

NAME : Tuhin John
BRANCH/SECTION: AIML/'C'
USN: 22BTRCL157

1. Create a Python script that takes a student's score (0-100) as input and prints their grade based on the following criteria:

Above 90: "Grade: A"

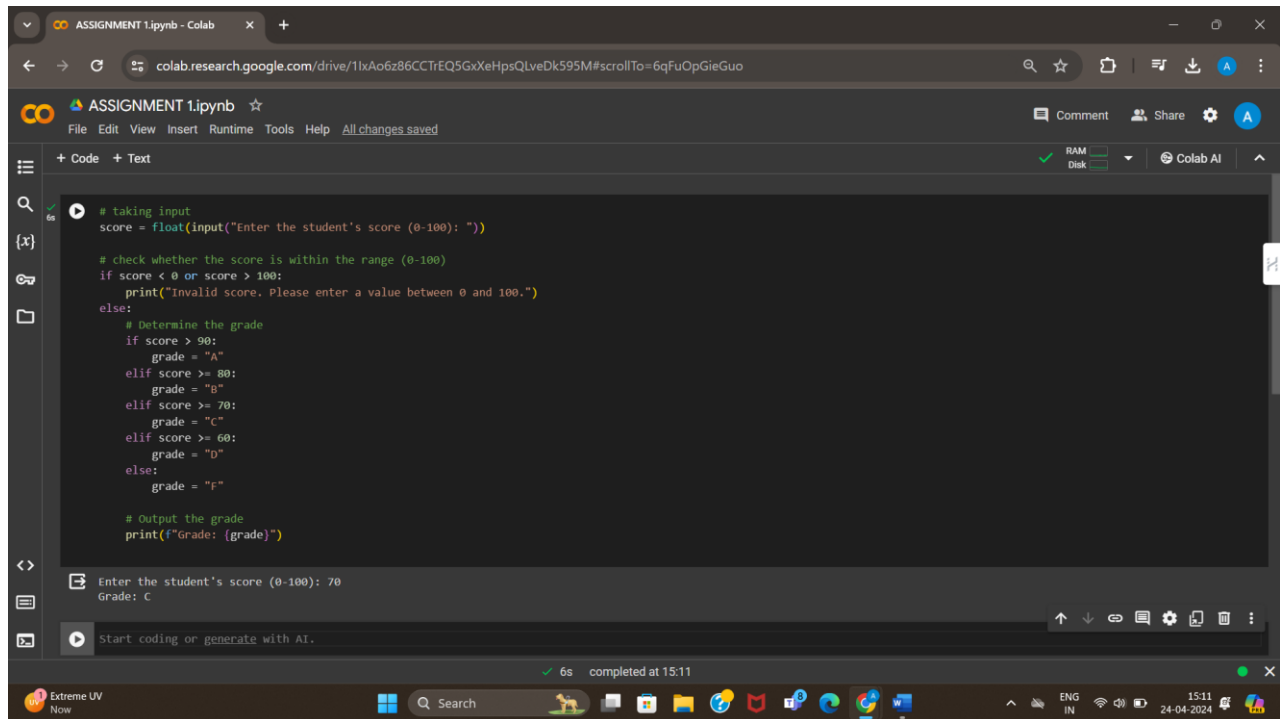
80 to 90: "Grade: B"

70 to 79: "Grade: C"

60 to 69: "Grade: D"

Below 60: "Grade: F"

OUTPUT:



```
# taking input
score = float(input("Enter the student's score (0-100): "))

# check whether the score is within the range (0-100)
if score < 0 or score > 100:
    print("Invalid score. Please enter a value between 0 and 100.")
else:
    # Determine the grade
    if score > 90:
        grade = "A"
    elif score >= 80:
        grade = "B"
    elif score >= 70:
        grade = "C"
    elif score >= 60:
        grade = "D"
    else:
        grade = "F"

# Output the grade
print(f"Grade: {grade}")
```

Enter the student's score (0-100): 70
Grade: C

2. Create a Python program that applies a discount to a purchase based on the amount spent. The program asks for the total amount and applies the following discount rates:

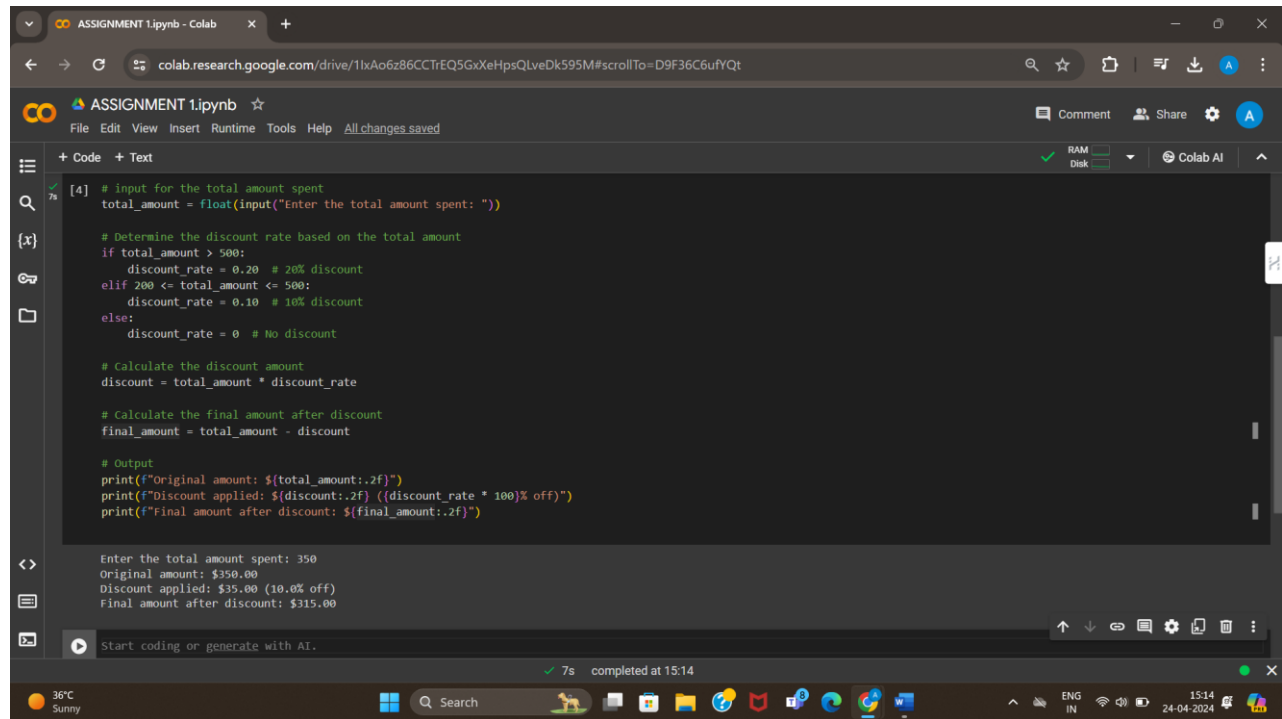
Spend over \$500: 20% discount

Spend \$200 - \$500: 10% discount

Spend below \$200: No discount

The program should print the original amount, the discount applied, and the final amount after the discount.

OUTPUT:



```
[4] # input for the total amount spent
total_amount = float(input("Enter the total amount spent: "))

# Determine the discount rate based on the total amount
if total_amount > 500:
    discount_rate = 0.20 # 20% discount
elif 200 <= total_amount <= 500:
    discount_rate = 0.10 # 10% discount
else:
    discount_rate = 0 # No discount

# Calculate the discount amount
discount = total_amount * discount_rate

# Calculate the final amount after discount
final_amount = total_amount - discount

# Output
print(f"Original amount: ${total_amount:.2f}")
print(f"Discount applied: ${discount:.2f} ({discount_rate * 100}% off)")
print(f"Final amount after discount: ${final_amount:.2f}")

Enter the total amount spent: 350
Original amount: $350.00
Discount applied: $35.00 (10.0% off)
Final amount after discount: $315.00
```

3. Create a program that asks for the user's birth month and day and then tells them their zodiac sign. For simplicity, you can use the following date ranges:

Aries: March 21 - April 19

Taurus: April 20 - May 20

Gemini: May 21 - June 20

Cancer: June 21 - July 22

Leo: July 23 - August 22

Virgo: August 23 - September 22

Libra: September 23 - October 22

Scorpio: October 23 - November 21

Sagittarius: November 22 - December 21

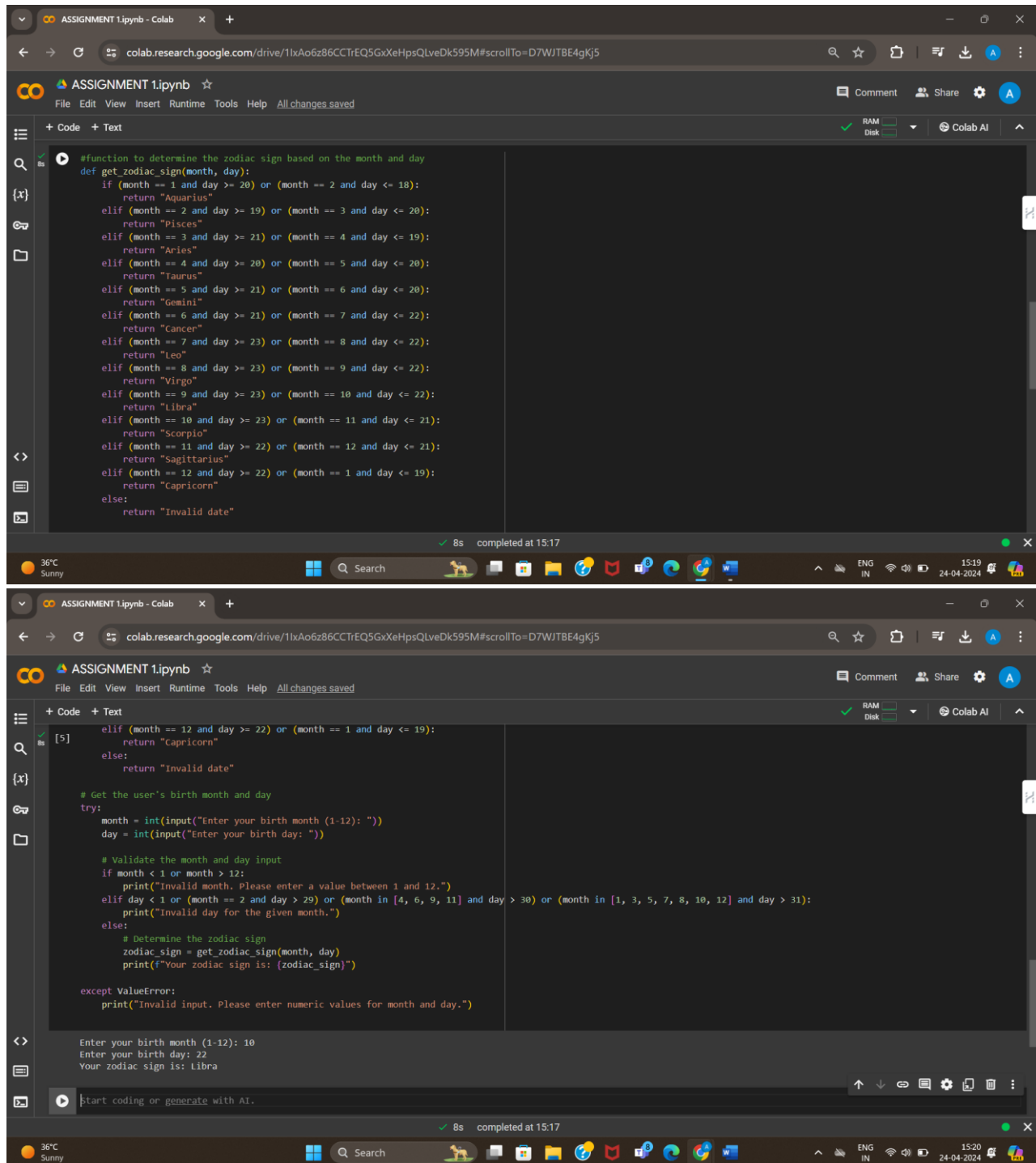
Capricorn: December 22 - January 19

Aquarius: January 20 - February 18

Pisces: February 19 - March 20

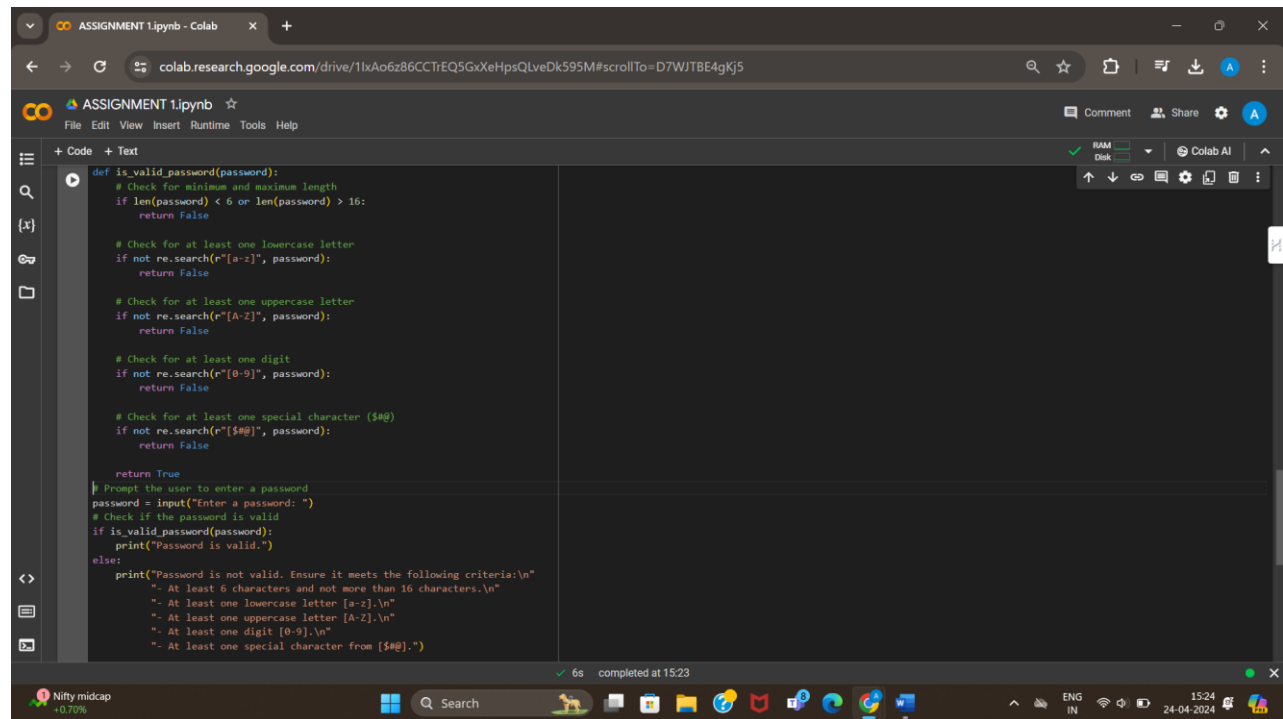
Make sure to handle invalid inputs gracefully.

OUTPUT:



4. Write a Python program to check the validity of a password entered by the user. The password is considered valid if it meets the following criteria:
 - At least 1 letter between [a-z] and 1 letter between [A-Z].
 - At least 1 number between [0-9].
 - At least 1 character from [\$#@].
 - Minimum length of 6 characters.
 - Maximum length of 16 characters.
 The program should print whether the password is valid or not based on these criteria.

OUTPUT:



The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code defines a function `is_valid_password(password)` that checks for minimum and maximum length, at least one lowercase letter, at least one uppercase letter, at least one digit, and at least one special character. It then prompts the user to enter a password and checks if it is valid. If valid, it prints "Password is valid."; otherwise, it prints "Password is not valid. Ensure it meets the following criteria:" followed by a list of requirements.

```
def is_valid_password(password):  
    # Check for minimum and maximum length  
    if len(password) < 6 or len(password) > 16:  
        return False  
  
    # Check for at least one lowercase letter  
    if not re.search(r"[a-z]", password):  
        return False  
  
    # Check for at least one uppercase letter  
    if not re.search(r"[A-Z]", password):  
        return False  
  
    # Check for at least one digit  
    if not re.search(r"[0-9]", password):  
        return False  
  
    # Check for at least one special character ($#@)  
    if not re.search(r"[$#@]", password):  
        return False  
  
    return True  
  
# Prompt the user to enter a password  
password = input("Enter a password: ")  
# Check if the password is valid  
if is_valid_password(password):  
    print("Password is valid.")  
else:  
    print("Password is not valid. Ensure it meets the following criteria:\n"  
          "- At least 6 characters and not more than 16 characters.\n"  
          "- At least one lowercase letter [a-z].\n"  
          "- At least one uppercase letter [A-Z].\n"  
          "- At least one digit [0-9].\n"  
          "- At least one special character from [$#@].")
```

```
Enter a password: av1122  
Password is not valid. Ensure it meets the following criteria:  
- At least 6 characters and not more than 16 characters.  
- At least one lowercase letter [a-z].  
- At least one uppercase letter [A-Z].  
- At least one digit [0-9].  
- At least one special character from [$#@].
```

```
Enter a password: Av@1012  
Password is valid.
```

- Implement a simple number guessing game. First, set a target number within a certain range (e.g., 1 to 100). Then, using a while loop, ask the user to guess the number. Provide feedback for each guess ("too high" or "too low"). The game ends when the user guesses the number correctly. Use a break statement to exit the loop once the correct number is guessed.

OUTPUT:

The first screenshot shows the code for a number guessing game in a Google Colab notebook. The code imports the random module, sets a target number between 1 and 100, and prints a welcome message. It then enters a while loop where the user is prompted to guess the number. Feedback is provided for each guess, and the loop breaks when the correct number is guessed. An exception handler catches ValueError for non-integer inputs.

```
import random

# Set a random target number between 1 and 100
target_number = random.randint(1, 100)

print("Welcome to the number guessing game!")
print("I have selected a number between 1 and 100. Try to guess it!")

# Start a while loop to allow multiple guesses
while True:
    # Ask the user for their guess
    try:
        user_guess = int(input("Enter your guess: "))

        # Provide feedback on the guess
        if user_guess < target_number:
            print("Too low! Try again.")
        elif user_guess > target_number:
            print("Too high! Try again.")
        else:
            print(f"Congratulations! You guessed the correct number: {target_number}")
            break # Exit the loop when the guess is correct
    except ValueError:
        print("Invalid input. Please enter a whole number.")
```

The second screenshot shows the output of the code. It displays the welcome message, the target number range, and the user's guesses. The output shows that the user guessed 43, 11, 22, 30, 37, 35, 33, 32, and 31. The game ends with a congratulatory message for guessing the correct number, 31.

```
Welcome to the number guessing game!
I have selected a number between 1 and 100. Try to guess it!
Enter your guess: 43
Too high! Try again.
Enter your guess: 11
Too low! Try again.
Enter your guess: 22
Too low! Try again.
Enter your guess: 30
Too low! Try again.
Enter your guess: 37
Too high! Try again.
Enter your guess: 35
Too high! Try again.
Enter your guess: 33
Too high! Try again.
Enter your guess: 32
Too high! Try again.
Enter your guess: 31
Congratulations! You guessed the correct number: 31
```

- Write a Python program that asks the user to enter a range (start and end numbers). Use a for loop to iterate through this range, and for each number, check if it is a prime number. If it is, print the number. Use the **continue** statement to skip non-prime numbers efficiently.

OUTPUT:

The first screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code defines a function `is_prime(n)` that checks if a number is prime. It then asks the user for a range and prints the prime numbers in that range. The code is as follows:

```
# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True # 2 and 3 are prime numbers
    if n % 2 == 0 or n % 3 == 0:
        return False # Multiples of 2 and 3 are not prime

    # Check divisibility by all odd numbers from 5 to sqrt(n)
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

# Ask the user for a range
try:
    start = int(input("Enter the start of the range: "))
    end = int(input("Enter the end of the range: "))

    if start > end:
        print("Start of the range should be less than or equal to the end.")
    else:
        print(f"Prime numbers between {start} and {end}:")

        # Iterate through the range and print prime numbers
        for num in range(start, end + 1):
            if is_prime(num):
                print(num)
            else:
                continue # Continue to the next iteration if not prime
except ValueError:
    print("Invalid input. Please enter whole numbers for the range.")
```

The second screenshot shows the output of the program. The user entered the start of the range as 30 and the end of the range as 70. The program printed the prime numbers between 30 and 70: 31, 37, 41, 43, 47, 53, 59, 61, and 67.

7. Create a Python program that iterates through a list of numbers (you can define the list in the code) and calculates the sum of the numbers. However, if the program encounters a number that is negative, it should stop adding any further numbers (i.e., break out of the loop) and print the current sum up to that point.

OUTPUT:

ASSIGNMENT 1.ipynb - Colab

colab.research.google.com/drive/1lxAo6z86CCTrEQ5GxXeHpsQLveDk595M#scrollTo=BFth89uwj6sJ

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Define a list of numbers
numbers = [22, 11, 10, 40, -27, 25, 35]

# Initialize a variable to hold the running sum
total_sum = 0

# Iterate through the list of numbers
for number in numbers:
    # If a negative number is encountered, break the loop
    if number < 0:
        print(f"Encountered a negative number ({number}). Stopping calculation.")
        break

    # Add the current number to the running sum
    total_sum += number

# Output the current sum
print(f"The sum of the numbers up to the negative value is: {total_sum}")
```

Encountered a negative number (-27). Stopping calculation.
The sum of the numbers up to the negative value is: 83

Start coding or generate with AI.

0s completed at 15:38

8. Write a Python program to print the following patterns

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

ASSIGNMENT 1.ipynb - Colab

colab.research.google.com/drive/1lxAo6z86CCTrEQ5GxXeHpsQLveDk595M#scrollTo=OnoHrcailm3

File Edit View Insert Runtime Tools Help

+ Code + Text

```
[17] # Function to print the Right Pascal's Triangle
def print_right_pascals_triangle(rows):
    # First part: Increasing triangle
    for i in range(1, rows + 1):
        # Print numbers from 1 to i
        for j in range(1, i + 1):
            print(j, end=" ") # Print each number followed by a space
        print() # Move to the next line

    # Second part: Decreasing triangle
    for i in range(rows - 1, 0, -1):
        # Print numbers from 1 to i
        for j in range(1, i + 1):
            print(j, end=" ") # Print each number followed by a space
        print() # Move to the next line

    # Ask the user for the number of rows
    try:
        rows = int(input("Enter the number of rows for the Right Pascal's Triangle: "))

        if rows < 1:
            print("Please enter a positive number of rows.")
        else:
            # Print the Right Pascal's Triangle
            print_right_pascals_triangle(rows)

    except ValueError:
        print("Invalid input. Please enter an integer.")
```

17s completed at 15:44

CODE:

```

1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21

```

CODE:

```

# Function to print the left half triangle with sequential numbers
def print_left_half_triangle(rows):
    # Start with a number and increment it with each iteration
    current_number = 1
    # Loop through each row
    for i in range(1, rows + 1):
        # Print the numbers for each row
        for j in range(i):
            print(current_number, end=" ") # Print the current number followed by a space
            current_number += 1 # Increment the number
        print() # Move to the next line after each row
    # Ask the user for the number of rows
    try:
        rows = int(input("Enter the number of rows for the left half triangle: "))

        if rows < 1:
            print("Please enter a positive number of rows.")
        else:
            # Print the left half triangle
            print_left_half_triangle(rows)
    except ValueError:
        print("Invalid input. Please enter an integer.")

```

Enter the number of rows for the left half triangle: 6

```

1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21

```

9. Create a program that asks for two numbers and prints all the numbers between them that are divisible by a third number asked from the user.

OUTPUT:

```

# Function to find numbers divisible by a given divisor within a specified range
def print_divisible_numbers(start, end, divisor):
    # Loop through the range of numbers
    for num in range(start, end + 1):
        # Check if the number is divisible by the divisor
        if num % divisor == 0:
            print(num, end=" ") # Print the number followed by a space
    print()

# Ask the user for two numbers to define the range
try:
    start = int(input("Enter the first number to define the range: "))
    end = int(input("Enter the second number to define the range: "))

    if start > end:
        # Ensure the first number is less than or equal to the second number
        start, end = end, start # Swap if the start is greater than the end
    # Ask for the divisor
    divisor = int(input("Enter the divisor: "))
    if divisor == 0:
        print("Divisor cannot be zero.")
    else:
        print("Numbers between (start) and (end) that are divisible by (divisor):")
        # Print the numbers in the range that are divisible by the divisor
        print_divisible_numbers(start, end, divisor)
except ValueError:
    print("Invalid input. Please enter valid integers.")

```

Enter the first number to define the range: 1

Enter the second number to define the range: 22

Enter the divisor: 3

Numbers between 1 and 22 that are divisible by 3:

```

3 6 9 12 15 18 21

```


10. Write a recursive function named `reverse_string` that takes a string as input and returns its reverse. The function must use recursion to accomplish this task and should not use any loops or slicing (`[::-1]`).

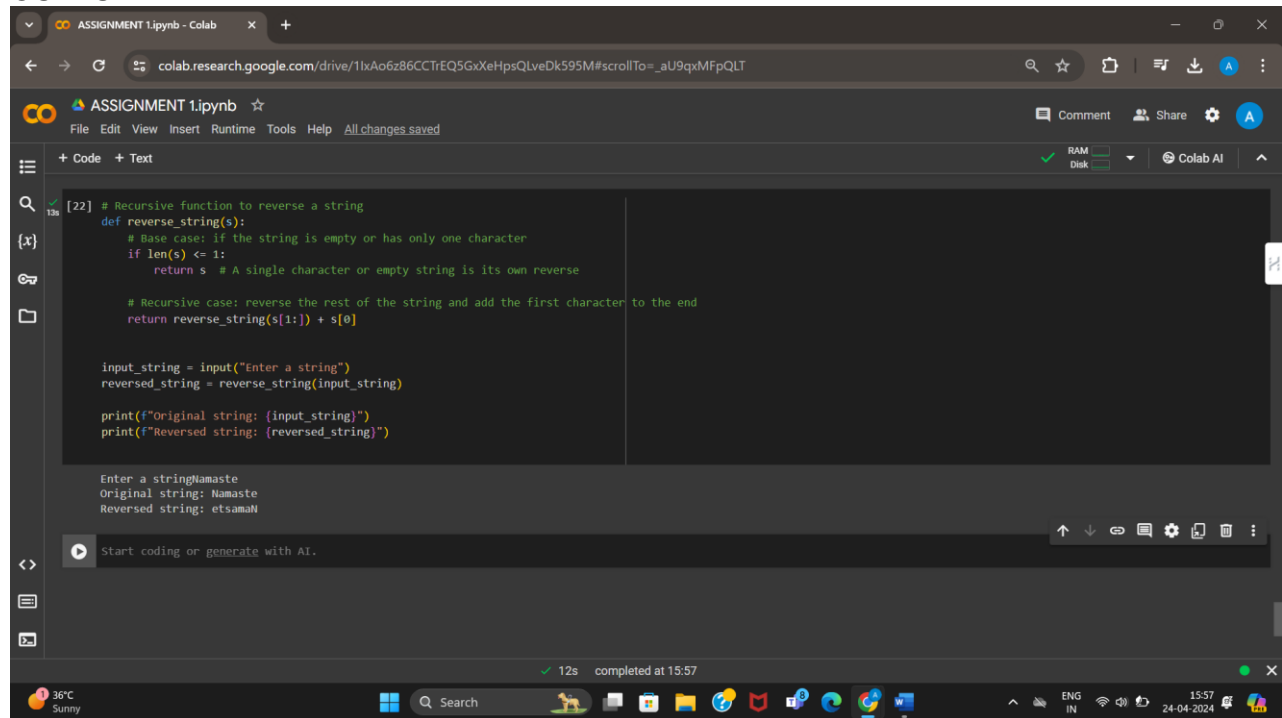
Example Usage:

```
print(reverse_string("hello"))
```

Expected Output:

"olleh"

OUTPUT:



The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code cell [22] contains a recursive function `reverse_string(s)`. The function has a base case where if the string length is 1 or less, it returns the string itself. The recursive case reverses the string from index 1 to the end and appends the first character to the end. Below the function, there is an input prompt and two print statements. The output shows the input string "Namaste" being reversed to "etsamaN".

```
[22] # Recursive function to reverse a string
def reverse_string(s):
    # Base case: if the string is empty or has only one character
    if len(s) <= 1:
        return s # A single character or empty string is its own reverse

    # Recursive case: reverse the rest of the string and add the first character to the end
    return reverse_string(s[1:]) + s[0]

input_string = input("Enter a string")
reversed_string = reverse_string(input_string)

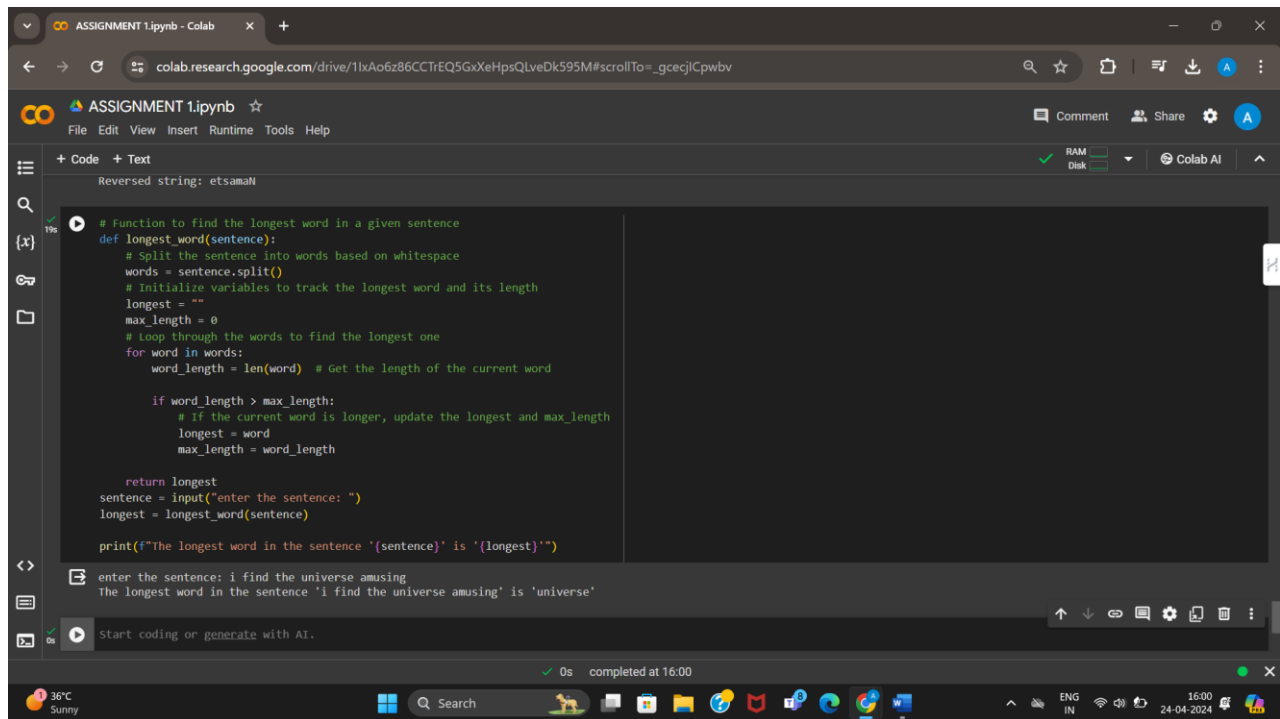
print(f"Original string: {input_string}")
print(f"Reversed string: {reversed_string}")

Enter a stringNamaste
Original string: Namaste
Reversed string: etsamaN
```

11. Create a function `longest_word(sentence)` that finds and returns the longest word in the given string sentence. If there are multiple words of the same length, return the first one encountered.

Example: `longest_word("I love programming")` should return "programming"

OUTPUT:



```
# Function to find the longest word in a given sentence
def longest_word(sentence):
    # Split the sentence into words based on whitespace
    words = sentence.split()
    # Initialize variables to track the longest word and its length
    longest = ""
    max_length = 0
    # Loop through the words to find the longest one
    for word in words:
        word_length = len(word) # Get the length of the current word

        if word_length > max_length:
            # If the current word is longer, update the longest and max_length
            longest = word
            max_length = word_length

    return longest
sentence = input("enter the sentence: ")
longest = longest_word(sentence)

print(f"The longest word in the sentence '{sentence}' is '{longest}'")
```

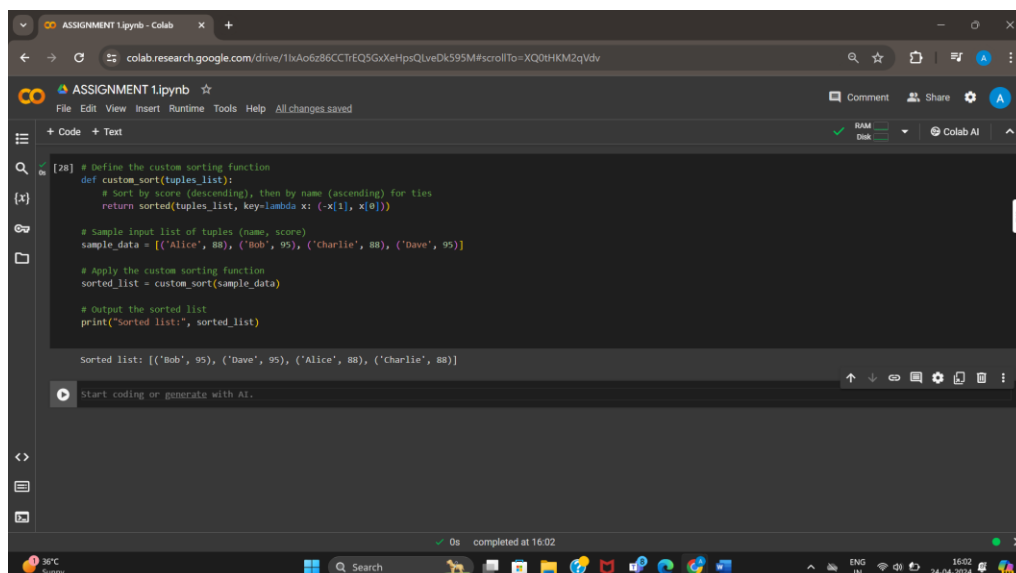
enter the sentence: i find the universe amusing
The longest word in the sentence 'i find the universe amusing' is 'universe'

12. Create a Python function named `custom_sort` that takes a list of tuples where each tuple contains a name and a score. The function should return a new list sorted by scores in descending order. If two tuples have the same score, they should be sorted alphabetically by name in ascending order. Test your function with a predefined list of tuples and print the sorted list.

Sample Input: `[('Alice', 88), ('Bob', 95), ('Charlie', 88), ('Dave', 95)]`

Sample Output: `[('Bob', 95), ('Dave', 95), ('Alice', 88), ('Charlie', 88)]`

OUTPUT:



```
# Define the custom sorting function
def custom_sort(tuples_list):
    # Sort by score (descending), then by name (ascending) for ties
    return sorted(tuples_list, key=lambda x: (-x[1], x[0]))

# Sample input list of tuples (name, score)
sample_data = [('Alice', 88), ('Bob', 95), ('Charlie', 88), ('Dave', 95)]

# Apply the custom sorting function
sorted_list = custom_sort(sample_data)

# Output the sorted list
print("Sorted list:", sorted_list)
```

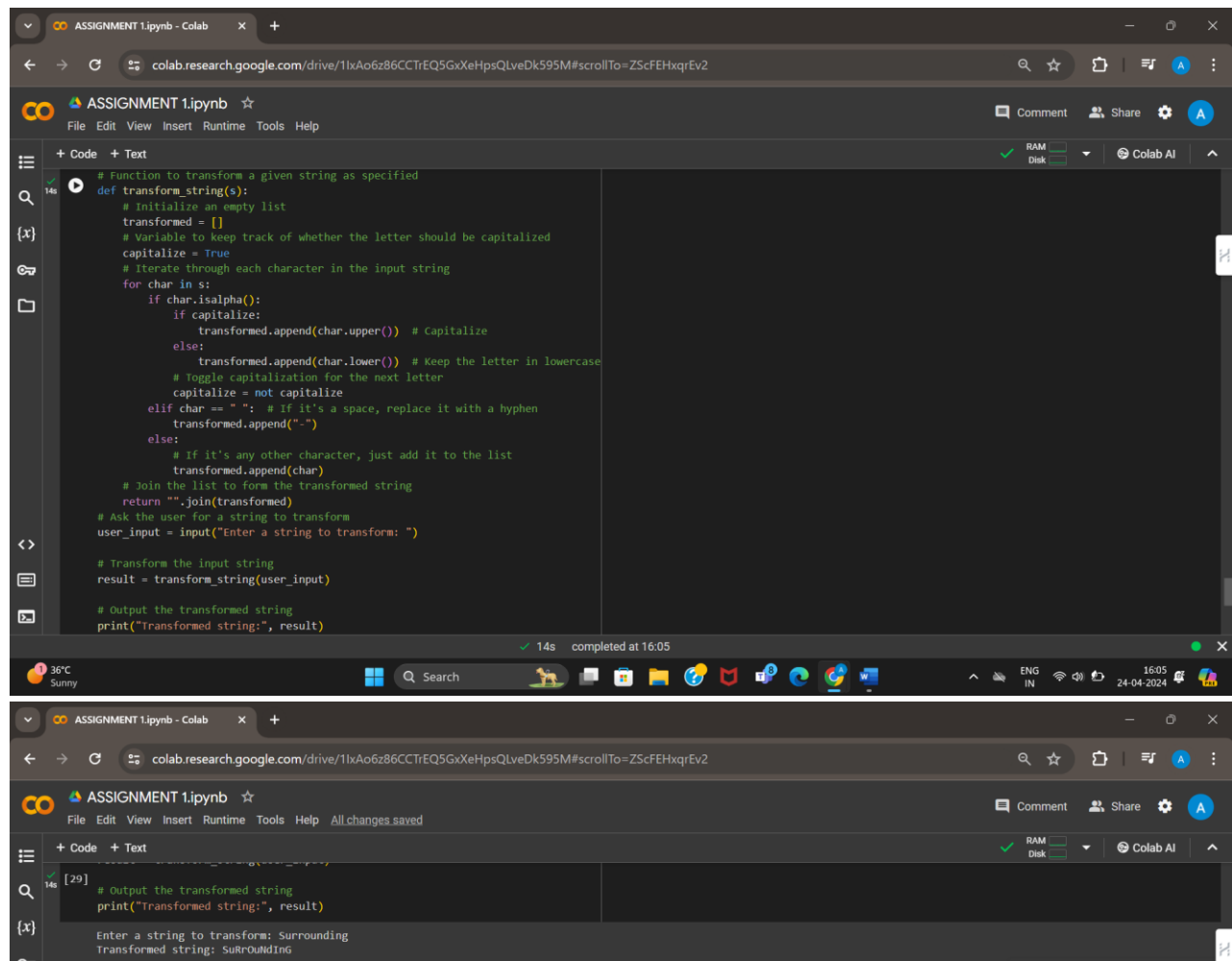
Sorted list: [('Bob', 95), ('Dave', 95), ('Alice', 88), ('Charlie', 88)]

13. Develop a Python function named `transform_string` that takes a string and performs the following transformations: it capitalizes every other letter starting with the first character (ignoring non-letter characters for the alternation pattern), and it replaces spaces with hyphens (-). For example, `hello world` becomes `HeLIO-WoRID`. After defining the function, ask the user for a string and print its transformation.

Sample Input: `hello world`

Sample Output: `HeLIO-WoRID`

OUTPUT:



The image displays two screenshots of a Google Colab notebook titled "ASSIGNMENT 1.ipynb".

The top screenshot shows the code for the `transform_string` function. The code is as follows:

```
# Function to transform a given string as specified
def transform_string(s):
    # Initialize an empty list
    transformed = []
    # Variable to keep track of whether the letter should be capitalized
    capitalize = True
    # Iterate through each character in the input string
    for char in s:
        if char.isalpha():
            if capitalize:
                transformed.append(char.upper()) # Capitalize
            else:
                transformed.append(char.lower()) # Keep the letter in lowercase
            # Toggle capitalization for the next letter
            capitalize = not capitalize
        elif char == " ":
            # If it's a space, replace it with a hyphen
            transformed.append("-")
        else:
            # If it's any other character, just add it to the list
            transformed.append(char)
    # Join the list to form the transformed string
    return "".join(transformed)
# Ask the user for a string to transform
user_input = input("Enter a string to transform: ")
# Transform the input string
result = transform_string(user_input)
# Output the transformed string
print("transformed string:", result)
```

The bottom screenshot shows the execution of the code. The input prompt "Enter a string to transform: " is displayed, and the user has entered "Surrounding". The output is "transformed string: SuRrounDing".

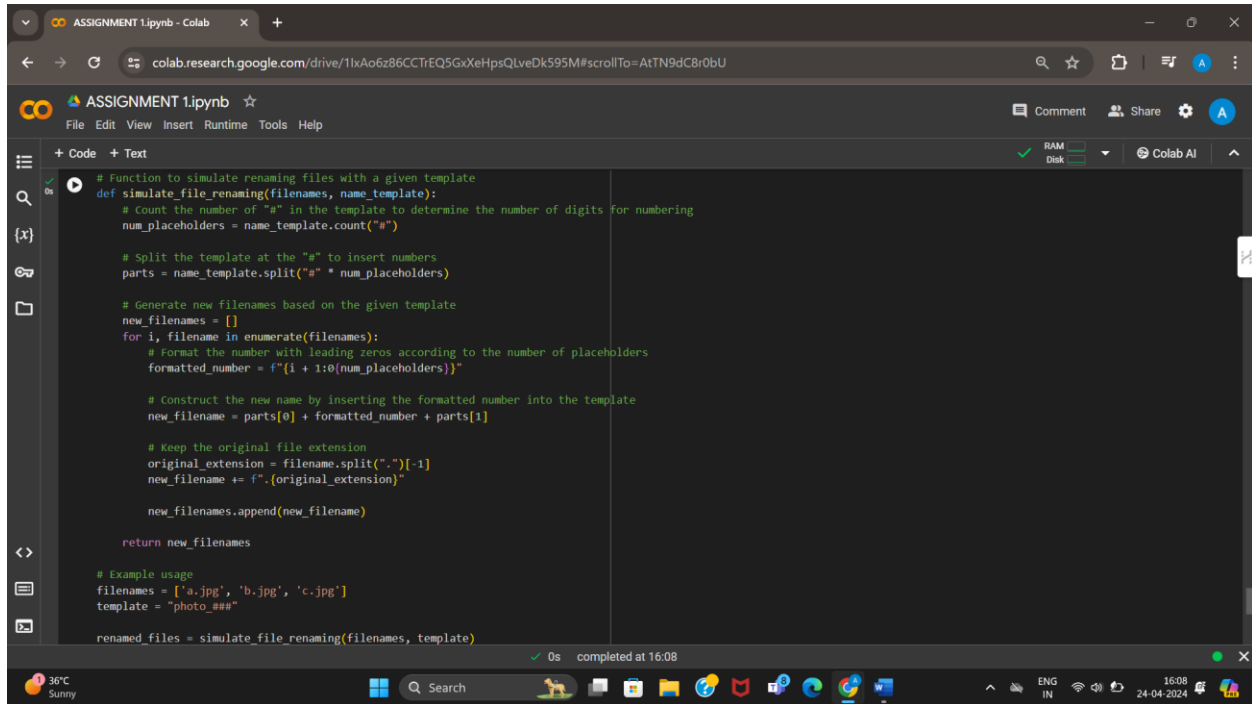
14. Create a function named `simulate_file_renaming` that takes two parameters: a list of filenames (as strings) and a new name template (a string containing a placeholder for a number, e.g., `image_##`). The function should return a list of strings representing the new filenames where the placeholder is replaced by an incremental number, starting from 1 and formatted to have leading zeros if necessary, according to the placeholder's length. For instance, renaming `['a.jpg', 'b.jpg', 'c.jpg']` with the template `photo_###` would

result in ['photo_001.jpg', 'photo_002.jpg', 'photo_003.jpg']. This exercise simulates the renaming process, so you should only return the renamed list without actually renaming any files.

Sample Input: ['a.jpg', 'b.jpg', 'c.jpg'], photo_###

Sample Output: ['photo_001.jpg', 'photo_002.jpg', 'photo_003.jpg']

OUTPUT:



The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code cell contains a Python function `simulate_file_renaming` that takes a list of filenames and a template string as input. The function counts the number of '#' characters in the template to determine the number of digits for numbering. It then splits the template at the '#' to insert the formatted numbers. The function also keeps the original file extension. Finally, it returns a list of the renamed filenames. The code is as follows:

```
# Function to simulate renaming files with a given template
def simulate_file_renaming(filenamees, name_template):
    # Count the number of "#" in the template to determine the number of digits for numbering
    num_placeholders = name_template.count("#")

    # Split the template at the "#" to insert numbers
    parts = name_template.split("#" * num_placeholders)

    # Generate new filenames based on the given template
    new_filenames = []
    for i, filename in enumerate(filenamees):
        # format the number with leading zeros according to the number of placeholders
        formatted_number = f"{i + 1:0{num_placeholders}d}"

        # Construct the new name by inserting the formatted number into the template
        new_filename = parts[0] + formatted_number + parts[1]

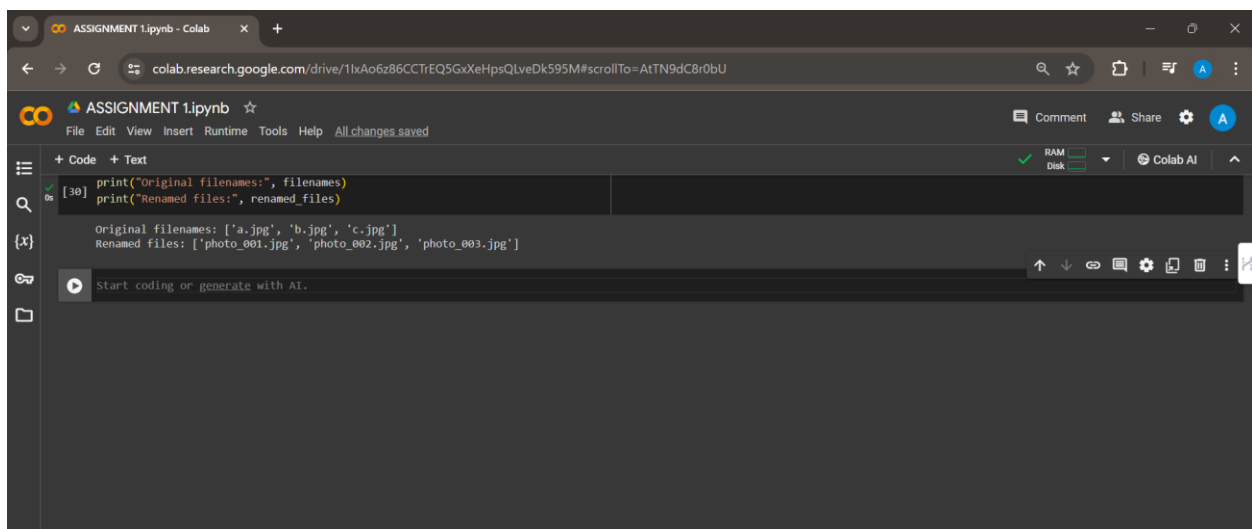
        # Keep the original file extension
        original_extension = filename.split(".")[1]
        new_filename += f".{original_extension}"

        new_filenames.append(new_filename)

    return new_filenames

# Example usage
filenamees = ['a.jpg', 'b.jpg', 'c.jpg']
template = "photo_###"

renamed_files = simulate_file_renaming(filenamees, template)
```



The screenshot shows the same Google Colab notebook, but now the code cell is executed. The output of the function is displayed in the output cell. The output is as follows:

```
Original filenames: ['a.jpg', 'b.jpg', 'c.jpg']
Renamed files: ['photo_001.jpg', 'photo_002.jpg', 'photo_003.jpg']
```

15. You are given a list of words. Write a Python function called `group_anagrams` that groups all anagrams together and returns them as a list of lists.

Two words are considered anagrams if they contain the same characters but in a different order.

Examples:

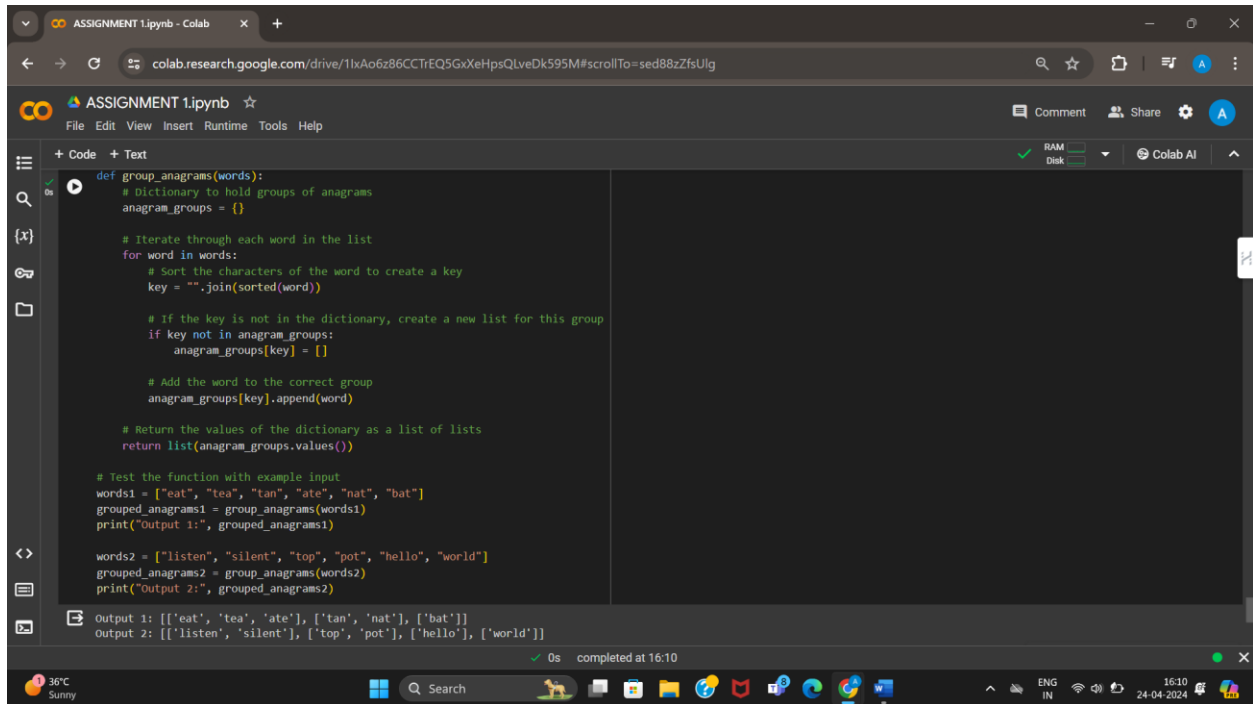
Input: ["eat", "tea", "tan", "ate", "nat", "bat"]

Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]

Input: ["listen", "silent", "top", "pot", "hello", "world"]

Output: [["listen", "silent"], ["top", "pot"], ["hello"], ["world"]]

OUTPUT:



The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code defines a function `group_anagrams(words)` that groups words into anagrams based on their sorted characters. It includes test cases for the examples provided in the text. The output of the notebook shows the results of the function calls.

```
def group_anagrams(words):  
    # Dictionary to hold groups of anagrams  
    anagram_groups = {}  
  
    # Iterate through each word in the list  
    for word in words:  
        # Sort the characters of the word to create a key  
        key = "".join(sorted(word))  
  
        # If the key is not in the dictionary, create a new list for this group  
        if key not in anagram_groups:  
            anagram_groups[key] = []  
  
        # Add the word to the correct group  
        anagram_groups[key].append(word)  
  
    # Return the values of the dictionary as a list of lists  
    return list(anagram_groups.values())  
  
# Test the function with example input  
words1 = ["eat", "tea", "tan", "ate", "nat", "bat"]  
grouped_anagrams1 = group_anagrams(words1)  
print("Output 1:", grouped_anagrams1)  
  
words2 = ["listen", "silent", "top", "pot", "hello", "world"]  
grouped_anagrams2 = group_anagrams(words2)  
print("Output 2:", grouped_anagrams2)
```

Output 1: [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
Output 2: [['listen', 'silent'], ['top', 'pot'], ['hello'], ['world']]

16. You are given a list of integers. Write a Python function called `max_subarray_sum` to find the contiguous subarray within the list that has the largest sum and return that sum.

For example, given the list `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the contiguous subarray with the largest sum is `[4, -1, 2, 1]`, and the maximum sum is 6.

Examples:

Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Output: 6 (corresponding to the subarray [4, -1, 2, 1])

Input: [1, 2, 3, 4, 5]

Output: 15 (corresponding to the subarray [1, 2, 3, 4, 5])

OUTPUT:

The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code defines a function `max_subarray_sum(arr)` that implements Kadane's algorithm. It initializes `max_sum` to `-inf` and `current_sum` to 0. It then iterates through each element in the array, updating `current_sum` and `max_sum` as it goes. After the loop, it returns `max_sum`. Test cases are provided for two arrays: `[-2, 1, -3, 4, -1, 2, 1, -5, 4]` and `[1, 2, 3, 4, 5]`. The output shows the expected results for these test cases.

```
# Function to find the maximum subarray sum using Kadane's algorithm
def max_subarray_sum(arr):
    # Initialize variables to keep track of the maximum sum and current sum
    max_sum = float('-inf') # Start with the smallest possible value
    current_sum = 0

    # Loop through each element in the list
    for num in arr:
        # Add the current number to the current sum
        current_sum += num

        # Update the max sum if the current sum is greater
        if current_sum > max_sum:
            max_sum = current_sum

    # If the current sum becomes negative, reset it to zero
    if current_sum < 0:
        current_sum = 0

    return max_sum

# Test cases
test1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
result1 = max_subarray_sum(test1)
print("Output 1:", result1) # Expected: 6 (subarray [4, -1, 2, 1])

test2 = [1, 2, 3, 4, 5]
result2 = max_subarray_sum(test2)
print("Output 2:", result2) # Expected: 15 (subarray [1, 2, 3, 4, 5])
```

Output 1: 6
Output 2: 15

17. Implement a function that performs a sequential search through a list for a specified target value. The function should return the index of the target if found, and -1 if the target is not in the list.

Sample Input: ([5, 3, 7, 1, 9], 7)

Sample Output: 2

OUTPUT:

The screenshot shows a Google Colab notebook titled "ASSIGNMENT 1.ipynb". The code defines a function `sequential_search(list, target)` that iterates over the list with indices and elements. If the current item matches the target, it returns the index. If the loop completes and the target is not found, it returns -1. An example usage is provided for the list `[5, 3, 7, 1, 9]` and target `7`. The output shows the index of the target.

```
# Function to perform a sequential search
def sequential_search(list, target):
    # Iterate over the list with indices and elements
    for index, item in enumerate(list):
        # If the current item matches the target, return the index
        if item == target:
            return index

    # If the loop completes and the target is not found, return -1
    return -1

# Example usage
sample_list = [5, 3, 7, 1, 9]
target = 7
# Perform the sequential search
result = sequential_search(sample_list, target)
# Output the result
print(f"Index of target {target}: ", result)
```

Index of target 7: 2

18. Design a method to encode a list of strings to a single string and another method to decode it back to a list of strings.

The encoded string should be concise and easily decodable. Assume there are no character restrictions for individual strings.

Examples:

19. Input: ["hello", "world"]

Encoded Output: "5#hello5#world" (or another unique format of your choice)

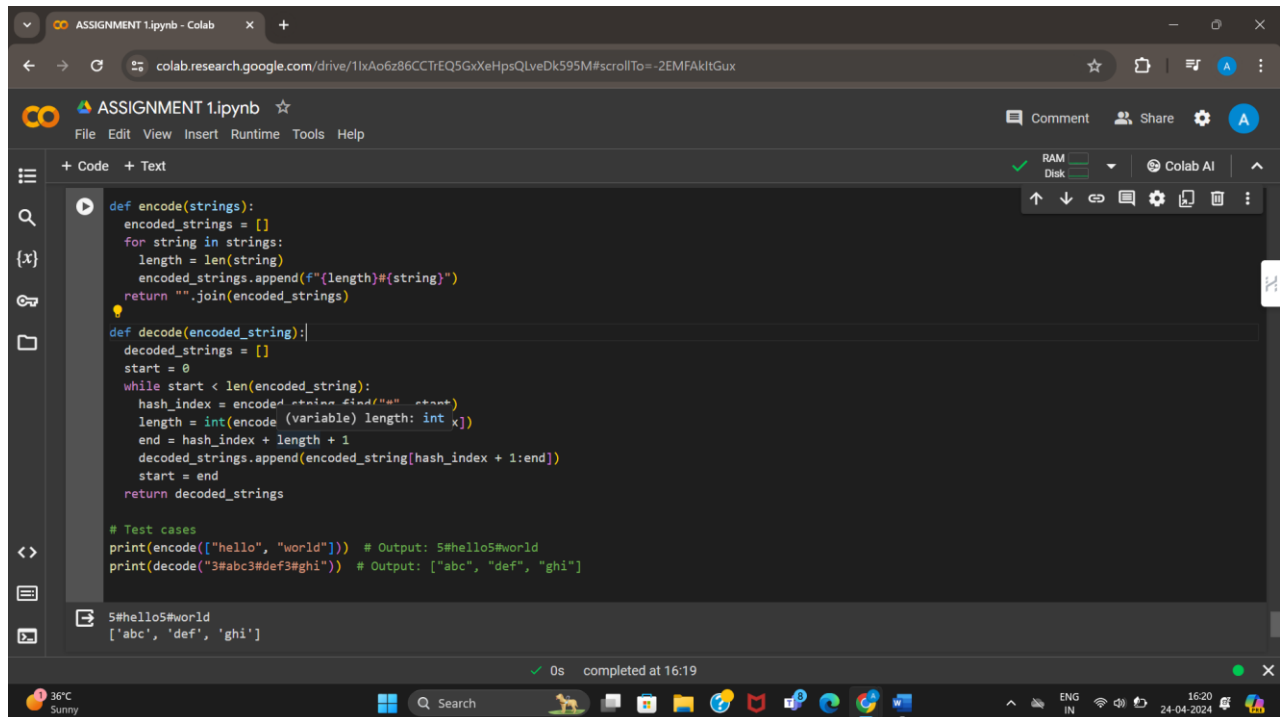
Decoded Output: ["hello", "world"]

20. Input: ["abc", "def", "ghi"]

Encoded Output: "3#abc3#def3#ghi"

Decoded Output: ["abc", "def", "ghi"]

OUTPUT:



```
def encode(strings):
    encoded_strings = []
    for string in strings:
        length = len(string)
        encoded_strings.append(f"{length}#{string}")
    return "".join(encoded_strings)

def decode(encoded_string):
    decoded_strings = []
    start = 0
    while start < len(encoded_string):
        hash_index = encoded_string.find("#")
        length = int(encoded_string[hash_index:hash_index+1])
        end = hash_index + length + 1
        decoded_strings.append(encoded_string[hash_index + 1:end])
        start = end
    return decoded_strings

# Test cases
print(encode(["hello", "world"])) # Output: 5#hello5#world
print(decode("3#abc3#def3#ghi")) # Output: ["abc", "def", "ghi"]
```

5#hello5#world
['abc', 'def', 'ghi']

0s completed at 16:19