**2403a51353 Batch-14**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE** | | | | | **DEPARTMENT OF COMPUTER SCIENCE ENGINEERING** | | | | |
| **ProgramName:**B. Tech | | | | **Assignment Type: Lab** | | | **AcademicYear:**2025-2026 | | |
| **CourseCoordinatorName** | | | | Venkataramana Veeramsetty | | | | | |
| **Instructor(s)Name** | | | | |  | | --- | | Dr. V. Venkataramana (Co-ordinator) | | Dr. T. Sampath Kumar | | Dr. Pramoda Patro | | Dr. Brij Kishor Tiwari | | Dr.J.Ravichander | | Dr. Mohammand Ali Shaik | | Dr. Anirodh Kumar | | Mr. S.Naresh Kumar | | Dr. RAJESH VELPULA | | Mr. Kundhan Kumar | | Ms. Ch.Rajitha | | Mr. M Prakash | | Mr. B.Raju | | Intern 1 (Dharma teja) | | Intern 2 (Sai Prasad) | | Intern 3 (Sowmya) | | NS\_2 ( Mounika) | | | | | | |
| **CourseCode** | | | 24CS002PC215 | **CourseTitle** | | AI Assisted Coding | | | |
| **Year/Sem** | | | II/I | **Regulation** | | R24 | | | |
| **Date and Day**  **of Assignment** | | | Week1 - Wednesday | **Time(s)** | |  | | | |
| **Duration** | | | 2 Hours | **Applicableto**  **Batches** | | 24CSBTB01 To 24CSBTB39 | | | |
| **AssignmentNumber:2.3**(Present assignment number)/**24**(Total number of assignments) | | | | | | | | | |
|  | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***ExpectedTime***  ***to complete*** |  |
|  | 1 | Lab 2: Exploring Additional AI Coding Tools – Gemini (Colab) and Cursor AI  **Lab Objectives:**   * To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab. * To understand and use Cursor AI for code generation, explanation, and refactoring. * To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI. * To perform code optimization and documentation using AI tools.   **Lab Outcomes (LOs):**  After completing this lab, students will be able to:   * Generate Python code using Google Gemini in Google Colab. * Analyze the effectiveness of code explanations and suggestions by Gemini. * Set up and use Cursor AI for AI-powered coding assistance. * Evaluate and refactor code using Cursor AI features. * Compare AI tool behavior and code quality across different platforms.   **Task Description#1**   * Use Google Gemini in Colab to write a function that reads a CSV file and calculates mean, min, max.   **Expected Output#1**   * Functional code with output and screenshot         **Output:**    **Task Description#2**   * Compare Gemini and Copilot outputs for a palindrome check function.   **Expected Output#2**   * Side-by-side comparison and observations  |  |  | | --- | --- | | **Gemini:**  def is\_palindrome(text):  """  Checks if a given string is a palindrome.    A palindrome is a word, phrase, or sequence that reads the same backward as forward.  This function ignores capitalization, spaces, and punctuation.  """  # Normalize the string: remove non-alphanumeric characters and convert to lowercase  cleaned\_text = ''.join(char.lower() for char in text if char.isalnum())    # Compare the cleaned string with its reverse  return cleaned\_text == cleaned\_text[::-1]  # Example usage:  print(is\_palindrome("A man, a plan, a canal: Panama"))  print(is\_palindrome("Hello, World!")) | **Copilot:**  def is\_palindrome(s):  # Remove non-alphanumeric characters and convert to lowercase  s = ''.join(filter(str.isalnum, s)).lower()    # Check if the string is equal to its reverse  return s == s[::-1]  # Test cases  print(is\_palindrome("racecar"))  print(is\_palindrome("level"))  print(is\_palindrome("hello"))  print(is\_palindrome("No lemon, no melon")) | | **Output: True**  **Output: False** | **True**  **True**  **False**  **True** |   **Comparision and Observations:**·**Clarity vs. Conciseness:** Gemini prioritizes clarity and education, offering a solution that's easy to understand for beginners. Copilot prioritizes speed and directness, providing a solution that's immediately useful for experienced programmers.  · **Preprocessing:** Both models correctly handle the need for **preprocessing** the input string (removing non-alphanumeric characters and normalizing case). They both use a similar approach of converting the string to lowercase and filtering out unwanted characters before performing the palindrome check.  ·**The Check:** The core logic for checking the palindrome (text == text[::-1]) is identical in both outputs, which is a standard and highly efficient method in Python.  · **Target Audience:** The main difference is the implied audience. Gemini's response is geared toward someone who wants to learn the concept, while Copilot's is for someone who already knows the concept and just needs the code.   |  |  |  | | --- | --- | --- | | **Feature** | **Gemini** | **Copilot** | | **Code style** | Often more concise and leverages Pythonic idioms. | Can be slightly more verbose, but still correct and readable. | | **Explainations** | Provides detailed comments and a descriptive function docstring. | Provides minimal to no explanation beyond the code itself. | | **Use Case** | Ideal for learning, as it explains *why* the code works. | Best for rapid development and code completion. | | **Output** | A complete, well-documented function with examples. | A direct, functional code snippet. |   **Task Description#3**   * Ask Gemini to explain a Python function (to calculate area of various shapes) line by line..   **Expected Output#3**   * Detailed explanation with code snippet   import math  def calculate\_area(shape, \*\*kwargs):      """      Calculates the area of various shapes.      Args:          shape (str): The shape for which to calculate the area ('circle' or 'rectangle').          \*\*kwargs: Keyword arguments for the shape's dimensions.                    For 'circle', provide 'radius'.                    For 'rectangle', provide 'length' and 'width'.      Returns:          float: The calculated area.      Raises:          ValueError: If the shape is not supported or required dimensions are missing.      """      if shape.lower() == 'circle':          if 'radius' in kwargs:              radius = kwargs['radius']              return math.pi \* radius\*\*2          else:              raise ValueError("For a circle, 'radius' must be provided.")      elif shape.lower() == 'rectangle':          if 'length' in kwargs and 'width' in kwargs:              length = kwargs['length']              width = kwargs['width']              return length \* width          else:              raise ValueError("For a rectangle, 'length' and 'width' must be provided.")      else:          raise ValueError("Unsupported shape. Please choose 'circle' or 'rectangle'.")  # Test case 1: Valid circle  try:      circle\_area\_test = calculate\_area('circle', radius=7)      print(f"Test Case 1 (Circle): Area = {circle\_area\_test}")  except ValueError as e:      print(f"Test Case 1 (Circle): Error - {e}")  # Test case 2: Valid rectangle  try:      rectangle\_area\_test = calculate\_area('rectangle', length=8, width=4)      print(f"Test Case 2 (Rectangle): Area = {rectangle\_area\_test}")  except ValueError as e:      print(f"Test Case 2 (Rectangle): Error - {e}")  # Test case 3: Circle with missing radius  try:      calculate\_area('circle')  except ValueError as e:      print(f"Test Case 3 (Circle - Missing Radius): Caught expected error - {e}")  # Test case 4: Rectangle with missing length  try:      calculate\_area('rectangle', width=5)  except ValueError as e:      print(f"Test Case 4 (Rectangle - Missing Length): Caught expected error - {e}")  # Test case 5: Rectangle with missing width  try:      calculate\_area('rectangle', length=10)  except ValueError as e:      print(f"Test Case 5 (Rectangle - Missing Width): Caught expected error - {e}")  # Test case 6: Unsupported shape  try:      calculate\_area('triangle', base=5, height=10)  except ValueError as e:      print(f"Test Case 6 (Unsupported Shape): Caught expected error - {e}")  # Test case 7: Circle with radius 0  try:      circle\_area\_zero\_radius = calculate\_area('circle', radius=0)      print(f"Test Case 7 (Circle - Radius 0): Area = {circle\_area\_zero\_radius}")  except ValueError as e:      print(f"Test Case 7 (Circle - Radius 0): Error - {e}")  # Test case 8: Rectangle with length 0  try:      rectangle\_area\_zero\_length = calculate\_area('rectangle', length=0, width=5)      print(f"Test Case 8 (Rectangle - Length 0): Area = {rectangle\_area\_zero\_length}")  except ValueError as e:      print(f"Test Case 8 (Rectangle - Length 0): Error - {e}")  # Test case 9: Rectangle with width 0  try:      rectangle\_area\_zero\_width = calculate\_area('rectangle', length=10, width=0)      print(f"Test Case 9 (Rectangle - Width 0): Area = {rectangle\_area\_zero\_width}")  except ValueError as e:      print(f"Test Case 9 (Rectangle - Width 0): Error - {e}")  **Task Description#4**   * Install and configure Cursor AI. Use it to generate a Python function (e.g., sum of squares).   **Expected Output#4**   * Screenshots of working environments with few prompts to generate python code   def sum\_of\_squares(*numbers*):      """      Calculate the sum of squares of a list of numbers.        Args:          numbers (list): A list of numbers (integers or floats)        Returns:          float: The sum of squares of all numbers in the list        Examples:  *>>>* sum\_of\_squares([1, 2, 3])          14.0  *>>>* sum\_of\_squares([2, 4, 6, 8])          120.0      """  *if* not *numbers*:  *return* 0.0    *return* sum(num \*\* 2 *for* num *in* *numbers*)  def sum\_of\_squares\_range(*start*, *end*):      """      Calculate the sum of squares for a range of consecutive integers.        Args:          start (int): Starting number (inclusive)          end (int): Ending number (inclusive)        Returns:          float: The sum of squares from start to end        Examples:  *>>>* sum\_of\_squares\_range(1, 3)          14.0  *>>>* sum\_of\_squares\_range(1, 5)          55.0      """  *if* *start* > *end*:  *return* 0.0    *return* sum(i \*\* 2 *for* i *in* range(*start*, *end* + 1))  *# Example usage and testing*  *if* \_\_name\_\_ == "\_\_main\_\_":  *# Test with a list of numbers*      test\_numbers = [1, 2, 3, 4, 5]      result1 = sum\_of\_squares(test\_numbers)      print(f"Sum of squares of {test\_numbers}: {result1}")    *# Test with a range*      start, end = 1, 5      result2 = sum\_of\_squares\_range(start, end)      print(f"Sum of squares from {start} to {end}: {result2}")    *# Test with empty list*      empty\_result = sum\_of\_squares([])      print(f"Sum of squares of empty list: {empty\_result}")    *# Test with negative numbers*      negative\_numbers = [-2, -1, 0, 1, 2]      result3 = sum\_of\_squares(negative\_numbers)      print(f"Sum of squares of {negative\_numbers}: {result3}")    **Task Description#5**   * Student need to write code to calculate sum of odd number and even numbers in the list   **Expected Output#5**   * Refactored code written by student with improved logic   def calculate\_odd\_even\_sums\_basic(*numbers*):      """      Basic implementation to calculate sum of odd and even numbers in a list.        Args:          numbers (list): A list of integers        Returns:          tuple: (sum\_of\_odd, sum\_of\_even)        Example:  *>>>* calculate\_odd\_even\_sums\_basic([1, 2, 3, 4, 5])          (9, 6)      """      odd\_sum = 0      even\_sum = 0    *for* num *in* *numbers*:  *if* num % 2 == 0:  *# Even number*              even\_sum += num  *else*:  *# Odd number*              odd\_sum += num    *return* odd\_sum, even\_sum  def calculate\_odd\_even\_sums\_refactored(*numbers*):      """      Refactored implementation with improved logic using list comprehensions and built-in functions.        Args:          numbers (list): A list of integers        Returns:          tuple: (sum\_of\_odd, sum\_of\_even)        Example:  *>>>* calculate\_odd\_even\_sums\_refactored([1, 2, 3, 4, 5])          (9, 6)      """  *if* not *numbers*:  *return* 0, 0    *# Use list comprehensions with sum() for better performance*      odd\_sum = sum(num *for* num *in* *numbers* *if* num % 2 != 0)      even\_sum = sum(num *for* num *in* *numbers* *if* num % 2 == 0)    *return* odd\_sum, even\_sum  def calculate\_odd\_even\_sums\_advanced(*numbers*):      """      Advanced implementation with additional features and error handling.        Args:          numbers (list): A list of numbers (integers or floats)        Returns:          dict: Dictionary with detailed results including counts and averages        Example:  *>>>* calculate\_odd\_even\_sums\_advanced([1, 2, 3, 4, 5])          {'odd\_sum': 9, 'even\_sum': 6, 'odd\_count': 3, 'even\_count': 2, 'odd\_avg': 3.0, 'even\_avg': 3.0}      """  *if* not *numbers*:  *return* {              'odd\_sum': 0, 'even\_sum': 0,              'odd\_count': 0, 'even\_count': 0,              'odd\_avg': 0.0, 'even\_avg': 0.0          }    *# Separate odd and even numbers*      odd\_numbers = [num *for* num *in* *numbers* *if* num % 2 != 0]      even\_numbers = [num *for* num *in* *numbers* *if* num % 2 == 0]    *# Calculate sums*      odd\_sum = sum(odd\_numbers)      even\_sum = sum(even\_numbers)    *# Calculate counts*      odd\_count = len(odd\_numbers)      even\_count = len(even\_numbers)    *# Calculate averages (avoid division by zero)*      odd\_avg = odd\_sum / odd\_count *if* odd\_count > 0 *else* 0.0      even\_avg = even\_sum / even\_count *if* even\_count > 0 *else* 0.0    *return* {          'odd\_sum': odd\_sum,          'even\_sum': even\_sum,          'odd\_count': odd\_count,          'even\_count': even\_count,          'odd\_avg': round(odd\_avg, 2),          'even\_avg': round(even\_avg, 2)      }  def analyze\_number\_distribution(*numbers*):      """      Analyze the distribution of numbers in the list with additional insights.        Args:          numbers (list): A list of numbers        Returns:          dict: Comprehensive analysis of the number list      """  *if* not *numbers*:  *return* {'error': 'Empty list provided'}    *# Basic odd/even analysis*      odd\_even\_result = calculate\_odd\_even\_sums\_advanced(*numbers*)    *# Additional analysis*      total\_sum = sum(*numbers*)      total\_count = len(*numbers*)      max\_num = max(*numbers*)      min\_num = min(*numbers*)    *# Find the largest odd and even numbers*      odd\_numbers = [num *for* num *in* *numbers* *if* num % 2 != 0]      even\_numbers = [num *for* num *in* *numbers* *if* num % 2 == 0]        largest\_odd = max(odd\_numbers) *if* odd\_numbers *else* None      largest\_even = max(even\_numbers) *if* even\_numbers *else* None      smallest\_odd = min(odd\_numbers) *if* odd\_numbers *else* None      smallest\_even = min(even\_numbers) *if* even\_numbers *else* None    *return* {          \*\*odd\_even\_result,          'total\_sum': total\_sum,          'total\_count': total\_count,          'overall\_average': round(total\_sum / total\_count, 2),          'max\_number': max\_num,          'min\_number': min\_num,          'largest\_odd': largest\_odd,          'largest\_even': largest\_even,          'smallest\_odd': smallest\_odd,          'smallest\_even': smallest\_even,          'range': max\_num - min\_num      }  *# Example usage and testing*  *if* \_\_name\_\_ == "\_\_main\_\_":  *# Test data*      test\_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]      test\_numbers2 = [11, 22, 33, 44, 55, 66, 77, 88, 99, 100]      test\_numbers3 = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]        print("=== Basic Implementation ===")      odd\_sum, even\_sum = calculate\_odd\_even\_sums\_basic(test\_numbers)      print(f"Numbers: {test\_numbers}")      print(f"Sum of odd numbers: {odd\_sum}")      print(f"Sum of even numbers: {even\_sum}")      print()        print("=== Refactored Implementation ===")      odd\_sum, even\_sum = calculate\_odd\_even\_sums\_refactored(test\_numbers)      print(f"Numbers: {test\_numbers}")      print(f"Sum of odd numbers: {odd\_sum}")      print(f"Sum of even numbers: {even\_sum}")      print()        print("=== Advanced Implementation ===")      result = calculate\_odd\_even\_sums\_advanced(test\_numbers2)      print(f"Numbers: {test\_numbers2}")  *for* key, value *in* result.items():          print(f"{key}: {value}")      print()        print("=== Comprehensive Analysis ===")      analysis = analyze\_number\_distribution(test\_numbers3)      print(f"Numbers: {test\_numbers3}")  *for* key, value *in* analysis.items():          print(f"{key}: {value}")      print()    *# Test with edge cases*      print("=== Edge Cases ===")      print("Empty list:")      print(calculate\_odd\_even\_sums\_refactored([]))        print("\nSingle number (odd):")      print(calculate\_odd\_even\_sums\_refactored([7]))        print("\nSingle number (even):")      print(calculate\_odd\_even\_sums\_refactored([8]))  **Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots**  **Evaluation Criteria:**   | **Criteria** | **Max Marks** | | --- | --- | | Successful Use of Gemini in Colab (Task#1 & #2) | 1.0 | | Code Explanation Accuracy (Gemini) (Task#3) | 0.5 | | Cursor AI Setup and Usage (Task#4) | 0.5 | | Refactoring and Improvement Analysis (Task#5) | 0.5 | | **Total** | **2.5 Marks** | | | | | | | Week1 - Wednesday |  |