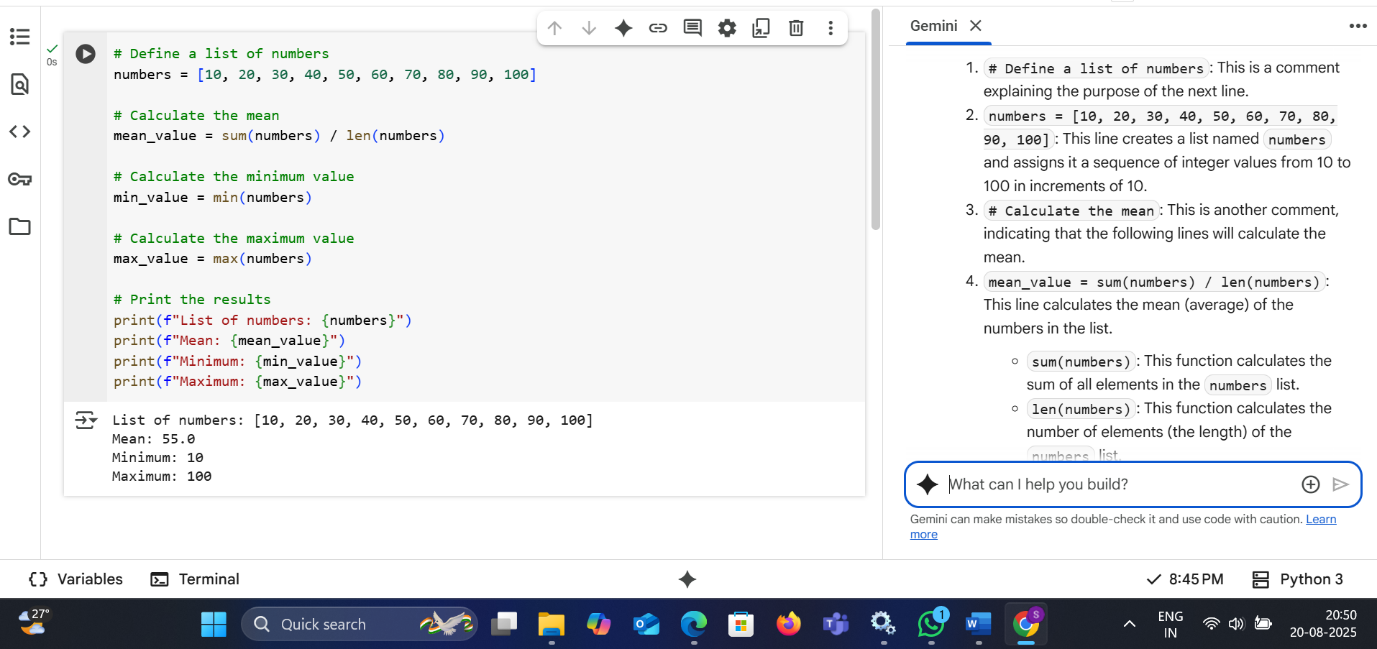
**ASSIGNMENT-2.1**

* **TASK-1**

Python function that reads a list of numbers and calculates the mean, minimum, and maximum values**.**

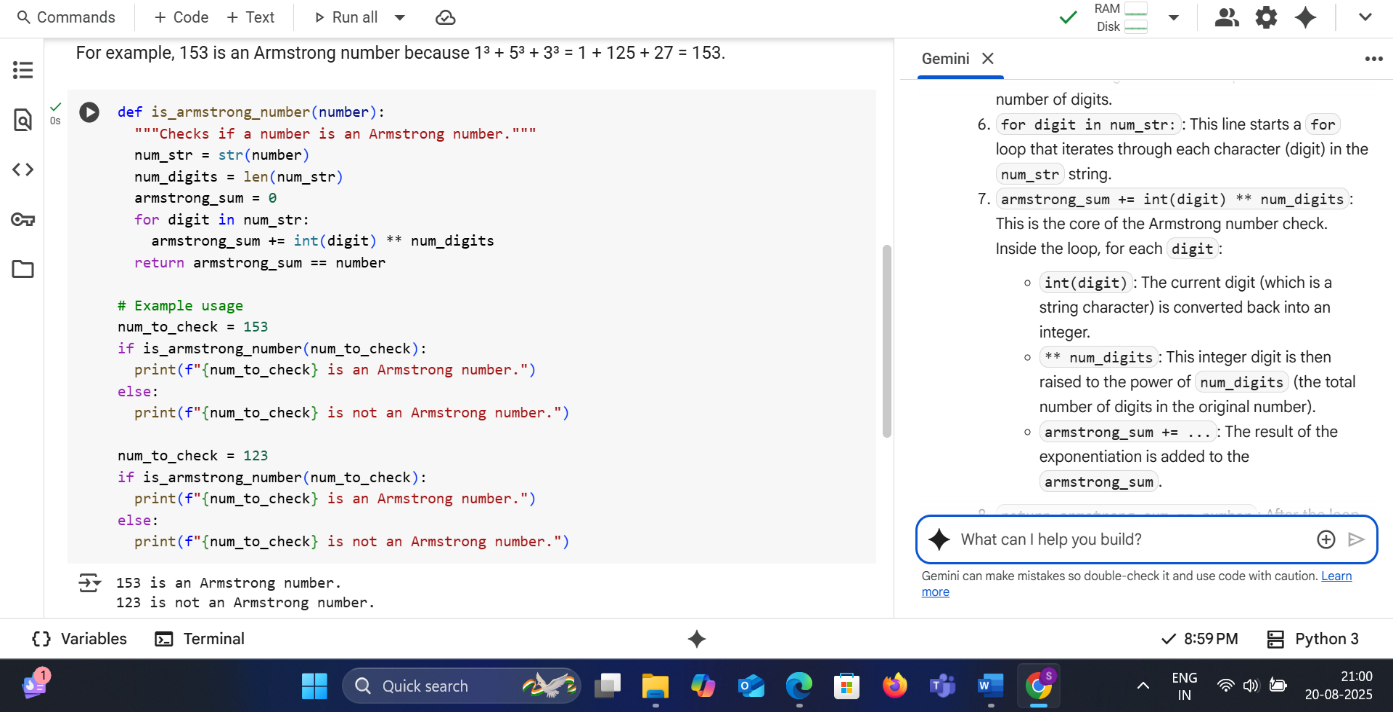
****

* **EXPLANATION:**

1. # Define a list of numbers: This is a comment explaining the purpose of the next line.
2. numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]: This line creates a list named numbers and assigns it a sequence of integer values from 10 to 100 in increments of 10.
3. # Calculate the mean: This is another comment, indicating that the following lines will calculate the mean.
4. mean\_value = sum(numbers) / len(numbers): This line calculates the mean (average) of the numbers in the list.
   * sum(numbers): This function calculates the sum of all elements in the numbers list.
   * len(numbers): This function calculates the number of elements (the length) of the numbers list.
   * The sum is then divided by the length to get the average, which is stored in the variable mean\_value.
5. # Calculate the minimum value: This comment indicates the calculation of the minimum value.
6. min\_value = min(numbers): This line finds the smallest value in the numbers list using the min() function and stores it in the variable min\_value.
7. # Calculate the maximum value: This comment indicates the calculation of the maximum value.
8. max\_value = max(numbers): This line finds the largest value in the numbers list using the max() function and stores it in the variable max\_value.
9. # Print the results: This comment indicates that the next lines will print the calculated results.
10. print(f"List of numbers: {numbers}"): This line prints the original list of numbers. The f"" syntax is a formatted string literal, which allows you to embed the value of variables directly within the string.
11. print(f"Mean: {mean\_value}"): This line prints the calculated mean value.
12. print(f"Minimum: {min\_value}"): This line prints the calculated minimum value.
13. print(f"Maximum: {max\_value}"): This line prints the calculated maximum value.

* **TASK-2**

Python function that checks whether a number is an Armstrong number. Document the steps, prompts, and outputs.

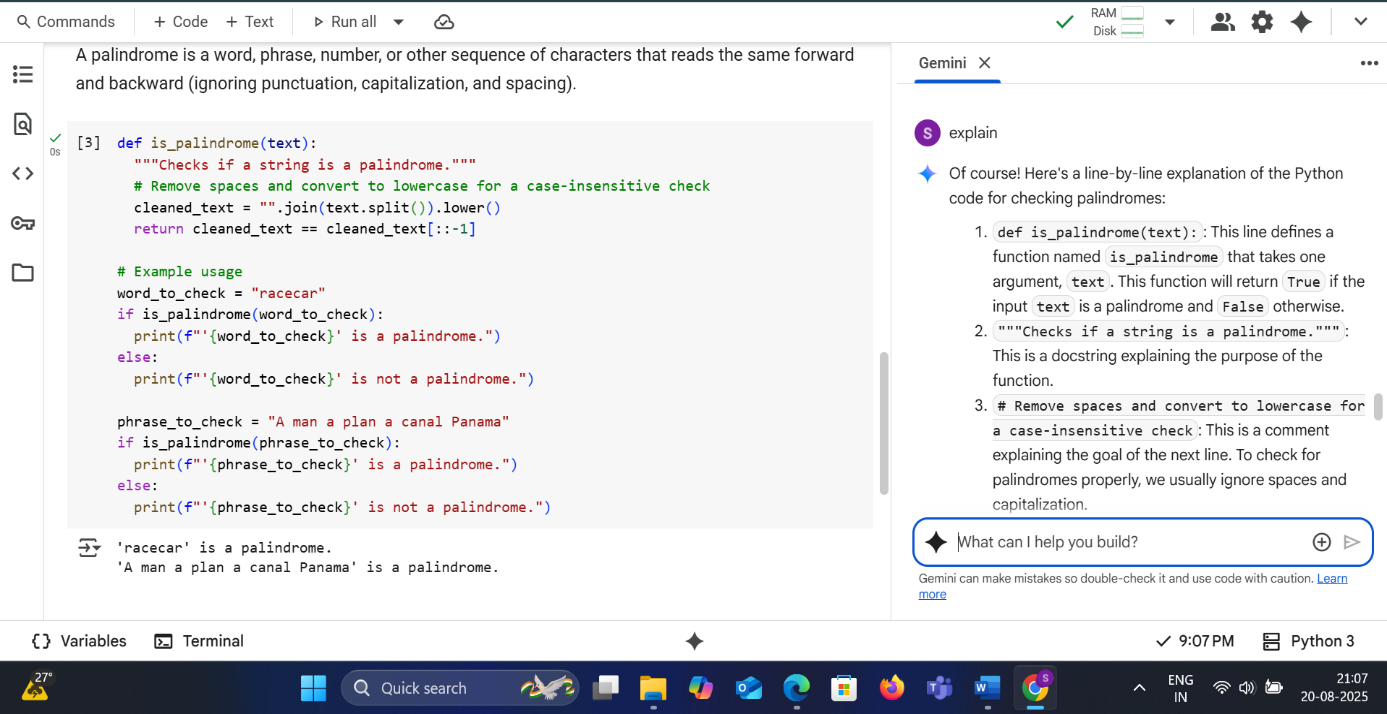


* **LINE BY LINE EXPLANATION:**

1. def is\_armstrong\_number(number):: This line defines a function named is\_armstrong\_number that takes one argument, number. This function will return True if the number is an Armstrong number and False otherwise.
2. """Checks if a number is an Armstrong number.""": This is a docstring, which explains what the function does. It's good practice to include these for your functions.
3. num\_str = str(number): This line converts the input number into a string and stores it in the variable num\_str. This is done so we can easily iterate through each digit of the number.
4. num\_digits = len(num\_str): This line calculates the number of digits in the original number by finding the length of the string representation (num\_str). This value is stored in num\_digits.
5. armstrong\_sum = 0: This line initializes a variable armstrong\_sum to 0. This variable will accumulate the sum of the digits raised to the power of the number of digits.
6. for digit in num\_str:: This line starts a for loop that iterates through each character (digit) in the num\_str string.
7. armstrong\_sum += int(digit) \*\* num\_digits: This is the core of the Armstrong number check. Inside the loop, for each digit:
   * int(digit): The current digit (which is a string character) is converted back into an integer.
   * \*\* num\_digits: This integer digit is then raised 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 armstrong\_sum == number: After the loop has finished processing all digits, this line compares the calculated armstrong\_sum with the original number.
   * If they are equal, the function returns True, indicating that the number is an Armstrong number.
   * If they are not equal, the function returns False.
9. # Example usage: This is a comment indicating the following lines demonstrate how to use the function.
10. num\_to\_check = 153: This line sets the variable num\_to\_check to the value 153.
11. if is\_armstrong\_number(num\_to\_check):: This line calls the is\_armstrong\_number function with num\_to\_check (which is 153). It then checks if the function returns True.
12. print(f"{num\_to\_check} is an Armstrong number."): If the if condition is True (meaning 153 is an Armstrong number), this line prints a message confirming it.
13. else:: This keyword marks the alternative block of code to be executed if the if condition is False.
14. print(f"{num\_to\_check} is not an Armstrong number."): If the if condition is False, this line prints a message indicating that the number is not an Armstrong number.
15. The following lines repeat the example usage for the number 123 to show a case where the function returns False.

* **TASK-3**

Palindrome-checking function and document the explanation provided by Gemini.

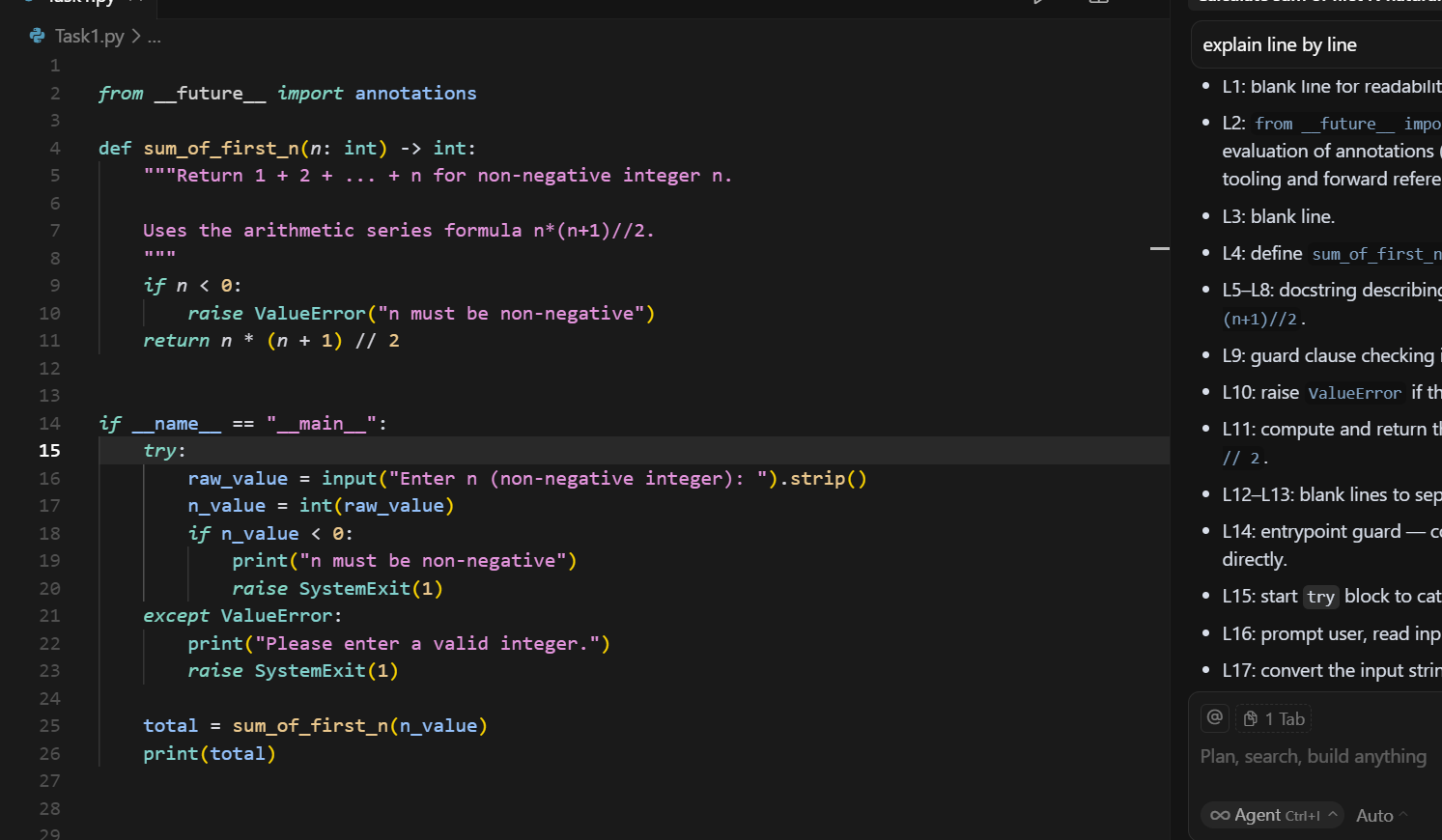


* **LINE BY LINE EXPLANATION**:

1. def is\_palindrome(text):: This line defines a function named is\_palindrome that takes one argument, text. This function will return True if the input text is a palindrome and False otherwise.
2. """Checks if a string is a palindrome.""": This is a docstring explaining the purpose of the function.
3. # Remove spaces and convert to lowercase for a case-insensitive check: This is a comment explaining the goal of the next line. To check for palindromes properly, we usually ignore spaces and capitalization.
4. cleaned\_text = "".join(text.split()).lower(): This line performs the cleaning of the input text:
   * text.split(): This splits the input text into a list of words, using spaces as the delimiter.
   * "".join(...): This joins the words back together into a single string, but without any spaces in between.
   * .lower(): This converts the resulting string to all lowercase characters. The cleaned and lowercased string is stored in the variable cleaned\_text.
5. return cleaned\_text == cleaned\_text[::-1]: This line performs the actual palindrome check and returns the result.
   * cleaned\_text[::-1]: This is a Python slicing technique that creates a reversed copy of the cleaned\_text string. The [::-1] slice means start from the end, go to the beginning, with a step of -1 (which reverses the string).
   * cleaned\_text == ...: This compares the original cleaned\_text with its reversed version.
   * If they are equal, the function returns True (it's a palindrome).
   * If they are not equal, the function returns False (it's not a palindrome).
6. # Example usage: This is a comment indicating the following lines demonstrate how to use the function.
7. word\_to\_check = "racecar": This line sets the variable word\_to\_check to the string "racecar".
8. if is\_palindrome(word\_to\_check):: This line calls the is\_palindrome function with "racecar" and checks if it returns True.
9. print(f"'{word\_to\_check}' is a palindrome."): If the if condition is True, this line prints a message indicating that "racecar" is a palindrome.
10. else:: This marks the alternative block of code if the if condition is False.
11. print(f"'{word\_to\_check}' is not a palindrome."): If the if condition is False, this line prints a message that the word is not a palindrome.
12. The following lines repeat the example usage with the phrase "A man a plan a canal Panama" to demonstrate how the code handles phrases with spaces and mixed capitalization.

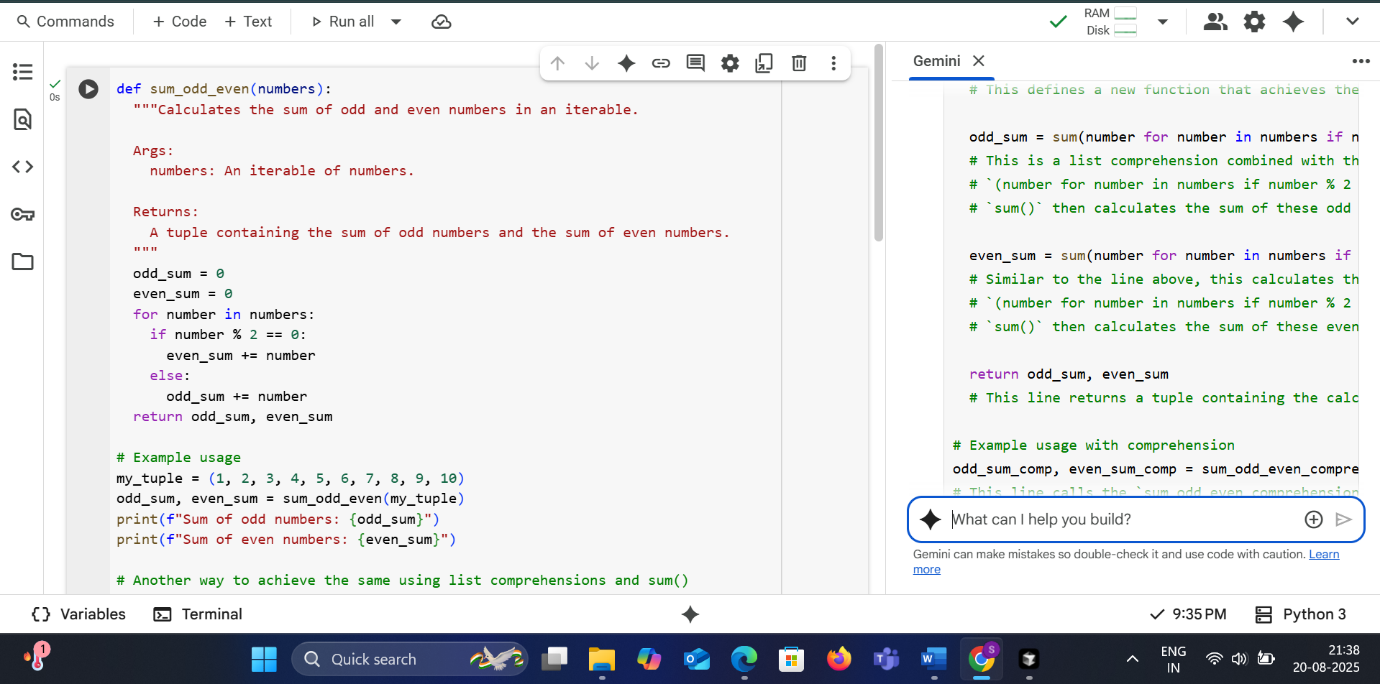
* **TASK-4**

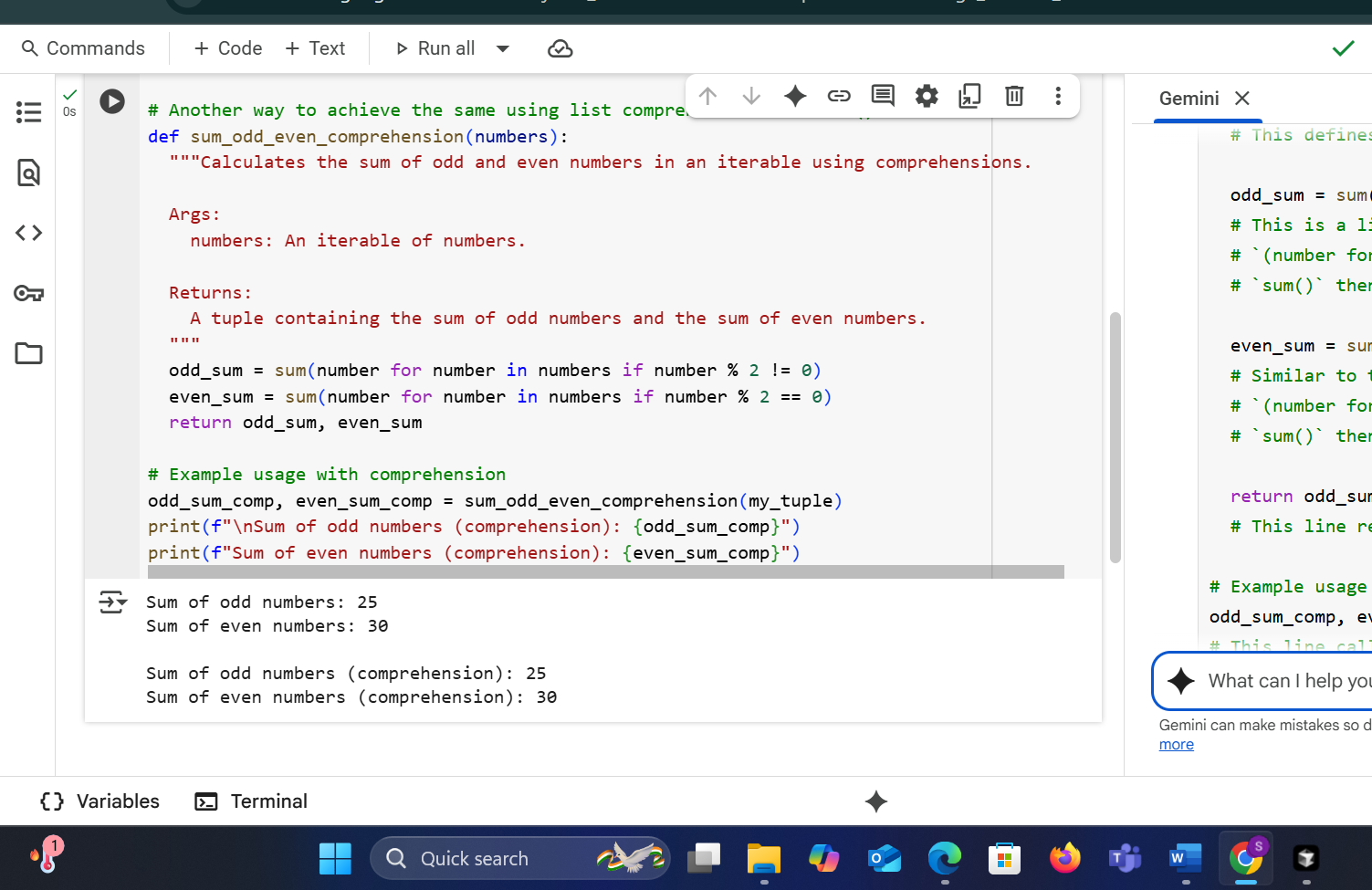
Install and configure Cursor AI. Use it to generate a Python function (e.g., sum of the first N natural numbers) and test its output.

****

* **LINE BY LINE EXPLANATION**:
* L1: blank line for readability.
* L2: from \_\_future\_\_ import annotations enables postponed evaluation of annotations (keeps types as strings at runtime, helpful for tooling and forward references).
* L3: blank line.
* L4: define sum\_of\_first\_n taking n: int and returning int.
* L5–L8: docstring describing the function and that it uses the formula n\*(n+1)//2.
* L9: guard clause checking if n is negative.
* L10: raise ValueError if the input is invalid (negative).
* L11: compute and return the sum using integer arithmetic: n \* (n + 1) // 2.
* L12–L13: blank lines to separate function from script entrypoint.
* L14: entrypoint guard — code below runs only when the file is executed directly.
* L15: start try block to catch conversion errors.
* L16: prompt user, read input as text, and .strip() whitespace.
* L17: convert the input string to an integer.
* L18: check again that the integer is non-negative.
* L19: print an error message if it’s negative.
* L20: exit the program with status code 1 (indicates error).
* L21: catch ValueError raised when the input couldn’t be converted to int.
* L22: print a message asking for a valid integer.
* L23: exit with status code 1 due to invalid input.
* L24: blank line separating error handling from normal flow.
* L25: call sum\_of\_first\_n with the validated n\_value.
* L26: print the computed total.
* L27–L29: trailing blank lines (no effect; just spacing).
* In short: The script reads a non-negative integer n, validates it, computes 1+2+...+n using n\*(n+1)//2, and prints the result.
* **TASK-5**

write a Python program to calculate the sum of odd numbers and even numbers in a given tuple. ● Refactor the code to improve logic and readability





* **LINE BY LINE EXPLANATION**:

This code provides two ways to calculate the sum of odd and even numbers within a collection of numbers (like a list or tuple).

1. **The first method (sum\_odd\_even)** works by going through each number one by one. It checks if the number is even or odd using the remainder when divided by 2. Based on whether it's even or odd, it adds the number to either a running total for even numbers or a running total for odd numbers. Finally, it gives you back the two totals.
2. **The second method (sum\_odd\_even\_comprehension)** is a more compact way to do the same thing using a Python feature called list comprehensions (or in this case, generator expressions). It quickly filters out all the odd numbers and sums them up, and then separately filters out all the even numbers and sums them up. This is often considered more "Pythonic" for simple filtering and summing tasks.

Both methods achieve the same result: they tell you the total sum of all the odd numbers and the total sum of all the even numbers in the collection you provide.