

ONLINE LAB TASK

SUBMITTED BY:



SUBMITTED TO: MA’AM AMBREEN GUL

DATE: 14TH OCTOBER, 2025

COURSE: Information Security

***Transposition Cipher Tasks***

Q1. **Handle Different Key Sizes**

Modify the encode function to handle cases where the length of the key is not equal to the length of the plaintext. **Task**: Add padding to the plaintext when it is shorter than the key.

import random

import string

    # Task 1: Handle different key sizes with padding

def split\_len(seq, length):

    """Split sequence into chunks of specified length"""

    return [seq[i:i + length] for i in range(0, len(seq), length)]

def encode(key, plaintext):

    """

    Encrypt plaintext using columnar transposition cipher with padding

    """

    order = {int(val): num for num, val in enumerate(key)}

    # Convert to lowercase for consistent processing

    plaintext\_lower = plaintext.lower()

    key\_length = len(key)

    if len(plaintext\_lower) % key\_length != 0:

        # Add padding with random letters

        padding\_length = key\_length - (len(plaintext\_lower) % key\_length)

        padding = ''.join(random.choices(string.ascii\_lowercase, k=padding\_length))

        plaintext\_lower += padding

        print(f"Added padding: {padding}")

    ciphertext = ''

    for index in sorted(order.keys()):

        for part in split\_len(plaintext\_lower, len(key)):

            try:

                ciphertext += part[order[index]]

            except IndexError:

                pass

    return ciphertext

print("=== Task 1: Handle Different Key Sizes ===")

ciphertext = encode('321', 'HELLO')

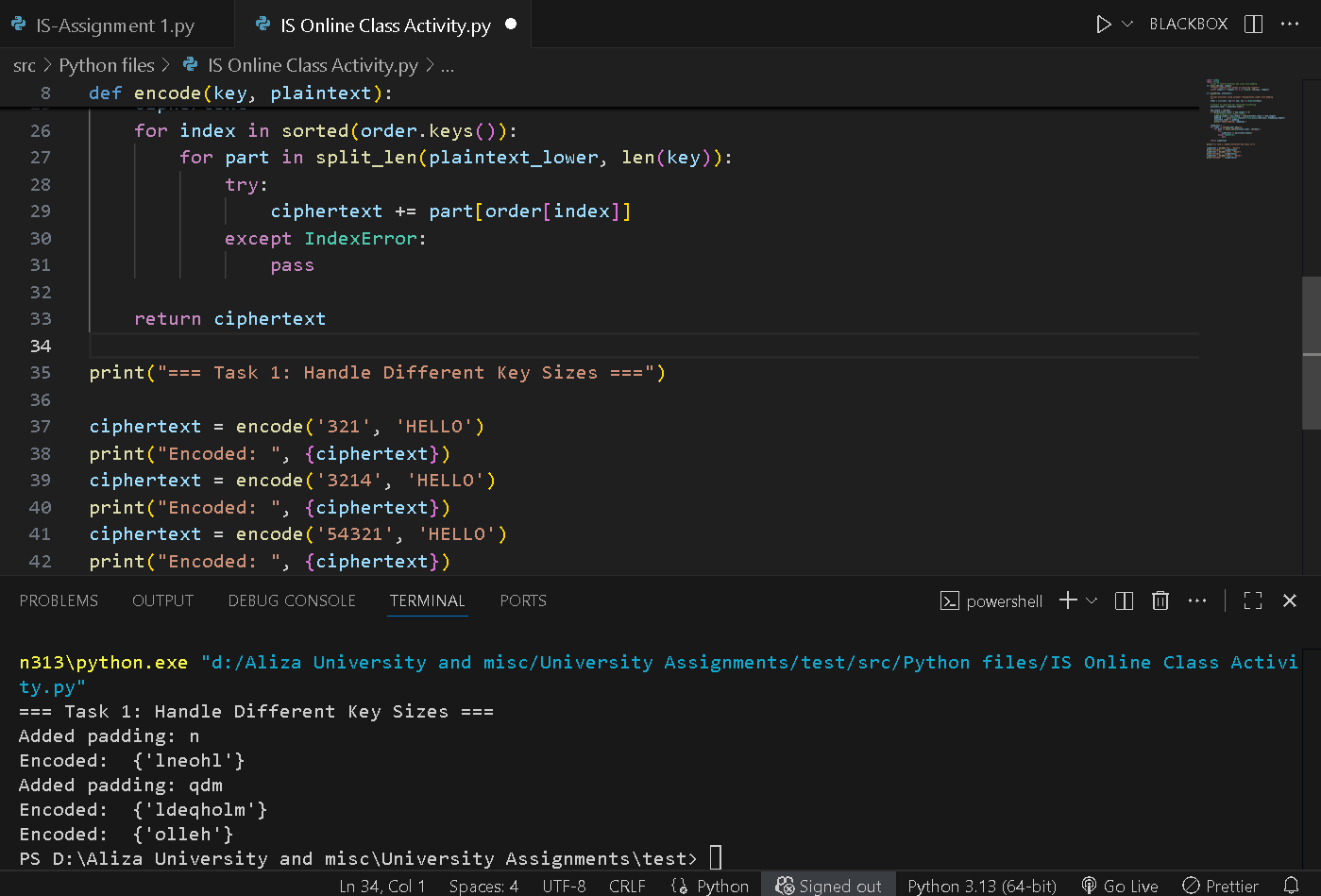
print("Encoded: ", {ciphertext})

ciphertext = encode('3214', 'HELLO')

print("Encoded: ", {ciphertext})

ciphertext = encode('54321', 'HELLO')

print("Encoded: ", {ciphertext})



Q2. **Decode Function**

Create a decode function that reverses the encode process. **Task**: Write a function decode(key, ciphertext) that deciphers the encrypted message and returns the original plaintext.

def split\_len(seq, length):

    """Split sequence into chunks of specified length"""

    return [seq[i:i + length] for i in range(0, len(seq), length)]

def encode(key, plaintext):

    """Original encode function"""

    order = {int(val): num for num, val in enumerate(key)}

    ciphertext = ''

    # Handle padding by ensuring plaintext length is multiple of key length

    key\_length = len(key)

    if len(plaintext) % key\_length != 0:

        # Add spaces for padding to make it work with original logic

        padding\_length = key\_length - (len(plaintext) % key\_length)

        plaintext += ' ' \* padding\_length

    for index in sorted(order.keys()):

        for part in split\_len(plaintext, len(key)):

            try:

                ciphertext += part[order[index]]

            except IndexError:

                pass

    return ciphertext

def decode(key, ciphertext):

    order = {int(val): num for num, val in enumerate(key)}

    key\_length = len(key)

    # Calculate number of rows

    num\_rows = len(ciphertext) // key\_length

    # Create the encrypted matrix (how ciphertext was arranged by columns)

    encrypted\_matrix = [[''] \* key\_length for \_ in range(num\_rows)]

    # Fill the encrypted matrix column by column (as it was written during encoding)

    current\_index = 0

    for col\_index in sorted(order.keys()):

        actual\_col = order[col\_index]

        for row in range(num\_rows):

            if current\_index < len(ciphertext):

                encrypted\_matrix[row][actual\_col] = ciphertext[current\_index]

                current\_index += 1

    # Read the matrix row by row to get original plaintext

    plaintext = ''

    for row in range(num\_rows):

        for col in range(key\_length):

            plaintext += encrypted\_matrix[row][col]

    return plaintext.rstrip()  # Remove padding spaces

print("=== Task 2: Decode Function ===")

key = '3214'

text = 'HELLO'

# Encode the message

encoded\_text = encode(key, text)

# Decode the message using function

decoded\_text = decode(key, encoded\_text)

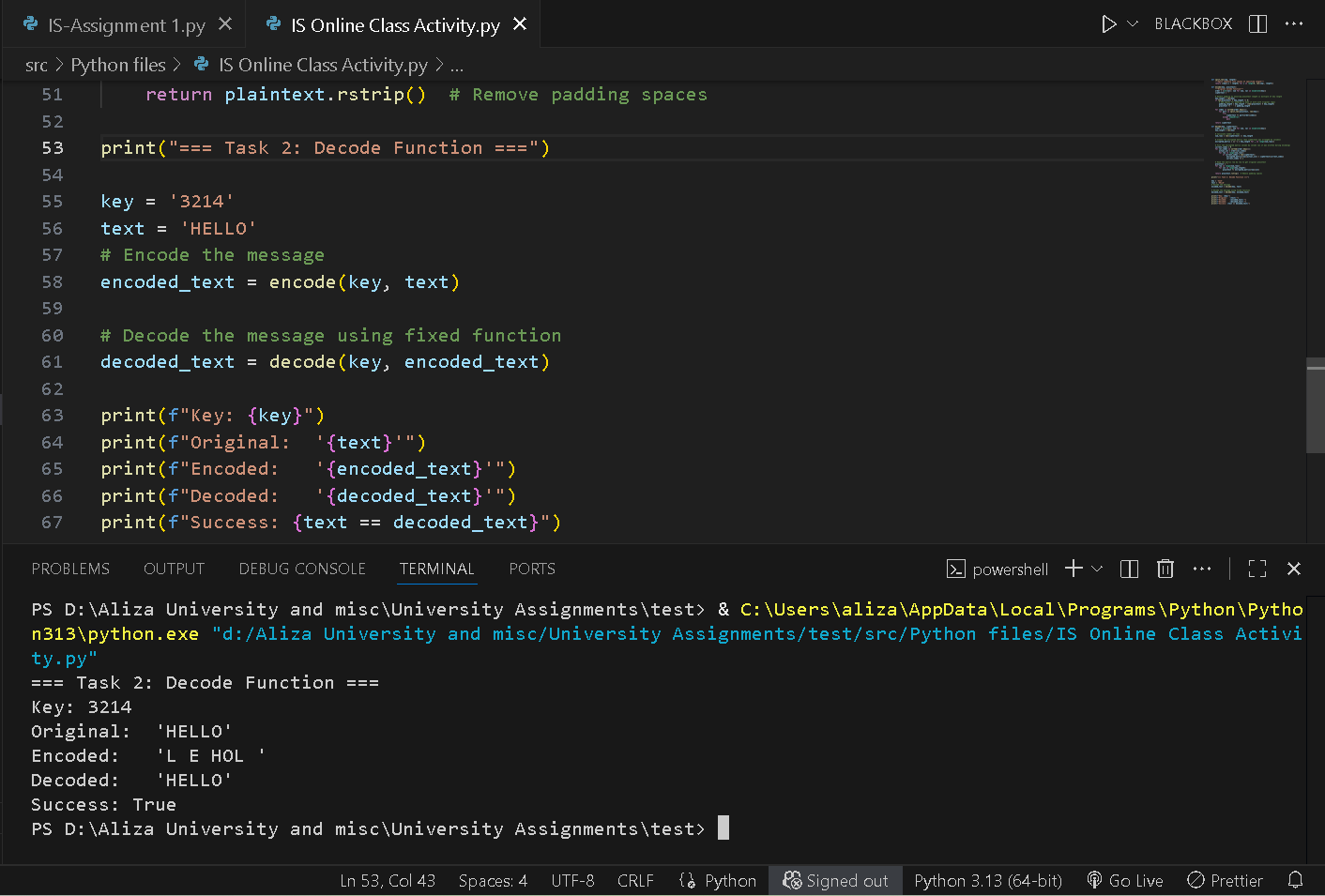
print(f"Key: {key}")

print(f"Original:  '{text}'")

print(f"Encoded:   '{encoded\_text}'")

print(f"Decoded:   '{decoded\_text}'")

print(f"Success: {text == decoded\_text}")

****

Q3. **Support for Uppercase and Lowercase Letters**

Modify the code to preserve the original case (uppercase and lowercase letters) in the plaintext.

Task: Adjust the encode function to handle both uppercase and lowercase letters, so it doesn’t always convert to lowercase.

import random

import string

def split\_len(seq, length):

    return [seq[i:i+length] for i in range(0, len(seq), length)]

def encode(key, plaintext):

    order = {int(val): num for num, val in enumerate(key)}

    key\_len = len(key)

    chars = list(plaintext)

    # Pad if needed

    if len(chars) % key\_len != 0:

        pad\_len = key\_len - (len(chars) % key\_len)

        # Use a special padding character that's unlikely to be in the original text

        chars.extend(['\0'] \* pad\_len)  # Using null character as padding

    ciphertext = ''

    for index in sorted(order.keys()):

        for part in split\_len(chars, key\_len):

            if order[index] < len(part):

                ciphertext += part[order[index]]

    return ciphertext

def decode(key, ciphertext):

    order = {int(val): num for num, val in enumerate(key)}

    key\_len = len(key)

    chars = list(ciphertext)

    num\_rows = len(chars) // key\_len

    grid = [[''] \* key\_len for \_ in range(num\_rows)]

    pos = 0

    for idx in sorted(order.keys()):

        col = order[idx]

        for row in range(num\_rows):

            if pos < len(chars):

                grid[row][col] = chars[pos]

                pos += 1

    plaintext = ''.join(''.join(row) for row in grid)

    # Remove null padding characters

    plaintext = plaintext.replace('\0', '')

    return plaintext

# Test

key = '2413'

text = "MixEDcaSe"

enc = encode(key, text)

dec = decode(key, enc)

success = text == dec

print(f"Key: {key} | '{text}' -> '{enc}' -> '{dec}' | {success}")

key = '4321'

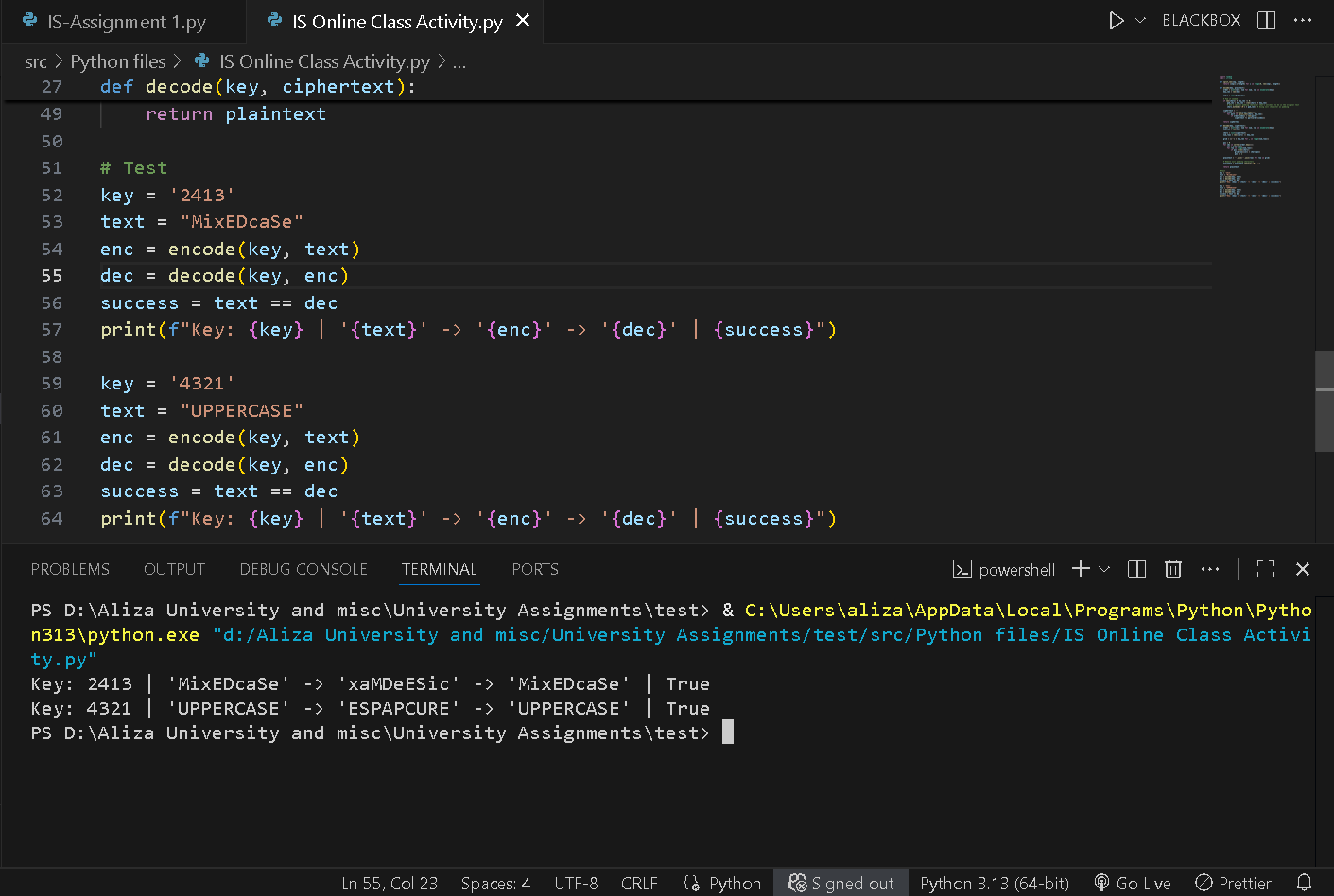
text = "UPPERCASE"

enc = encode(key, text)

dec = decode(key, enc)

success = text == dec

print(f"Key: {key} | '{text}' -> '{enc}' -> '{dec}' | {success}")

****

**Q4. Encrypt Full Sentences with Spaces**

Modify the encode function to handle spaces and punctuation without removing them.

**Task:** Ensure that spaces and punctuation are preserved and not encrypted when encoding full sentences.

import random

import string

def encode(key, plaintext):

    # Get column order from key

    order = sorted(range(len(key)), key=lambda x: key[x])

    # Store positions of non-alphabetic characters

    non\_alpha\_positions = []

    alpha\_chars = []

    for i, char in enumerate(plaintext):

        if char.isalpha():

            alpha\_chars.append(char)

        else:

            non\_alpha\_positions.append((i, char))

    original\_alpha\_count = len(alpha\_chars)

    # Pad alphabetic characters if needed

    key\_len = len(key)

    if len(alpha\_chars) % key\_len != 0:

        pad\_len = key\_len - (len(alpha\_chars) % key\_len)

        alpha\_chars.extend(['X'] \* pad\_len)

    # Create grid and encrypt

    num\_rows = len(alpha\_chars) // key\_len

    grid = [alpha\_chars[i:i+key\_len] for i in range(0, len(alpha\_chars), key\_len)]

    # Read columns in key order

    encrypted\_alpha = []

    for col in order:

        for row in range(num\_rows):

            encrypted\_alpha.append(grid[row][col])

    # Build final string by inserting non-alphabetic characters at their original positions

    result\_chars = list(encrypted\_alpha)

    for pos, char in non\_alpha\_positions:

        result\_chars.insert(pos, char)

    return ''.join(result\_chars), original\_alpha\_count

def decode(key, ciphertext, original\_alpha\_count):

    # Get column order from key (same as encoding)

    order = sorted(range(len(key)), key=lambda x: key[x])

    # Extract alphabetic characters and store non-alphabetic positions

    alpha\_chars = []

    non\_alpha\_positions = []

    for i, char in enumerate(ciphertext):

        if char.isalpha():

            alpha\_chars.append(char)

        else:

            non\_alpha\_positions.append((i, char))

    # Remove any padding by using original alpha count

    key\_len = len(key)

    total\_encrypted\_alpha = (original\_alpha\_count + key\_len - 1) // key\_len \* key\_len

    alpha\_chars = alpha\_chars[:total\_encrypted\_alpha]

    # Reconstruct grid

    num\_rows = len(alpha\_chars) // key\_len

    grid = [[''] \* key\_len for \_ in range(num\_rows)]

    # Fill columns in key order

    pos = 0

    for col in order:

        for row in range(num\_rows):

            if pos < len(alpha\_chars):

                grid[row][col] = alpha\_chars[pos]

                pos += 1

    # Read rows to get original alphabetic characters

    decrypted\_alpha = []

    for row in grid:

        decrypted\_alpha.extend(row)

    # Remove padding and keep only original alphabetic characters

    decrypted\_alpha = decrypted\_alpha[:original\_alpha\_count]

    # Build final string by combining decrypted alpha with non-alpha characters

    result\_chars = list(decrypted\_alpha)

    for pos, char in non\_alpha\_positions:

        if pos <= len(result\_chars):

            result\_chars.insert(pos, char)

    return ''.join(result\_chars)

# Test cases

test\_cases = [

    "Hello World",

    "This is a test sentence!",

    "MixEDcaSe WITH spaces, and punctuation!",

    "A"]

print("Testing columnar transposition with punctuation:")

for text in test\_cases:

    key = '2413'

    enc, alpha\_count = encode(key, text)

    dec = decode(key, enc, alpha\_count)

    print(f"'{text}' -> '{enc}' -> '{dec}' | {text == dec}")

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Q5. Dynamic Key Generation**

Automatically generate a random key if the user does not provide one. Task: Write a function that generates a random key based on the length of the plaintext.

import random

import string

def generate\_key(plaintext\_length):

    """Generate a random key based on plaintext length"""

    # Determine key length (between 3 and 8, or based on text length)

    if plaintext\_length <= 5:

        key\_length = 3

    elif plaintext\_length <= 10:

        key\_length = 4

    elif plaintext\_length <= 20:

        key\_length = 5

    else:

        key\_length = min(8, max(4, plaintext\_length // 6))

    # Create a random permutation of numbers 1 to key\_length

    numbers = list(range(1, key\_length + 1))

    random.shuffle(numbers)

    # Convert to string key

    key = ''.join(str(num) for num in numbers)

    return key

def encode(key, plaintext):

    # Get column order from key

    order = sorted(range(len(key)), key=lambda x: key[x])

    # Store positions of non-alphabetic characters

    non\_alpha\_positions = []

    alpha\_chars = []

    for i, char in enumerate(plaintext):

        if char.isalpha():

            alpha\_chars.append(char)

        else:

            non\_alpha\_positions.append((i, char))

    original\_alpha\_count = len(alpha\_chars)

    # Pad alphabetic characters if needed

    key\_len = len(key)

    if len(alpha\_chars) % key\_len != 0:

        pad\_len = key\_len - (len(alpha\_chars) % key\_len)

        alpha\_chars.extend(['X'] \* pad\_len)

    # Create grid and encrypt

    num\_rows = len(alpha\_chars) // key\_len

    grid = [alpha\_chars[i:i+key\_len] for i in range(0, len(alpha\_chars), key\_len)]

    # Read columns in key order

    encrypted\_alpha = []

    for col in order:

        for row in range(num\_rows):

            encrypted\_alpha.append(grid[row][col])

    # Build final string by inserting non-alphabetic characters at their original positions

    result\_chars = list(encrypted\_alpha)

    for pos, char in non\_alpha\_positions:

        result\_chars.insert(pos, char)

    return ''.join(result\_chars), original\_alpha\_count

def decode(key, ciphertext, original\_alpha\_count):

    # Get column order from key (same as encoding)

    order = sorted(range(len(key)), key=lambda x: key[x])

    # Extract alphabetic characters and store non-alphabetic positions

    alpha\_chars = []

    non\_alpha\_positions = []

    for i, char in enumerate(ciphertext):

        if char.isalpha():

            alpha\_chars.append(char)

        else:

            non\_alpha\_positions.append((i, char))

    # Remove any padding by using original alpha count

    key\_len = len(key)

    total\_encrypted\_alpha = (original\_alpha\_count + key\_len - 1) // key\_len \* key\_len

    alpha\_chars = alpha\_chars[:total\_encrypted\_alpha]

    # Reconstruct grid

    num\_rows = len(alpha\_chars) // key\_len

    grid = [[''] \* key\_len for \_ in range(num\_rows)]

    # Fill columns in key order

    pos = 0

    for col in order:

        for row in range(num\_rows):

            if pos < len(alpha\_chars):

                grid[row][col] = alpha\_chars[pos]

                pos += 1

    # Read rows to get original alphabetic characters

    decrypted\_alpha = []

    for row in grid:

        decrypted\_alpha.extend(row)

    # Remove padding and keep only original alphabetic characters

    decrypted\_alpha = decrypted\_alpha[:original\_alpha\_count]

    # Build final string by combining decrypted alpha with non-alpha characters

    result\_chars = list(decrypted\_alpha)

    for pos, char in non\_alpha\_positions:

        if pos <= len(result\_chars):

            result\_chars.insert(pos, char)

    return ''.join(result\_chars)

# Test with dynamic key generation

test\_cases = [

    "Hello World",

    "This is a test sentence!",

    "MixEDcaSe WITH spaces, and punctuation!",

    "Programming is fun!",

    "Short",

    "A",

    "Hi!",

    "Test!"]

print("Testing with Dynamic Key Generation:")

for text in test\_cases:

    # Generate key based on text length

    auto\_key = generate\_key(len(text))

    # Encode and decode with auto-generated key

    enc, alpha\_count = encode(auto\_key, text)

    dec = decode(auto\_key, enc, alpha\_count)

    print(f"Auto Key: {auto\_key} (length: {len(auto\_key)})")

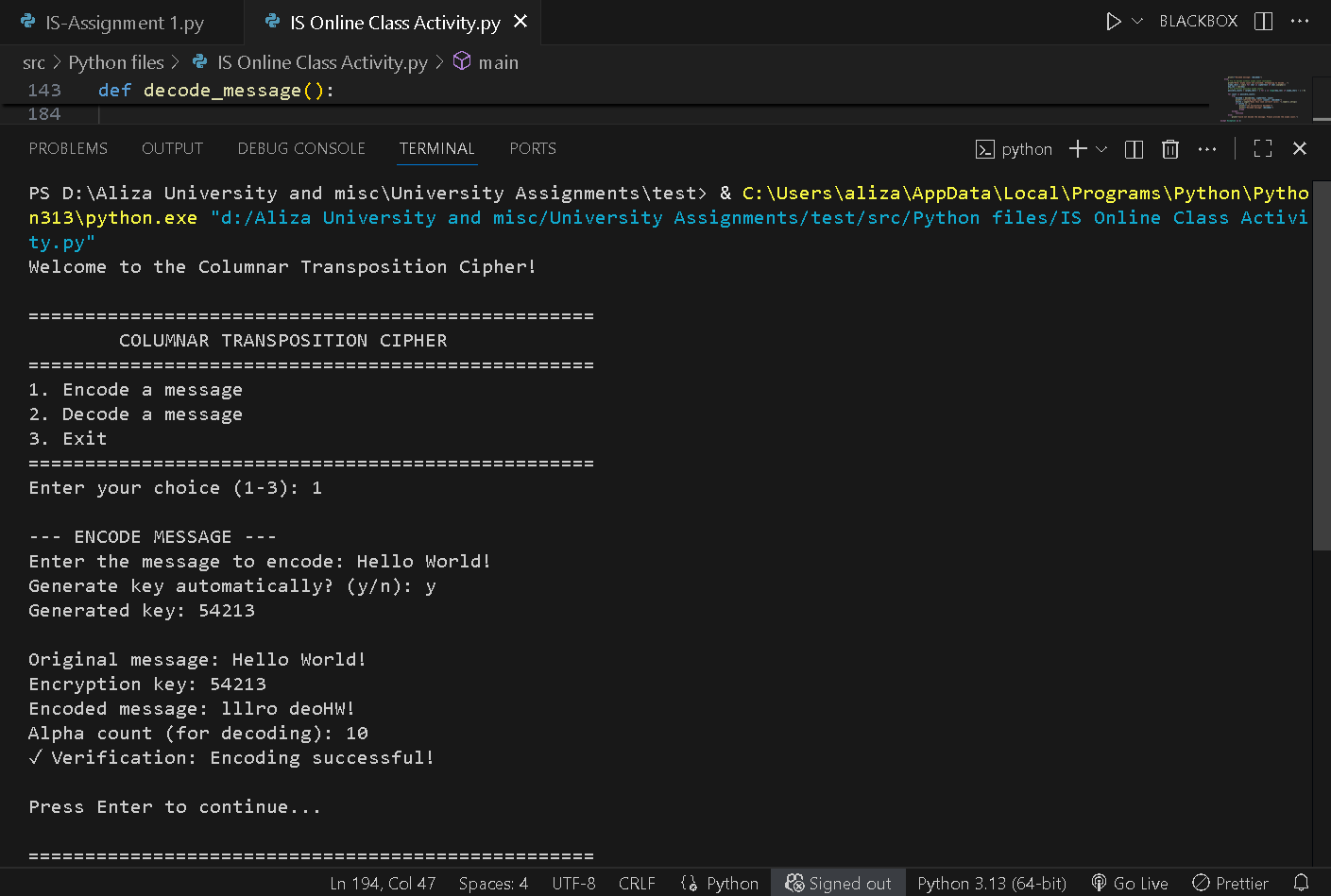
    print(f"'{text}' -> '{enc}' -> '{dec}' | {text == dec}")

A screenshot of a computer program

AI-generated content may be incorrect.

**Q6. Add a Menu Interface**

Create a simple command-line interface where the user can choose to encode or decode a message. **Task**: Write a menu system where the user can input a choice to either encode, decode, or exit.



A screenshot of a computer

AI-generated content may be incorrect.