

Lab Assignment 2.1

Task 1: Statistical Summary for Survey Data

Scenario:

- You are a data analyst intern working with survey responses stored as numerical lists.

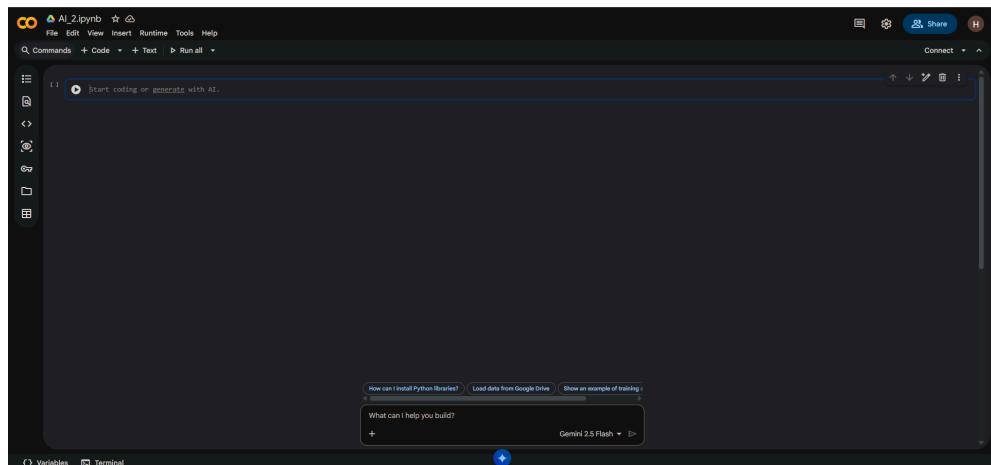
Task:

- Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

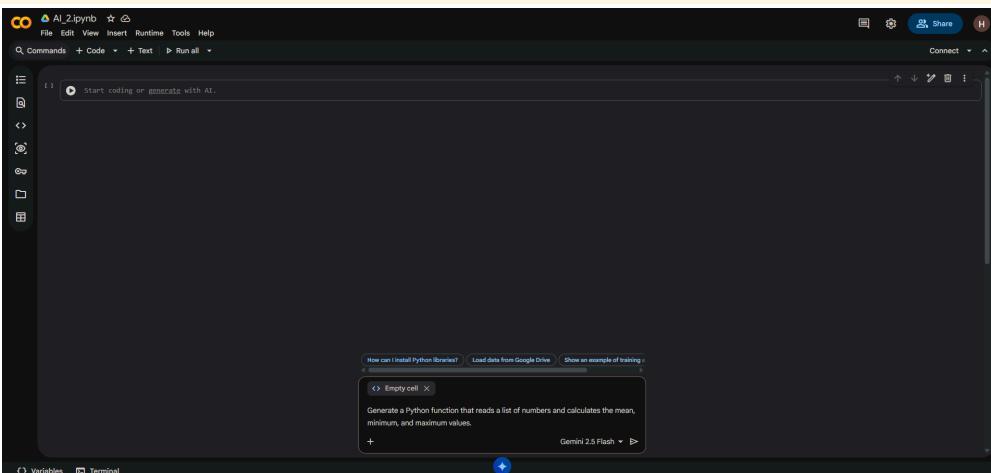
Expected Output:

- Correct Python function
- Output shown in Colab
- Screenshot of Gemini prompt and result

Google Colab Workspace Screenshots:



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



AI_2.ipynb

```
Start coding or generate with AI.
```

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

Working...

What can I help you build?

Gemini 2.5 Flash

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

Variables Terminal

This screenshot shows the Jupyter Notebook interface with a code cell containing the following Python code:

```
#def analyze_numbers(numbers):
#    """
#        Calculates the mean, minimum, and maximum values from a list of numbers.
#    Args:
#        numbers (list): A list of numerical values.
#    Returns:
#        tuple: A tuple containing (mean, minimum_value, maximum_value).
#    """
#    if not numbers:
#        return None, None, None
#    mean_value = sum(numbers) / len(numbers)
#    min_value = min(numbers)
#    max_value = max(numbers)
#    return mean_value, min_value, max_value
```

The cell has a tooltip asking "Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values." Below the cell, a Gemini 2.5 Flash window is open with the prompt "What can I help you build?" and the response "Working...".

AI_2.ipynb

```
+def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum values from a list of numbers.
    Args:
        numbers (list): A list of numerical values.
    Returns:
        tuple: A tuple containing (mean, minimum_value, maximum_value).
    """
    if not numbers:
        return None, None, None
    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)
    return mean_value, min_value, max_value
```

Example usage:
my_list = [10, 20, 30, 40, 50]
mean, minimum, maximum = analyze_numbers(my_list)
print("List: ", my_list)
print("Mean: ", mean)
print("Minimum: ", minimum)
print("Maximum: ", maximum)
#
empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print("List: ", empty_list)
print("Mean: ", mean_empty)
print("Minimum: ", min_empty)
print("Maximum: ", max_empty)

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

Empty cell

Here is a Python function that calculates the mean, minimum, and maximum values from a list of numbers.

Accept & Run Accept Cancel

What can I help you build?

Gemini 2.5 Flash

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

Variables Terminal

This screenshot shows the Jupyter Notebook interface with the same code as the previous one, plus an additional example for an empty list. The tooltip remains the same: "Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values." The Gemini window now includes the generated code for the empty list case and asks "What can I help you build?" with options "Accept & Run" and "Accept".

AI_2.ipynb

```
def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum values from a list of numbers.
    Args:
        numbers (list): A list of numerical values.
    Returns:
        tuple: A tuple containing (mean, minimum_value, maximum_value).
    """
    if not numbers:
        return None, None, None
    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)
    return mean_value, min_value, max_value
```

Example usage:
my_list = [10, 20, 30, 40, 50]
mean, minimum, maximum = analyze_numbers(my_list)
print("List: ", my_list)
print("Mean: ", mean)
print("Minimum: ", minimum)
print("Maximum: ", maximum)
#
empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print("List: ", empty_list)
print("Mean: ", mean_empty)
print("Minimum: ", min_empty)
print("Maximum: ", max_empty)

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

Empty cell

Here is a Python function that calculates the mean, minimum, and maximum values from a list of numbers.

Execute example usage for analyze_numbers Explain the analyze_numbers function

How can I add error handling to this function

What can I help you build?

Gemini 2.5 Flash

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

Variables Terminal

This screenshot shows the Jupyter Notebook interface with the full code and examples. The tooltip is still the same. The Gemini window now includes sections for "Execute example usage for analyze_numbers" and "Explain the analyze_numbers function", along with the previous prompts and responses.

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left, a code cell contains Python code for calculating statistics from a list of numbers. On the right, a Gemini AI sidebar provides documentation and examples for the function. The AI sidebar includes sections for generating the function, emptying the cell, and explaining the code. It also has buttons for executing example usage and explaining the function, and a field for asking what can help build it.

```
[8] def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum values from a list of numbers.

    Args:
        numbers (list): A list of numerical int values.

    Returns:
        tuple: A tuple containing (mean_value, min_value, max_value).

    if not numbers:
        return None, None, None

    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return mean_value, min_value, max_value

try:
    numerical_values = list(map(int, input("Enter numerical values separated by spaces: ").split()))
    if not numerical_values:
        print("No numerical values entered.")
    else:
        mean, minimum, maximum = analyze_numbers(numerical_values)
        print(f"Mean: {mean}")
        print(f"Minimum: {minimum}")
        print(f"Maximum: {maximum}")
except ValueError:
    print("Invalid input. Please enter only numerical int values.")

... Enter numerical values separated by spaces: 3.5 3.5
Invalid Input. Please enter only numerical values.
```

Final Code:

```
def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum values from a list of
    numbers.

    Args:
        numbers (list): A list of numerical int values.

    Returns:
        tuple: A tuple containing (mean_value, min_value, max_value).

    if not numbers:
        return None, None, None

    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return mean_value, min_value, max_value

try:
    numerical_values = list(map(int, input("Enter numerical values separated
    by spaces: ").split()))
    if not numerical_values:
        print("No numerical values entered.")
    else:
```

```

mean, minimum, maximum = analyze_numbers(numerical_values)
print(f"List: {numerical_values}")
print(f"Mean: {mean}")
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")
except ValueError:
    print("Invalid input. Please enter only numerical int values.")

```

Task 2: Armstrong Number – AI Comparison

Scenario:

- You are evaluating AI tools for numeric validation logic.

Task:

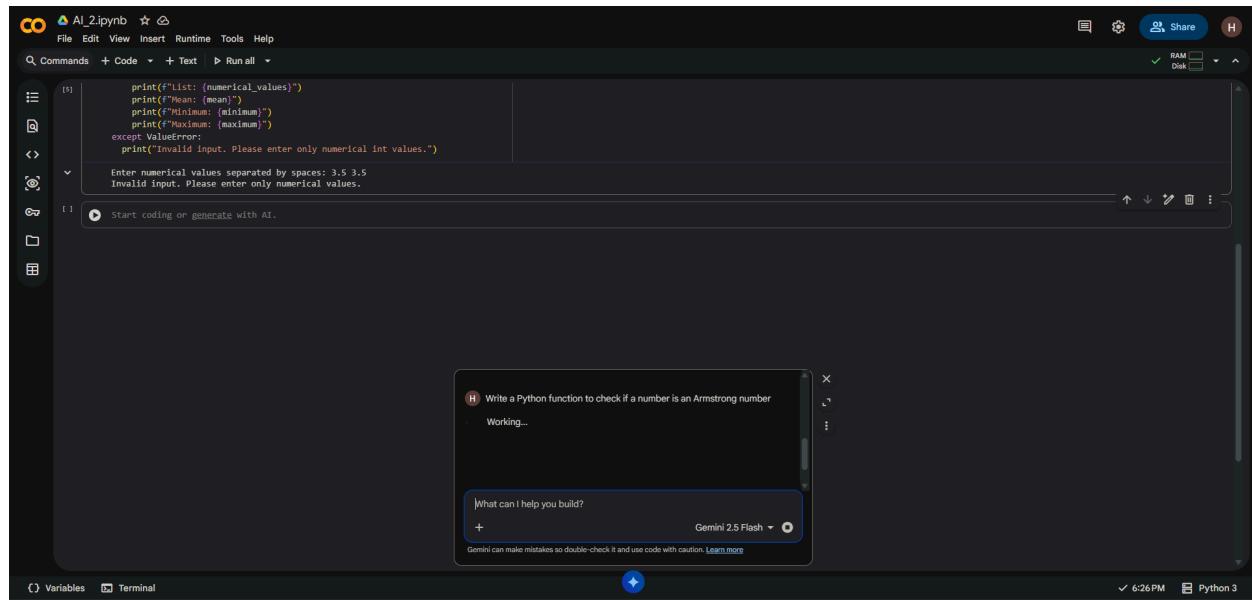
- Generate an Armstrong number checker using Gemini and GitHub Copilot.
- Compare their outputs, logic style, and clarity.

Expected Output:

- Side-by-side comparison table
- Screenshots of prompts and generated code

Google Colab Workspace Screenshots:

Write a Python function to check if a number is an Armstrong number



The screenshot shows the Google Colab interface with a dark theme. On the left, a code cell contains a Python function named `is_armstrong`. The function checks if a given number is an Armstrong number by summing its digits each raised to the power of the number of digits. It includes examples and test cases. On the right, the Gemini AI panel is open, showing a prompt to write a Python function to check for Armstrong numbers, followed by a message from Gemini apologizing for a previous error and providing the correct code.

```

def is_armstrong(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number, plenary number,
    or pluperfect digital invariant) is a number that is the sum of its
    own digits each raised to the power of the number of digits.

    For example, 153 is an Armstrong number because  $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ .

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    if not isinstance(number, int) or number < 0:
        # Armstrong numbers are typically defined for non-negative integers.
        # Returning False for non-integers or negative numbers.
        return False

    num_str = str(number)
    n = len(num_str)
    armstrong_sum = 0
    for digit_char in num_str:
        digit = int(digit_char)
        armstrong_sum += digit ** n

    return armstrong_sum == number

# Example usage:
# print("Is 153 an Armstrong number? {is_armstrong(153)}") # Expected: True
# print("Is 9 an Armstrong number? {is_armstrong(9)}") # Expected: True
# print("Is 370 an Armstrong number? {is_armstrong(370)}") # Expected: True
# print("Is 10 an Armstrong number? {is_armstrong(10)}") # Expected: False
# print("Is 1634 an Armstrong number? {is_armstrong(1634)}") # Expected: True

```

Google Colab Gemini Code:

```

def is_armstrong(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number, plenary
    number,
    or pluperfect digital invariant) is a number that is the sum of its
    own digits each raised to the power of the number of digits.

    For example, 153 is an Armstrong number because  $1^3 + 5^3 + 3^3 = 1 +$ 
     $125 + 27 = 153$ .

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    if not isinstance(number, int) or number < 0:
        # Armstrong numbers are typically defined for non-negative
        # integers.
        # Returning False for non-integers or negative numbers.
        return False

    num_str = str(number)

```

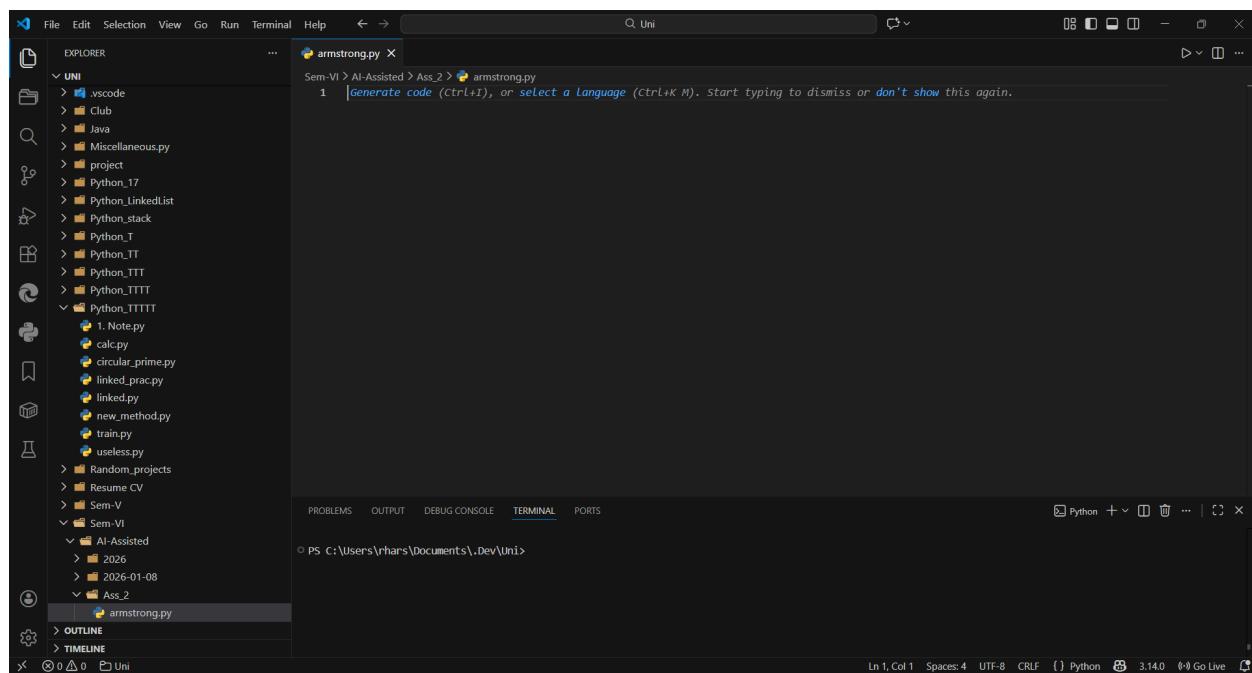
```

n = len(num_str)
armstrong_sum = 0
for digit_char in num_str:
    digit = int(digit_char)
    armstrong_sum += digit ** n

return armstrong_sum == number

# Example usage:
# print(f"Is 153 an Armstrong number? {is_armstrong(153)}") # Expected:
True
# print(f"Is 9 an Armstrong number? {is_armstrong(9)}")      # Expected: True
# print(f"Is 370 an Armstrong number? {is_armstrong(370)}") # Expected:
True
# print(f"Is 10 an Armstrong number? {is_armstrong(10)}")    # Expected:
False
# print(f"Is 1634 an Armstrong number? {is_armstrong(1634)}") # Expected:
True

```



The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a project structure under 'UNI' named 'vscode'. Inside 'vscode' are folders like Club, Java, Miscellaneous.py, project, Python_17, Python_LinkedList, Python_stack, Python_T, Python_TT, Python_TTT, and Python_TTTT. Within Python_TTTT, there are files: 1.Note.py, calc.py, circular_prime.py, linked_pracy.py, linked.py, new_method.py, train.py, and useless.py.
- Editor:** The active file is 'armstrong.py' located at 'Sem-VI > AI-Assisted > Ass_2 > armstrong.py'. The code completion interface is open, displaying the following prompt:

```
Write a Python function to check if a number is an Armstrong number
```
- Terminal:** The terminal shows the command PS C:\Users\rhars\Documents\Dev\Uni and the path PS C:\Users\rhars\Documents\Dev\Uni>.
- Status Bar:** Shows the file is Python, has 3.14.0, and is in Go Live mode.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Same as the first screenshot, showing the 'vscode' project structure.
- Editor:** The active file is 'armstrong.py' with the generated code:

```
def is_armstrong(num):
    """
    Check if a number is an Armstrong number.
    An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits.

    Args:
        num: An integer to check

    Returns:
        bool: True if the number is an Armstrong number, False otherwise
    """
    # convert to string to get number of digits
    num_str = str(abs(num))
    num_digits = len(num_str)

    # calculate sum of digits raised to power of number of digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    # check if it equals the original number
    return sum_of_powers == abs(num)

# Test cases
if __name__ == "__main__":
    test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 123, 100, 0, 1, 9]

    for num in test_numbers:
        if is_armstrong(num):
            print(f"{num} is an Armstrong number")
        else:
            print(f"{num} is not an Armstrong number")
```
- Terminal:** The terminal shows the command PS C:\Users\rhars\Documents\Dev\Uni and the path PS C:\Users\rhars\Documents\Dev\Uni>.
- Status Bar:** Shows the file is Python, has 3.14.0, and is in Go Live mode.

The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar, which lists various projects and files under a 'UNI' folder. A file named 'armstrong.py' is selected in the 'Ass_2' folder. The main pane displays the Python code for the 'is_armstrong' function. The code includes docstrings, type hints, and a test case block.

```

def is_armstrong(num):
    """
    Check if a number is an Armstrong number.
    An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits.
    Args:
        num: An integer to check
    Returns:
        bool: True if the number is an Armstrong number, False otherwise
    """
    # Convert to string to get number of digits
    num_str = str(abs(num))
    num_digits = len(num_str)

    # calculate sum of digits raised to power of number of digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    # check if it equals the original number
    return sum_of_powers == abs(num)

# Test cases
if __name__ == "__main__":
    test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 123, 100, 0, 1, 9]

    for num in test_numbers:
        if is_armstrong(num):
            print(f"{num} is an Armstrong number")
        else:
            print(f"{num} is not an Armstrong number")

```

At the bottom of the editor, status bar information includes: Ln 32, Col 1, Spaces: 4, UTF-8, Python, 3.14.0, Go Live.

Co-pilot Code:

```

def is_armstrong(num):
    """
    Check if a number is an Armstrong number.
    An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits.

    Args:
        num: An integer to check

    Returns:
        bool: True if the number is an Armstrong number, False otherwise
    """
    # Convert to string to get number of digits
    num_str = str(abs(num))
    num_digits = len(num_str)

    # Calculate sum of digits raised to power of number of digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    # Check if it equals the original number
    return sum_of_powers == abs(num)

```

```

# Test cases
if __name__ == "__main__":
    test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 123, 100, 0, 1,
9]

    for num in test_numbers:
        if is_armstrong(num):
            print(f"{num} is an Armstrong number")
        else:
            print(f"{num} is not an Armstrong number")

```

Task 3: Leap Year Validation Using Cursor AI

Scenario:

- You are validating a calendar module for a backend system.

Task:

- Use Cursor AI to generate a Python program that checks whether a given year is a leap year.
- Use at least two different prompts and observe changes in code.

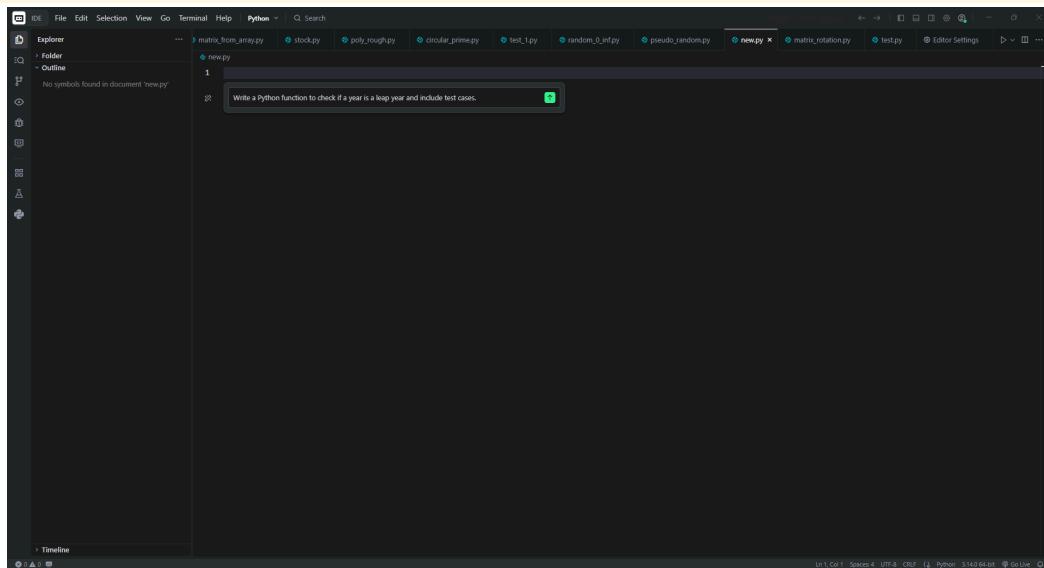
Expected Output:

- Two versions of code
- Sample inputs/outputs
- Brief comparison

Version - 1:

Instruction Given:

Write a Python function to check if a year is a leap year and include test cases.



The screenshot shows a Python IDE interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Terminal, Help, and Python. The left sidebar has sections for Explorer, Outline, and Timeline. The main code editor window contains the following Python code:

```
1 def is_leap_year_basic(year):
2     """
3         Basic leap year checker
4     """
5     if year % 4 == 0:
6         if year % 100 == 0:
7             if year % 400 == 0:
8                 return True
9             else:
10                return False
11        else:
12            return True
13    else:
14        return False
15
16 # Test
17 years = [2000, 2004, 1900, 2024, 2023]
18 for y in years:
19     print(f"{y}: {is_leap_year_basic(y)}")
20
```

The status bar at the bottom indicates Line 16, Col 16, Spaces 4, UTF-8, CR LF, Python 3.14.0 64-bit, and Go Live.

Code:

```
def is_leap_year_basic(year):
    """
        Basic leap year checker
    """

    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            else:
                return False
        else:
            return True
    else:
        return False

# Test
years = [2000, 2004, 1900, 2024, 2023]
for y in years:
    print(f"{y}: {is_leap_year_basic(y)})
```

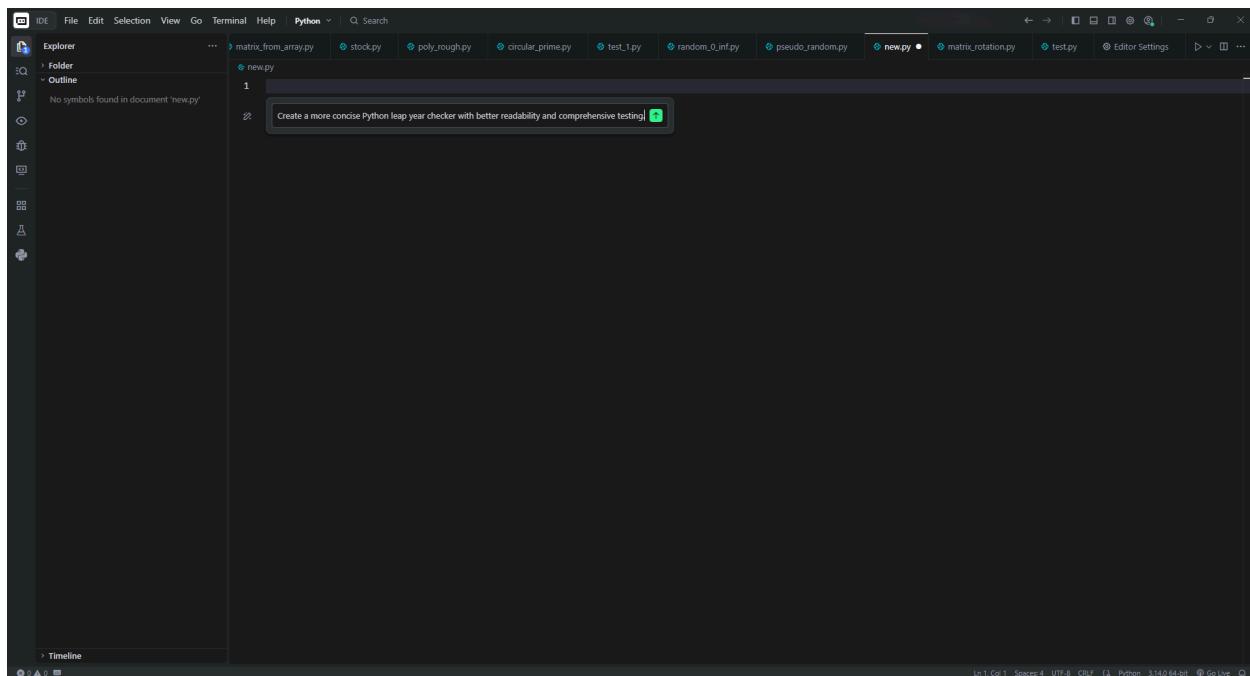
Code Output:

```
2000: True
2004: True
1900: False
2024: True
2023: False
```

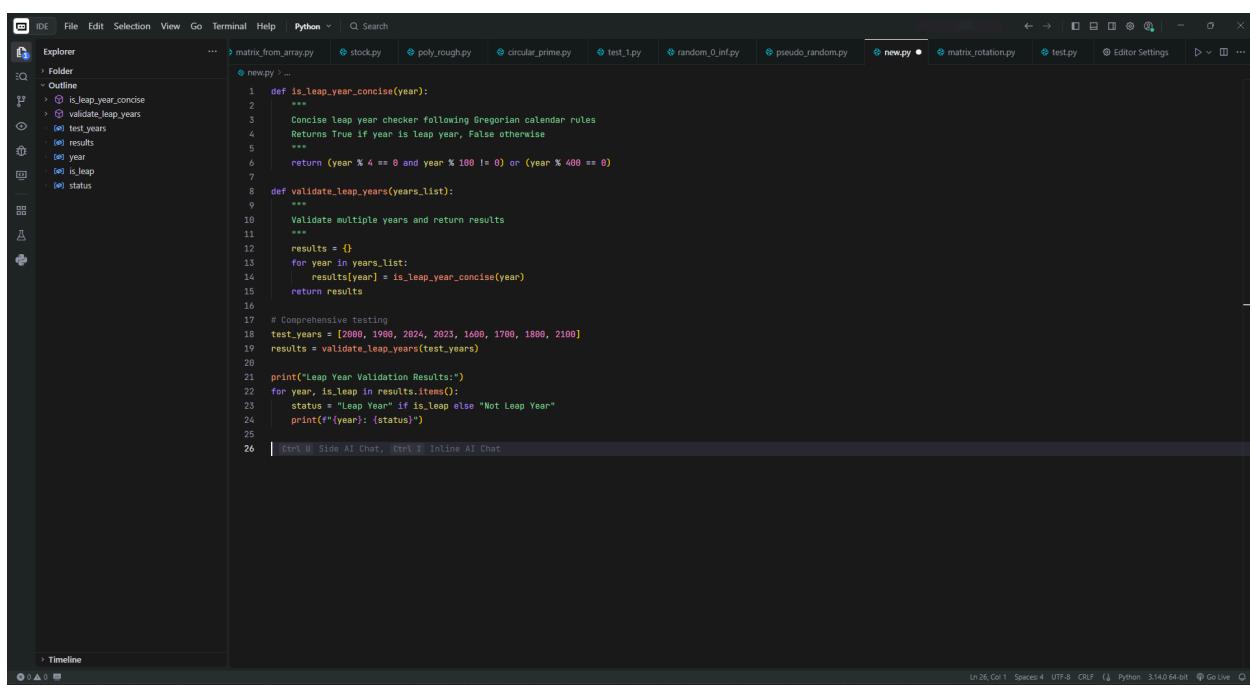
Version - 2:

Instruction Given:

Create a more concise Python leap year checker with better readability and comprehensive testing.



The screenshot shows a dark-themed IDE interface with a Python file named 'new.py' open. A code completion tooltip is displayed over the first line of code, containing the instruction: "Create a more concise Python leap year checker with better readability and comprehensive testing". The code area is mostly blank, with the first line starting with '1'. The left sidebar includes an Explorer panel showing other files like 'matrix_from_array.py', 'stock.py', etc., and an Outline panel which is currently empty. The bottom status bar indicates the file is saved and shows Python 3.14.0 64-bit.



The screenshot shows the same IDE interface with the 'new.py' file now containing the following Python code:

```
def is_leap_year_concise(year):
    """
    Concise leap year checker following Gregorian calendar rules
    Returns True if year is leap year, False otherwise
    """
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

def validate_leap_years(years_list):
    """
    Validate multiple years and return results
    """
    results = {}
    for year in years_list:
        results[year] = is_leap_year_concise(year)
    return results

# Comprehensive testing
test_years = [2000, 1900, 2024, 2023, 1600, 1700, 1800, 2100]
results = validate_leap_years(test_years)
```

The code includes two functions: 'is_leap_year_concise' and 'validate_leap_years'. It also contains a list of test years and a call to the validation function. The code is well-formatted with docstrings and appropriate whitespace. The status bar at the bottom indicates the file has 26 lines and is saved.

Code:

```
def is_leap_year_concise(year):
    """
        Concise leap year checker following Gregorian calendar rules
        Returns True if year is leap year, False otherwise
    """
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

def validate_leap_years(years_list):
    """
        Validate multiple years and return results
    """
    results = {}
    for year in years_list:
        results[year] = is_leap_year_concise(year)
    return results

# Comprehensive testing
test_years = [2000, 1900, 2024, 2023, 1600, 1700, 1800, 2100]
results = validate_leap_years(test_years)

print("Leap Year Validation Results:")
for year, is_leap in results.items():
    status = "Leap Year" if is_leap else "Not Leap Year"
    print(f"{year}: {status}")
```

Code Output:

```
Leap Year Validation Results:
2000: Leap Year
1900: Not Leap Year
2024: Leap Year
2023: Not Leap Year
1600: Leap Year
1700: Not Leap Year
1800: Not Leap Year
2100: Not Leap Year
```

Brief comparison:

Version 1 (Basic)	Version 2 (Concise)
Uses nested if-else statements	Single return statement with logical operators

More lines of code (12 lines)	Fewer lines (5 lines for main function)
Easier for beginners to understand	Follows Python style guide
Step-by-step logic	Combined logical conditions
Includes basic testing	Includes comprehensive testing function

Task 4: Student Logic + AI Refactoring (Odd/Even Sum)

Scenario:

- Company policy requires developers to write logic before using AI.

Task:

- Write a Python program that calculates the sum of odd and even numbers in a tuple, then refactor it using any AI tool.

Expected Output:

- Original code
- Refactored code
- Explanation of improvements

Original Code written by me:

```
def even_odd_sum_tuple(num):
    even_sum = 0
    odd_sum = 0

    for i in range(1, num+1):
        if i%2 == 0:
            even_sum += i
        else:
            odd_sum += i

    return even_sum, odd_sum

user_input = int(input("Enter a number: "))

even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
```

Refactored code:

Instructions given:
Refactor the code `in` order to make it handle input validation,
readability `and` handle edge cases.

```
def even_odd_sum_tuple(n: int) -> tuple[int, int]:  
    """  
        Calculate the sum of even and odd numbers from 1 to n.  
  
    Parameters:  
        n (int): The upper limit (must be a positive integer).  
  
    Returns:  
        tuple[int, int]: (sum_of_even_numbers, sum_of_odd_numbers)  
    """  
    if n < 1:  
        return (0, 0) # No positive range to sum  
  
    even_sum, odd_sum = 0, 0  
  
    for i in range(1, n + 1):  
        if i % 2 == 0:  
            even_sum += i  
        else:  
            odd_sum += i  
  
    return even_sum, odd_sum  
  
try:  
    user_input = int(input("Enter a positive integer: "))  
    if user_input < 1:  
        print("Please enter a positive integer greater than 0.")  
    else:  
        even, odd = even_odd_sum_tuple(user_input)  
        print(f"Sum of even numbers: {even}")  
        print(f"Sum of odd numbers: {odd}")  
  
except ValueError:  
    print("Invalid input! Please enter a valid integer.")
```

Explanation of improvements:

Input validation

- Making sure the user enters a valid integer.
- Handling negative numbers gracefully (return (0,0) or raise an error).

Readability

- Using clear variable names.
- Adding docstrings and comments.

Edge cases

- If input is 0, both sums should be 0.
- If input is negative, we can either reject it or compute sums up to that number (here I'll reject it for clarity).