user to enter a book author and stores the input in the author variable.
11. book2 = Book(title, author): This line creates a second instance of the Book class, using the title and author provided by the user input. The resulting Book object is assigned to the variable book2.
12. print("\nYour book details:"): This prints a newline and a header for the second book's details.
13. book2.summary(): This calls the summary() method on the book2 object, printing the title and author that the user entered.

In summary, the code defines a blueprint for books, creates one example book, prompts the user for details to create a second book, and then displays the details for both.

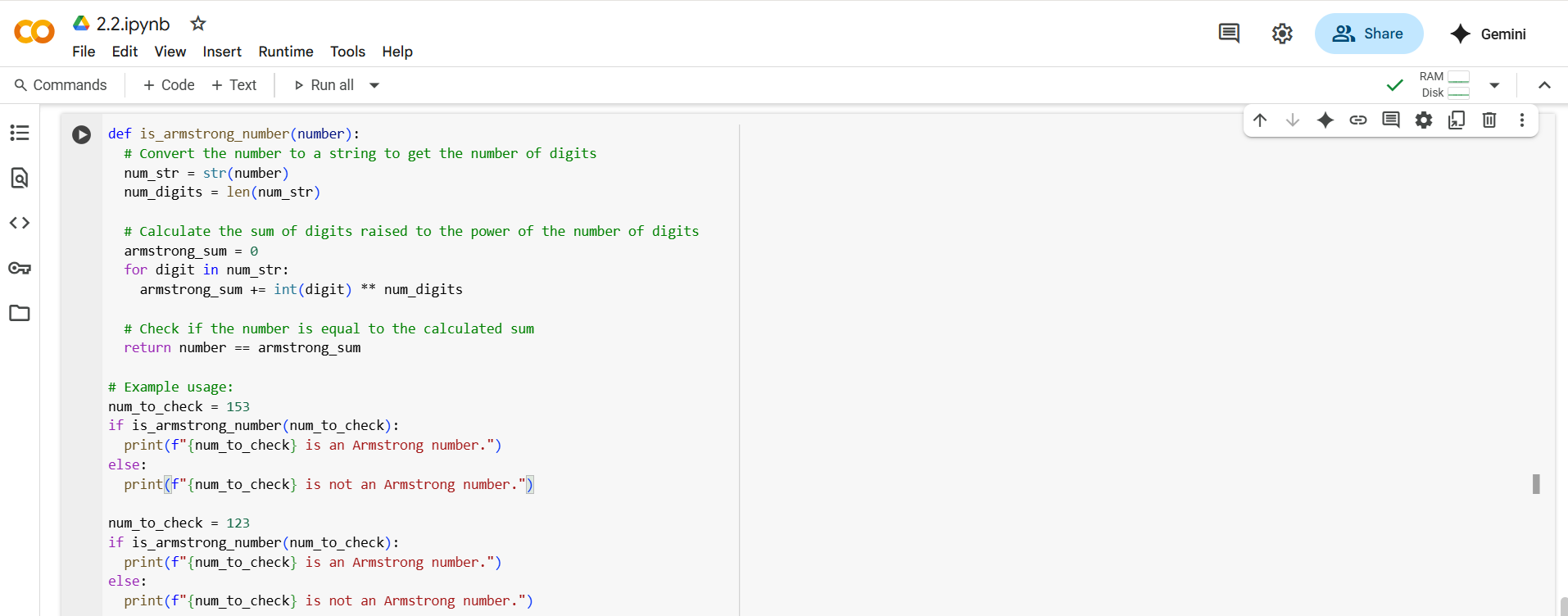
**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Prompt-4:** Write a program that checks whether a number is an Armstrong number.

**Program in gemini:**

****

**Explanation:**

This code defines a function to determine if a given number is an Armstrong number. Here's a breakdown:

1. def is\_armstrong\_number(number):: This line defines a function named is\_armstrong\_number that takes one argument, number.
2. """Docstring""": This is a docstring explaining what an Armstrong number is and what the function does.
3. num\_str = str(number): The input number is converted to a string. This is done so we can easily access each digit individually.
4. num\_digits = len(num\_str): The length of the string representation of the number is determined. This gives us the number of digits in the original number.
5. armstrong\_sum = 0: A variable armstrong\_sum is initialized to 0. This variable will store the sum of each digit raised to the power of the number of digits.
6. for digit in num\_str:: This loop iterates through each character (digit) in the num\_str string.
7. armstrong\_sum += int(digit) \*\* num\_digits: Inside the loop:
   * int(digit): The current digit (which is a string character) is converted back to an integer.
   * \*\* num\_digits: This raises the integer value of the digit to the power of num\_digits (the total number of digits in the original number).
   * armstrong\_sum += ...: The result of the exponentiation is added to the armstrong\_sum.
8. return number == armstrong\_sum: After the loop finishes, the function compares the original number with the calculated armstrong\_sum. If they are equal, it means the number is an Armstrong number, and the function returns True. Otherwise, it returns False.
9. # Example usage:: This section demonstrates how to use the is\_armstrong\_number function.
10. num\_to\_check = 153: A variable num\_to\_check is assigned the value 153.
11. if is\_armstrong\_number(num\_to\_check): ... else: ...: This is an if-else statement that calls the is\_armstrong\_number function with num\_to\_check. Based on the boolean value returned by the function (True or False), it prints a message indicating whether the number is an Armstrong number or not. The same is then done for the number 123.

In essence, the code calculates the sum of each digit raised to the power of the total number of digits and then checks if this sum is equal to the original number.

**Output:**

**A white background with black text

AI-generated content may be incorrect.**

**Modified Program using Cursor:**

**A screen shot of a computer screen

AI-generated content may be incorrect.**

**Explanation in Cursor:**

Performance Improvements:

1. Single string conversion: Convert to string only once instead of multiple times
2. Generator expression: Use sum() with generator expression for better memory efficiency
3. Early returns: Handle edge cases early to avoid unnecessary computation

Structural Improvements:

1. Type hints: Added proper type annotations for better code clarity
2. Comprehensive docstring: Detailed documentation with examples
3. Input validation: Proper error handling for invalid inputs
4. Edge case handling: Special handling for 0 and negative numbers

Additional Features:

1. Range finder: find\_armstrong\_numbers\_in\_range() to find all Armstrong numbers in a range
2. Detailed analysis: get\_armstrong\_number\_info() provides step-by-step calculation
3. Interactive testing: User input functionality
4. Comprehensive test cases: Tests multiple known Armstrong numbers

Key Performance Benefits:

* Memory efficient: Uses generator expressions instead of list comprehensions
* Faster execution: Single string conversion and optimized calculations
* Better error handling: Graceful handling of invalid inputs
* Modular design: Separate functions for different use cases

The improved version is more robust, efficient, and user-friendly while maintaining the same core functionality!

**Output after modifying in cursor:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Prompt-5**: Generate code for sorting a list of dictionaries by a  
specific key (e.g., age).

**Program in Gemini:**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Explanation:**

This code demonstrates how to sort a list of dictionaries in Python based on the values associated with a specific key within each dictionary.

Here's a breakdown:

1. people = [...]: This line initializes a list named people. Each element in this list is a dictionary, representing a person with 'name' and 'age' keys.
2. sorted\_people = sorted(people, key=lambda x: x['age']): This line sorts the people list and stores the sorted result in sorted\_people.
   * sorted(): This is a built-in Python function that returns a new sorted list from the items in an iterable.
   * people: This is the list we want to sort.
   * key=lambda x: x['age']: This is the crucial part for sorting dictionaries.
     + key: This argument specifies a function of one argument that is used to extract a comparison key from each list element.
     + lambda x: x['age']: This is a small anonymous function (a lambda function). It takes one argument, x (which represents each dictionary in the people list during the sorting process), and returns the value associated with the key 'age' from that dictionary (x['age']). The sorted() function uses these returned 'age' values to determine the order of the dictionaries in the sorted list.
3. print("Sorted by age (ascending):") and the following loop: This part prints a header and then iterates through the sorted\_people list, printing each sorted dictionary. By default, sorted() sorts in ascending order.
4. sorted\_people\_desc = sorted(people, key=lambda x: x['age'], reverse=True): This line sorts the people list again, but this time in descending order.
   * reverse=True: This argument to the sorted() function specifies that the sorting should be done in descending order.
5. print("\nSorted by age (descending):") and the following loop: This part prints a header for the descending sort and then iterates through and prints the sorted\_people\_desc list.
6. sorted\_by\_name = sorted(people, key=lambda x: x['name']): This line demonstrates how to sort by a different key. The key is changed to lambda x: x['name'], so the sorting is now based on the values associated with the 'name' key.
7. print("\nSorted by name (ascending):") and the following loop: This part prints a header for the name sort and then iterates through and prints the sorted\_by\_name list.

In summary, the code uses the sorted() function with a lambda function as the key to specify which dictionary value to use for sorting. This allows for flexible sorting of lists containing dictionaries based on any desired key.

**Output in Gemini:**

A screenshot of a computer

AI-generated content may be incorrect.

**Program in Cursor:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Explanation:**

This Python script demonstrates different methods for sorting lists of dictionaries. Here's a comprehensive breakdown:

**Main Functions**

1. **sort\_dict\_list\_by\_key(dict\_list, key, reverse=False)**

* Sorts a list of dictionaries by a single key
* Uses sorted() with a lambda function to extract the specified key value
* Returns a new sorted list

1. **sort\_dict\_list\_by\_multiple\_keys(dict\_list, keys, reverse=False)**

* Sorts by multiple keys in priority order
* Creates tuples of values for comparison
* Useful for secondary sorting (e.g., sort by age, then by salary)

1. **sort\_dict\_list\_by\_custom\_function(dict\_list, key\_func, reverse=False)**

* Allows custom sorting logic using any function
* Enables complex calculations for sorting criteria

1. **display\_people(people, title)**

* Formats and displays the people data in a table format
* Shows name, age, city, and salary columns

**Sample Data**

The script uses a list of 7 people with attributes:

* **Name**: Person's name
* **Age**: Age in years
* **City**: City of residence
* **Salary**: Annual salary

**Three Sorting Methods Demonstrated**

1. **Basic Key Sorting** (Lines 32-40)

* Sorts by age in ascending and descending order
* Simple lambda function: lambda x: x.get(key, 0)

1. **Multiple Key Sorting** (Lines 42-48)

* Sorts by age first, then by salary
* Uses tuple comparison: (age, salary)

1. **Custom Function Sorting** (Lines 50-65)

* Calculates "salary efficiency" (salary/age)
* Demonstrates complex sorting logic

**Key Concepts Explained**

* **Lambda Functions**: Anonymous functions for extracting values
* **Sorted() Function**: Python's built-in sorting with key and reverse parameters
* **Alternative Methods**: list.sort(), operator.itemgetter(), etc.
* **Performance Considerations**: Trade-offs between different approaches

**Interactive Features**

The script includes an interactive demonstration where users can:

* Choose which key to sort by
* Select ascending or descending order
* See the results displayed in a formatted table

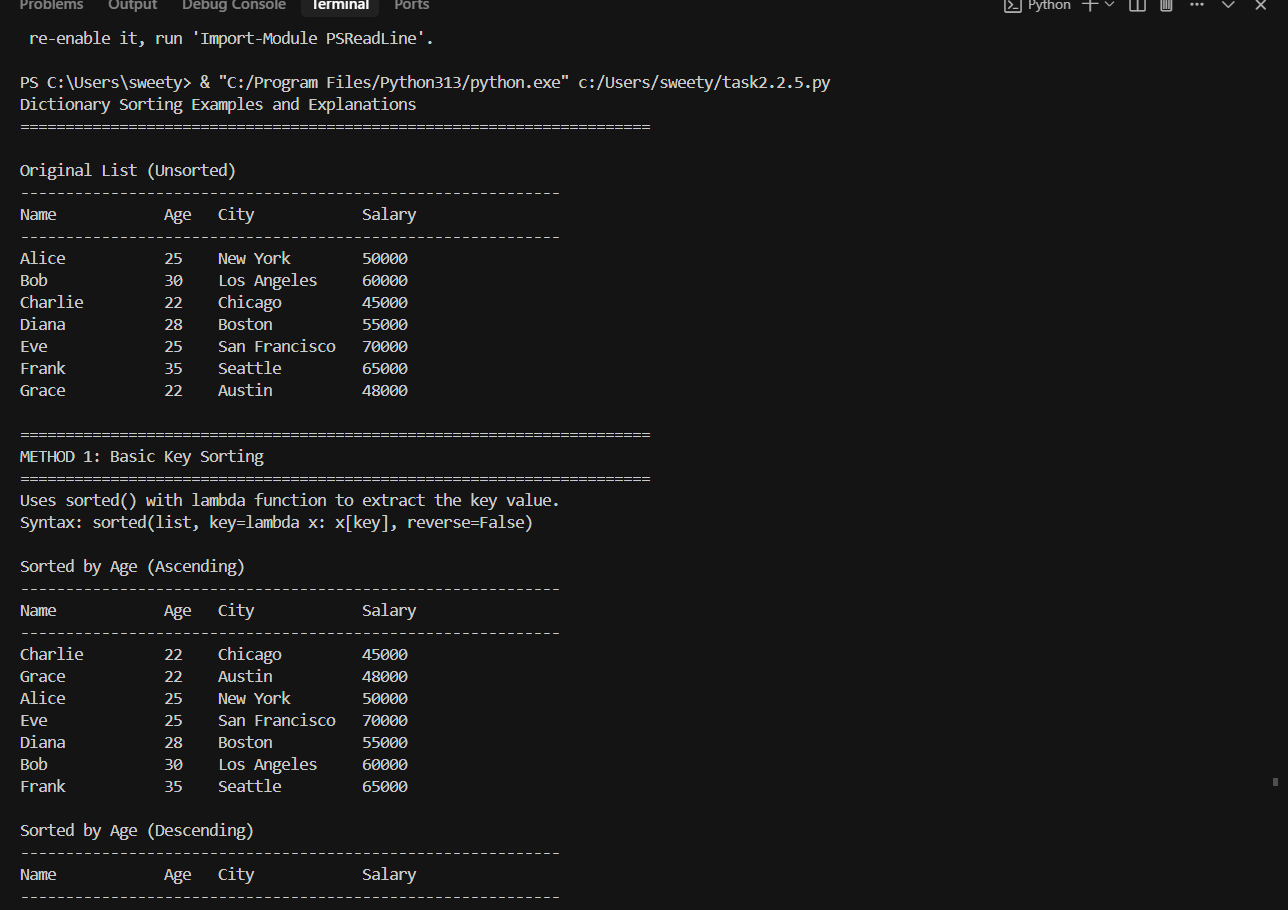
**Educational Value**

This code serves as a comprehensive tutorial covering:

* Dictionary sorting techniques
* Lambda function usage
* Multiple key sorting strategies
* Custom sorting logic
* Error handling and user interaction

The script is well-documented with explanations and practical examples, making it an excellent learning resource for understanding Python dictionary sorting concepts.

**Output in Cursor:**



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Comparision between gemini and Cursor:**

**Strengts:**

1. **Comprehensive and Educational**: Your code is much more thorough, covering multiple sorting methods (basic, multiple keys, custom functions)
2. **Well-Structured**: Uses functions to encapsulate sorting logic, making it reusable
3. **Rich Data**: Uses a more complex dataset with 4 fields (name, age, city, salary)
4. **Educational Value**: Includes detailed explanations, concepts, and interactive demonstrations
5. **Error Handling**: Uses .get() method to handle missing keys safely
6. **Professional Output**: Beautiful formatted table display with proper alignment
7. **Advanced Features**: Demonstrates complex sorting like salary efficiency calculations

**Performance Considerations:**

* Uses lambda functions which are slightly slower than operator.itemgetter()
* Multiple function calls add minimal overhead
* Well-optimized for educational purposes

**Gemini-Generated Code**

**Strengths:**

1. **Simple and Concise**: Easy to understand for beginners
2. **Direct Approach**: Shows the core concept without abstraction
3. **Quick Implementation**: Gets the job done with minimal code

**Limitations:**

1. **Basic Functionality**: Only covers simple sorting by single keys
2. **No Error Handling**: Uses direct key access (x['age']) which could fail with missing keys
3. **Limited Data**: Simple 2-field dataset
4. **No Reusability**: Code is not modular
5. **Basic Output**: Simple print statements without formatting

**Key Differences in Clarity and Performance**

**Clarity:**

* **Your code**: Much clearer for learning purposes with detailed explanations and multiple examples
* **Gemini code**: Simpler but less educational, good for quick reference

**Performance:**

* **Your code**: Slightly slower due to function overhead and lambda usage, but negligible for typical use cases
* **Gemini code**: Faster for simple cases, but lacks safety features

**Best Practices:**

* **Your code**: Follows better practices with error handling, documentation, and modular design
* **Gemini code**: More basic, could fail with edge cases

**Recommendation**

Your code is significantly better for:

* **Learning and teaching** dictionary sorting concepts
* **Production use** due to error handling and modularity
* **Comprehensive understanding** of different sorting approaches

The Gemini code is better for:

* **Quick prototyping** when you need simple sorting
* **Minimal code** requirements

Your implementation demonstrates a much more professional and educational approach to the problem!