**Practical:1**

**Aim:** **Implement Caesar cipher encryption-decryption**

**Description:**

Implementing the Caesar cipher encryption-decryption involves a straightforward approach to shifting letters in the alphabet by a fixed number of positions. Here's a detailed description:

### **Encryption Process:**

1. **Input**:
   * Plain text message.
   * Shift value (key), which determines how many positions each letter should be shifted in the alphabet.
2. **Steps**:
   * Convert all letters in the plain text to uppercase to standardise handling (optional but recommended).
   * Iterate through each character in the plain text:
     + If the character is a letter (A-Z), compute its new position by adding the shift value.
     + Ensure to wrap around the alphabet if shifting goes beyond 'Z' (e.g., shifting 'Z' by 1 wraps around to 'A').
     + Leave non-alphabetic characters unchanged (spaces, punctuation, etc.).
   * Construct the encrypted message using the shifted characters.
3. **Output**:
   * Encrypted message where each letter has been substituted with a shifted letter based on the provided key.

### **Decryption Process:**

1. **Input**:
   * Encrypted message.
   * Shift value used for encryption.
2. **Steps**:
   * Convert all letters in the encrypted message to uppercase.
   * Iterate through each character in the encrypted message:
     + If the character is a letter (A-Z), compute its original position by subtracting the shift value.
     + Ensure to wrap around the alphabet if shifting goes before 'A' (e.g., shifting 'A' by 1 wraps around to 'Z').
     + Leave non-alphabetic characters unchanged.
   * Reconstruct the decrypted message using the original characters.
3. **Output**:
   * Decrypted message where each letter has been shifted back to its original position based on the provided key.

**Code:**

**def caesar\_cipher(text, shift, decrypt=False):**

**shifted\_text = ""**

**for char in text:**

**if char.isalpha(): # Check if the character is a letter**

**if decrypt:**

**shifted = ord(char) - shift**

**else:**

**shifted = ord(char) + shift**

**if char.islower():**

**if decrypt and shifted < ord('a'):**

**shifted += 26**

**elif not decrypt and shifted > ord('z'):**

**shifted -= 26**

**elif char.isupper():**

**if decrypt and shifted < ord('A'):**

**shifted += 26**

**elif not decrypt and shifted > ord('Z'):**

**shifted -= 26**

**shifted\_text += chr(shifted)**

**else:**

**shifted\_text += char # Non-alphabetic characters remain unchanged**

**return shifted\_text**

**# Get user input for text, mode (encryption or decryption), and shift amount**

**plain\_text = input("Enter text: ")**

**while True:**

**mode = input("Enter 'e' for encryption or 'd' for decryption: ").lower()**

**if mode == 'e' or mode == 'd':**

**break**

**else:**

**print("Invalid input. Please enter 'e' for encryption or 'd' for decryption.")**

**while True:**

**try:**

**shift\_amount = int(input("Enter the shift amount (integer): "))**

**break**

**except ValueError:**

**print("Please enter a valid integer.")**

**# Perform encryption or decryption based on user input**

**if mode == 'e':**

**encrypted\_text = caesar\_cipher(plain\_text, shift\_amount)**

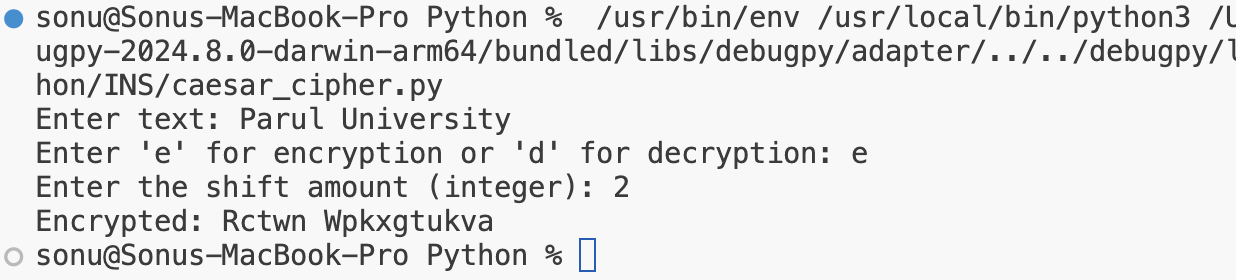
**print("Encrypted:", encrypted\_text)**

**elif mode == 'd':**

**decrypted\_text = caesar\_cipher(plain\_text, shift\_amount, decrypt=True)**

**print("Decrypted:", decrypted\_text)**

**Output:**



**Practical:2**

**Aim:**  **Implement Monoalphabetic cipher encryption-decryption**

**Description:**

Implementing a Monoalphabetic cipher encryption-decryption involves substituting each letter in the plaintext with another letter from a fixed mapping. Here's a detailed description:

### Encryption Process:

1. Input:
   * Plain text message.
   * Monoalphabetic substitution mapping (key), which specifies the substitution of each letter in the alphabet with another letter.
2. Steps:
   * Create a mapping between each letter in the alphabet (A-Z) and its corresponding substitution according to the key.
   * Convert all letters in the plain text to uppercase for standardisation (optional but simplifies handling).
   * Iterate through each character in the plain text:
     + Substitute each letter with its corresponding mapped letter according to the key.
     + Non-alphabetic characters (spaces, punctuation, etc.) remain unchanged.
   * Construct the encrypted message using the substituted characters.
3. Output:
   * Encrypted message where each letter has been substituted with another letter based on the monoalphabetic substitution key.

### Decryption Process:

1. Input:
   * Encrypted message.
   * Monoalphabetic substitution mapping (same key used for encryption).
2. Steps:
   * Convert all letters in the encrypted message to uppercase.
   * Reverse the substitution process:
     + For each character in the encrypted message, find its corresponding original letter using the decryption key.
     + Non-alphabetic characters remain unchanged.
   * Reconstruct the decrypted message using the original characters.
3. Output:
   * Decrypted message where each letter has been substituted back to its original form based on the monoalphabetic substitution key.

**Code:**

**normal\_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',**

**'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',**

**'s', 't', 'u', 'v', 'w', 'x', 'y', 'z']**

**coded\_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',**

**'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',**

**'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']**

**def string\_encryption(s):**

**encrypted\_string = ""**

**for char in s:**

**if char.lower() in normal\_char:**

**encrypted\_string += coded\_char[normal\_char.index(char.lower())]**

**else:**

**encrypted\_string += char**

**return encrypted\_string**

**def string\_decryption(s):**

**decrypted\_string = ""**

**for char in s:**

**if char in coded\_char:**

**decrypted\_string += normal\_char[coded\_char.index(char)]**

**else:**

**decrypted\_string += char**

**return decrypted\_string**

**if \_\_name\_\_ == "\_\_main\_\_":**

**plain\_text = input("Enter the plain text to encrypt: ")**

**print("Plain text:", plain\_text)**

**encrypted\_message = string\_encryption(plain\_text.lower())**

**print("Encrypted message:", encrypted\_message)**

**decrypted\_message = string\_decryption(encrypted\_message)**

**print("Decrypted message:", decrypted\_message)**

**normal\_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',**

**'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',**

**'s', 't', 'u', 'v', 'w', 'x', 'y', 'z']**

**coded\_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',**

**'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',**

**'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']**

**def string\_encryption(s):**

**encrypted\_string = ""**

**for char in s:**

**if char.lower() in normal\_char:**

**encrypted\_string += coded\_char[normal\_char.index(char.lower())]**

**else:**

**encrypted\_string += char**

**return encrypted\_string**

**def string\_decryption(s):**

**decrypted\_string = ""**

**for char in s:**

**if char in coded\_char:**

**decrypted\_string += normal\_char[coded\_char.index(char)]**

**else:**

**decrypted\_string += char**

**return decrypted\_string**

**if \_\_name\_\_ == "\_\_main\_\_":**

**plain\_text = input("Enter the plain text to encrypt: ")**

**print("Plain text:", plain\_text)**

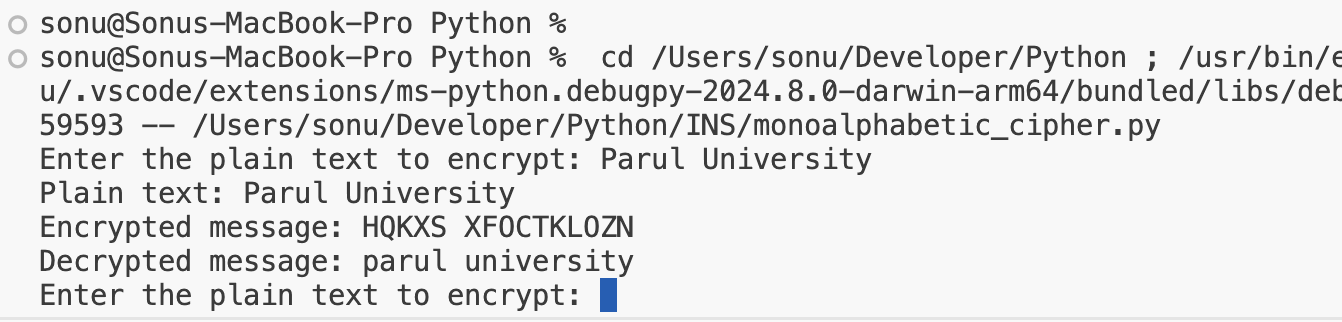
**encrypted\_message = string\_encryption(plain\_text.lower())**

**print("Encrypted message:", encrypted\_message)**

**decrypted\_message = string\_decryption(encrypted\_message)**

**print("Decrypted message:", decrypted\_message)**

**Output:**



**Practical:3**

**Aim:**  **Implement Playfair cipher encryption-decryption**

**Description:**

Implementing the Playfair cipher for encryption-decryption involves a more complex approach compared to simpler ciphers like Caesar or Monoalphabetic. Here's a detailed description:

### **Playfair Cipher Overview:**

The Playfair cipher operates on pairs of letters (digraphs), rather than single letters as in many other ciphers. It uses a 5x5 grid of letters (usually constructed from a keyword) to facilitate encryption and decryption.

### **Encryption Process:**

1. **Key Setup**:
   * Generate a 5x5 matrix (Playfair square) using a keyword. The keyword is used to fill the matrix without repeating any letters, and typically 'I' and 'J' are treated as the same letter.
2. **Message Preparation**:
   * Convert the plaintext into digraphs (pairs of letters). If there's an odd number of letters, add a filler (often 'X') at the end.
3. **Rules for Digraph Formation**:
   * If both letters of a digraph are the same, insert a filler (often 'X') between them.
   * If a digraph consists of two identical letters separated by a space or punctuation, treat them as two separate letters.
4. **Encryption Steps**:
   * For each digraph:
     + Determine their positions in the Playfair square.
     + Apply specific rules to determine the ciphertext digraph:
       - If the letters are in the same row, replace each letter with the letter to its immediate right (wrap around if needed).
       - If the letters are in the same column, replace each letter with the letter immediately below it (wrap around if needed).
       - If the letters form a rectangle, replace them with the letters on the same row but at the opposite corners of the rectangle.
       - If the letters are different and not in the same row or column, form a rectangle and replace each letter with the letter on the same row but in the column of the other letter of the rectangle.
5. **Output**:
   * The encrypted message consists of the ciphertext digraphs.

### **Decryption Process:**

1. **Key Setup**:
   * Generate the same 5x5 matrix (Playfair square) using the same keyword.
2. **Message Preparation**:
   * Convert the ciphertext into digraphs.
3. **Decryption Steps**:
   * For each digraph:
     + Determine their positions in the Playfair square.
     + Apply the reverse rules used in encryption to determine the plaintext digraph.
4. **Output**:
   * The decrypted message consists of the plaintext digraphs.

**Code:**

**def generate\_playfair\_key\_matrix(key):**

**alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" # J is combined with I**

**key = key.upper().replace("J", "I")**

**# Remove duplicates while preserving order**

**key\_no\_duplicates = ""**

**for char in key:**

**if char not in key\_no\_duplicates:**

**key\_no\_duplicates += char**

**# Add remaining letters of the alphabet**

**key\_no\_duplicates += ''.join([c for c in alphabet if c not in key\_no\_duplicates])**

**matrix = [list(key\_no\_duplicates[i:i + 5]) for i in range(0, 25, 5)]**

**return matrix**

**def preprocess\_text(text):**

**text = text.upper().replace('J', 'I')**

**text = ''.join([c for c in text if c.isalpha()])**

**pairs = []**

**i = 0**

**while i < len(text):**

**a = text[i]**

**if i + 1 < len(text):**

**b = text[i + 1]**

**if a != b:**

**pairs.append(a + b)**

**i += 2**

**else:**

**pairs.append(a + 'X')**

**i += 1**

**else:**

**pairs.append(a + 'X')**

**i += 1**

**return pairs**

**def find\_position(matrix, char):**

**for row in range(5):**

**for col in range(5):**

**if matrix[row][col] == char:**

**return row, col**

**def playfair\_encrypt\_decrypt(text, key, mode='encrypt'):**

**matrix = generate\_playfair\_key\_matrix(key)**

**text\_pairs = preprocess\_text(text)**

**result = ''**

**for a, b in text\_pairs:**

**row\_a, col\_a = find\_position(matrix, a)**

**row\_b, col\_b = find\_position(matrix, b)**

**if row\_a == row\_b:**

**if mode == 'encrypt':**

**result += matrix[row\_a][(col\_a + 1) % 5] + matrix[row\_b][(col\_b + 1) % 5]**

**else:**

**result += matrix[row\_a][(col\_a - 1) % 5] + matrix[row\_b][(col\_b - 1) % 5]**

**elif col\_a == col\_b:**

**if mode == 'encrypt':**

**result += matrix[(row\_a + 1) % 5][col\_a] + matrix[(row\_b + 1) % 5][col\_b]**

**else:**

**result += matrix[(row\_a - 1) % 5][col\_a] + matrix[(row\_b - 1) % 5][col\_b]**

**else:**

**result += matrix[row\_a][col\_b] + matrix[row\_b][col\_a]**

**return result**

**# Main program**

**def main():**

**message = input("Enter the message: ")**

**key = input("Enter the key: ")**

**encrypted\_message = playfair\_encrypt\_decrypt(message, key, mode='encrypt')**

**print(f"Encrypted Message: {encrypted\_message}")**

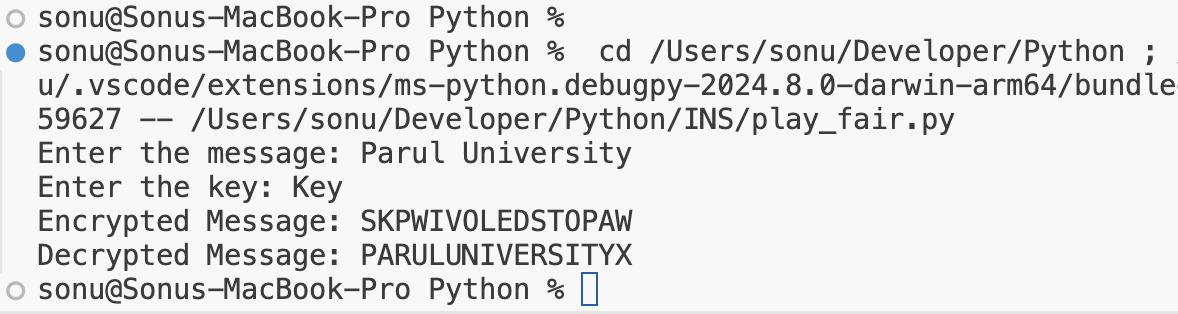
**decrypted\_message = playfair\_encrypt\_decrypt(encrypted\_message, key, mode='decrypt')**

**print(f"Decrypted Message: {decrypted\_message}")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Output:**

****

**Practical:4**

**Aim:**  **Implement Polyalphabetic cipher encryption-decryption**

**Description:**

Implementing a Polyalphabetic cipher for encryption-decryption involves using multiple substitution alphabets, typically represented by a keyword or phrase. Here’s a detailed description:

### Polyalphabetic Cipher Overview:

Polyalphabetic ciphers use multiple substitution alphabets to enhance security compared to monoalphabetic ciphers. The most famous example is the Vigenère cipher.

### Encryption Process:

1. Key Setup:
   * Choose a keyword or phrase that determines the sequence of substitution alphabets.
   * Convert the keyword to a numeric key (A=0, B=1, ..., Z=25).
2. Message Preparation:
   * Convert the plaintext to uppercase for standardization (optional but simplifies handling).
   * Align the plaintext with the keyword by repeating the keyword as necessary to match the length of the plaintext.
   * Convert each letter of the plaintext and the corresponding letter of the keyword into their numeric equivalents (A=0, B=1, ..., Z=25).
3. Encryption Steps:
   * For each letter in the plaintext:
     + Determine the shift amount by the numeric value of the corresponding letter in the keyword.
     + Shift the letter in the plaintext forward by the calculated shift amount (modulo 26 to wrap around the alphabet).
     + Construct the ciphertext by replacing each letter with the shifted letter.
4. Output:
   * The encrypted message (ciphertext) consists of the letters shifted according to the keyword-derived alphabets.

### Decryption Process:

1. Key Setup:
   * Use the same keyword and convert it to numeric values (A=0, B=1, ..., Z=25).
2. Message Preparation:
   * Convert the ciphertext to uppercase.
   * Align the ciphertext with the keyword by repeating the keyword as necessary.
3. Decryption Steps:
   * For each letter in the ciphertext:
     + Determine the shift amount by the numeric value of the corresponding letter in the keyword.
     + Shift the letter in the ciphertext backward by the calculated shift amount (modulo 26 to handle wrapping).
     + Construct the plaintext by replacing each letter with the shifted letter.
4. Output:
   * The decrypted message (plaintext) consists of the letters shifted back according to the keyword-derived alphabets.

**Code:**

**import string**

**def clean\_text(text):**

**""" Removes non-alphabetic characters and converts to uppercase """**

**cleaned\_text = []**

**for char in text.upper():**

**if char in string.ascii\_uppercase:**

**cleaned\_text.append(char)**

**return ''.join(cleaned\_text)**

**def encrypt\_vigenere(plaintext, keyword):**

**plaintext = clean\_text(plaintext)**

**keyword = clean\_text(keyword)**

**keyword\_length = len(keyword)**

**ciphertext = []**

**for i in range(len(plaintext)):**

**shift = ord(keyword[i % keyword\_length]) - ord('A')**

**encrypted\_char = chr((ord(plaintext[i]) - ord('A') + shift) % 26 + ord('A'))**

**ciphertext.append(encrypted\_char)**

**return ''.join(ciphertext)**

**def decrypt\_vigenere(ciphertext, keyword):**

**ciphertext = clean\_text(ciphertext)**

**keyword = clean\_text(keyword)**

**keyword\_length = len(keyword)**

**plaintext = []**

**for i in range(len(ciphertext)):**

**shift = ord(keyword[i % keyword\_length]) - ord('A')**

**decrypted\_char = chr((ord(ciphertext[i]) - ord('A') - shift + 26) % 26 + ord('A'))**

**plaintext.append(decrypted\_char)**

**return ''.join(plaintext)**

**# Get user input**

**plaintext\_or\_ciphertext = input("Enter plaintext or ciphertext (letters only): ").strip()**

**keyword = input("Enter keyword (letters only): ").strip()**

**# Encrypt and decrypt based on user input**

**encrypted\_text = encrypt\_vigenere(plaintext\_or\_ciphertext, keyword)**

**decrypted\_text = decrypt\_vigenere(encrypted\_text, keyword)**

**print("Plaintext/Ciphertext provided:", plaintext\_or\_ciphertext)**

**print("Keyword provided:", keyword)**

**print("Encrypted:", encrypted\_text)**

**print("Decrypted:", decrypted\_text)**

**Output:**

****

**Practical:5**

**Aim:**  **Implement Hill cipher encryption-decryption**

**Description:**

Implementing the Hill cipher for encryption-decryption involves matrix operations on blocks of plaintext letters, which provides a higher level of security compared to simpler ciphers. Here’s a detailed description:

### Hill Cipher Overview:

The Hill cipher is a polygraphic substitution cipher that uses linear algebra to encrypt and decrypt messages. It operates on blocks of plaintext letters (typically 2x2 or 3x3 matrices) and their corresponding ciphertext.

### Encryption Process:

1. Key Setup:
   * Choose a square matrix (encryption key matrix) of size 2x2 or 3x3, where each element is a number modulo 26 (typically representing letters A=0, B=1, ..., Z=25).
   * Ensure the key matrix is invertible (its determinant must be non-zero modulo 26).
2. Message Preparation:
   * Convert the plaintext to uppercase for standardisation.
   * Break the plaintext into blocks of letters that match the size of the key matrix (e.g., pairs of letters for a 2x2 matrix).
3. Encryption Steps:
   * Represent each block of plaintext letters as a vector (column matrix) of numeric equivalents.
   * Multiply each vector by the encryption key matrix (modulo 26).
   * Convert the resulting numbers back to letters to form the ciphertext.
4. Output:
   * The encrypted message (ciphertext) consists of the blocks of letters transformed by the key matrix.

### Decryption Process:

1. Key Setup:
   * Use the inverse of the encryption key matrix. The inverse matrix must also be modulo 26.
2. Message Preparation:
   * Convert the ciphertext to uppercase.
   * Break the ciphertext into blocks matching the size of the key matrix.
3. Decryption Steps:
   * Represent each block of ciphertext letters as a vector (column matrix) of numeric equivalents.
   * Multiply each vector by the inverse of the encryption key matrix (modulo 26).
   * Convert the resulting numbers back to letters to reconstruct the plaintext.
4. Output:
   * The decrypted message (plaintext) consists of the blocks of letters transformed back by the inverse key matrix.

**Code:**

**# Function to convert a string to a list of numbers (A=0, B=1, ..., Z=25)**

**def string\_to\_numbers(text):**

**return [ord(char) - ord('A') for char in text.upper() if 'A' <= char <= 'Z']**

**# Function to convert a list of numbers back to a string**

**def numbers\_to\_string(numbers):**

**return ''.join([chr(num + ord('A')) for num in numbers])**

**# Function to convert a key matrix from letters or numbers to a numeric format**

**def key\_to\_numeric(key):**

**if isinstance(key[0][0], str): # Check if the first element is a string (assumes entire matrix is the same type)**

**return [[ord(char.upper()) - ord('A') for char in row] for row in key]**

**else:**

**return key**

**# Function to compute the determinant of a 2x2 matrix**

**def determinant\_2x2(matrix):**

**return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]**

**# Function to compute the modular inverse of a 2x2 matrix modulo 26**

**def mod\_inverse\_2x2(matrix, modulus):**

**det = determinant\_2x2(matrix)**

**det\_inv = pow(det, -1, modulus) # Calculate modular inverse of determinant**

**adjugate = [[matrix[1][1], -matrix[0][1]], [-matrix[1][0], matrix[0][0]]]**

**adjugate\_mod = [[adj % modulus for adj in row] for row in adjugate]**

**return [[(det\_inv \* adj) % modulus for adj in row] for row in adjugate\_mod]**

**# Function to multiply a 2x2 matrix and a 2-element vector modulo 26**

**def matrix\_vector\_multiply(matrix, vector, modulus):**

**return [(matrix[0][0] \* vector[0] + matrix[0][1] \* vector[1]) % modulus,**

**(matrix[1][0] \* vector[0] + matrix[1][1] \* vector[1]) % modulus]**

**# Function to pad the plaintext to make its length a multiple of n (for encryption)**

**def pad\_text(text, n):**

**padded\_text = text**

**while len(padded\_text) % n != 0:**

**padded\_text += 'X'**

**return padded\_text**

**# Function to encrypt plaintext using Hill cipher with a user-provided key matrix**

**def hill\_cipher\_encrypt(plaintext, key):**

**n = 2 # Size of the key matrix is fixed to 2x2 for Hill cipher**

**plaintext = pad\_text(plaintext, n)**

**plaintext\_numbers = string\_to\_numbers(plaintext)**

**key\_numeric = key\_to\_numeric(key)**

**ciphertext\_numbers = []**

**for i in range(0, len(plaintext\_numbers), n):**

**chunk = plaintext\_numbers[i:i+n]**

**encrypted\_chunk = matrix\_vector\_multiply(key\_numeric, chunk, 26)**

**ciphertext\_numbers.extend(encrypted\_chunk)**

**return numbers\_to\_string(ciphertext\_numbers)**

**# Function to decrypt ciphertext using Hill cipher with a user-provided key matrix**

**def hill\_cipher\_decrypt(ciphertext, key):**

**n = 2 # Size of the key matrix is fixed to 2x2 for Hill cipher**

**key\_numeric = key\_to\_numeric(key)**

**key\_inverse = mod\_inverse\_2x2(key\_numeric, 26)**

**ciphertext\_numbers = string\_to\_numbers(ciphertext)**

**plaintext\_numbers = []**

**for i in range(0, len(ciphertext\_numbers), n):**

**chunk = ciphertext\_numbers[i:i+n]**

**decrypted\_chunk = matrix\_vector\_multiply(key\_inverse, chunk, 26)**

**plaintext\_numbers.extend(decrypted\_chunk)**

**return numbers\_to\_string(plaintext\_numbers)**

**# Main function**

**def main():**

**# Input the plaintext and key from user**

**plaintext = input("Enter the plaintext: ").replace(" ", "").upper()**

**print("Enter the key matrix (2x2 matrix, each row on a new line, letters or numbers separated by space):")**

**key = []**

**for i in range(2):**

**row = input().strip().split()**

**try:**

**row = [int(num) for num in row] # Try converting to integers**

**except ValueError:**

**pass # If conversion fails, keep as strings**

**key.append(row)**

**print("Key Matrix:", key)**

**# Encrypt the plaintext**

**ciphertext = hill\_cipher\_encrypt(plaintext, key)**

**print("Encrypted:", ciphertext)**

**# Decrypt the ciphertext**

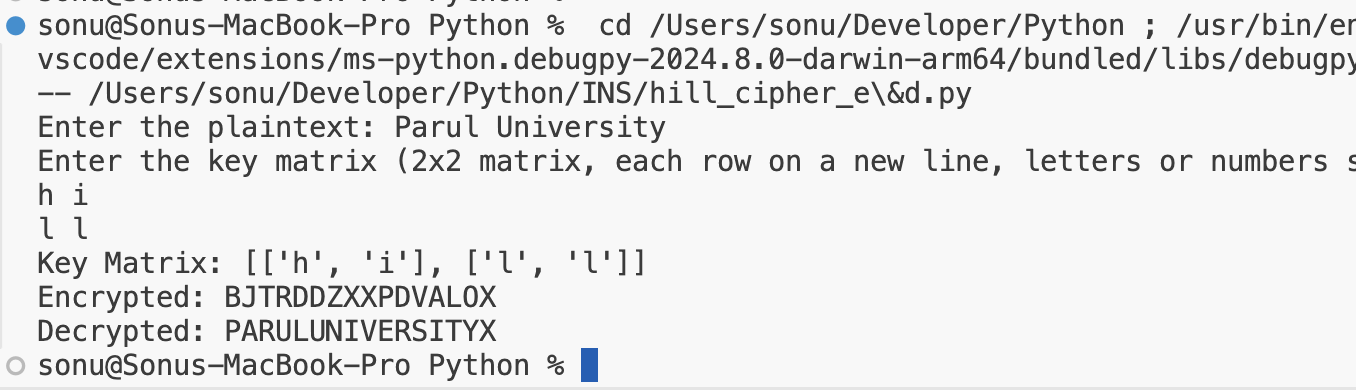
**decrypted\_text = hill\_cipher\_decrypt(ciphertext, key)**

**print("Decrypted:", decrypted\_text)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Output:**

****