Prac 1\21430790_Prac_1.py

```python
# Name: Alexandros Theodorou
# Student num: 21430790

'''
IMPORTANT!!

- Due 13 March 2024 (before 8h30)
- No late submissions (AMS and Turnitin) accepted after 8h30!
- The prac test starts at 10h30 in the Netlabs.

- Rename this file to "<YourStudentNumber>_Prac_1.py", for example: "19056789_Prac_1.py"
- Comment your code (follow best practice)
- Submit .py to AMS and a .pdf to ClickUp (TurnItIn)
- Also, please upload your turnitin receipt to the AMS.
- Remove all print statements - and helper functions (that weren't provided) - used for unit testing.

- Please read the practical guide for instructions!
'''

import string
import numpy as np
#from PIL import Image
#import matplotlib.pyplot as plt

# 3.1 Playfair Cipher
# ---------------------------------------------------------------------------------------------------
####################################################################################################

def playfair_get_key(isText: bool, key: str) -> np.ndarray: # 3.1.1
    if isText:
        alph = list(string.ascii_lowercase)
        x=0
        new_key = ""
        while x<len(key):
            if key[x].lower() in alph and key[x].lower() not in new_key: # remove not letters and duplicates
                new_key+=key[x].lower()
            x+=1
        key_list = list(new_key)
        key_matrix = key_list

        y=0
        while len(key_matrix)<25:
            if alph[y] not in key_list and alph[y]!= "j": # fill rest of key matrix
                key_matrix.append(alph[y])                  # with chars not in key or not j
            y+=1
        key_matrix = np.array(key_matrix)
        key_matrix= key_matrix.reshape(5,5)

        return  key_matrix
    else:
        unique_chars = []
        for char in key:
            if char not in unique_chars:
                unique_chars.append(char)

        ascii_values = [ord(char) % 256 for char in unique_chars]


        all_values = list(range(256))
        for val in ascii_values:
            if val in all_values:
                all_values.remove(val)

        key_values = ascii_values + all_values

        key_matrix = np.array(key_values).reshape(16, 16)

        return key_matrix

def playfair_get_pos_in_key(val, val2, keyMat: np.ndarray) -> np.ndarray: # 3.1.2
    x_val_1 =-1
    y_val_1 = -1
    x_val_2= -1
    y_val_2 = -1

    # Find values in arrays
    for i in range(keyMat.shape[0]):
        for j in range(keyMat.shape[1]):
            if val == keyMat[i][j]:
                x_val_1 = i
                y_val_1 = j
            if val2 == keyMat[i][j]:
                x_val_2 = i
                y_val_2 = j
    pos = np.array([x_val_1,y_val_1,x_val_2,y_val_2])
    return pos

def playfair_get_encryption_pos(pos: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.1.3
    x_len = keyMat.shape[0]
    y_len = keyMat.shape[1]
    # Both vals in same row
    if pos[0] == pos[2]:
        pos[1] = pos[1] + 1 if pos[1] < x_len - 1 else 0
        pos[3] = pos[3] +1 if pos[3]< x_len-1 else 0
        return pos
    # Both vals in same column
    if pos[1] == pos[3]:
        pos[0] = pos[0] +1 if pos[0]<y_len-1 else 0
        pos[2] =pos[2]+1 if pos[2]< y_len-1 else 0
        return pos
    else:
        temp = pos[1]
        pos[1] = pos[3]
        pos[3] = temp
        return pos

def playfair_get_decryption_pos(pos: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.1.4
    x_len = keyMat.shape[0]
    y_len = keyMat.shape[1]
    # Both vals in same row
    if pos[0] == pos[2]:
        pos[1] = pos[1] -1 if pos[1]!=0 else x_len-1
        pos[3] = pos[3]-1 if pos[3]!= 0 else x_len-1
        return pos
    # Both vals in same column
    if pos[1] == pos[3]:
        pos[0] = pos[0]-1 if pos[0]!=0 else y_len-1
        pos[2] = pos[2]-1 if pos[2]!=0 else y_len-1
        return pos
    else:
        temp = pos[1]
        pos[1] = pos[3]
        pos[3] = temp
        return pos

def playfair_preprocess_text(plaintext: str) -> str: # 3.1.5
    alph = list(string.ascii_lowercase)
    x=0
    new_text = ""
    while x<len(plaintext):
        if plaintext[x].lower() in alph:
            if plaintext[x].lower() == "j":
                    new_text+= "i"
            else:
                    new_text+=plaintext[x].lower()

        x+=1
    y = 0
    while y<len(new_text)-2:
        x1 = new_text[y]
        x2 = new_text[y+1]

        if x1 == x2:
            new_text = new_text[:y+1]+'x' +new_text[y+1:]
            y+=2

        else:
            y+=2
    if len(new_text) % 2 !=0:
        new_text = new_text+ 'x'
    return new_text

def playfair_encrypt_text(plaintext: str, key: str) -> str: # 3.1.6

    new_text = playfair_preprocess_text(plaintext)
    key_matrix = playfair_get_key(True,key)
    cipher = ""
    # Encrypt every two characters
    for i in range(0,len(new_text),2):
        x = new_text[i]
        y = new_text[i+1]
        pos= playfair_get_pos_in_key(x,y,key_matrix)
        new_pos = playfair_get_encryption_pos(pos,key_matrix)
        cipher = cipher + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]
    return cipher

def playfair_decrypt_text(ciphertext: str, key: str) -> str: # 3.1.7
    #new_text = playfair_preprocess_text(ciphertext)
    key_matrix = playfair_get_key(True,key)
    plaintext = ""
    # decrypt every two characters
    for i in range(0,len(ciphertext),2):
        x = ciphertext[i]
        y = ciphertext[i+1]
```

```python
            pos= playfair_get_pos_in_key(x,y,key_matrix)
            new_pos = playfair_get_decryption_pos(pos,key_matrix)
            plaintext = plaintext + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]

    x= 0
    new_text = ""
    while x<len(plaintext):
        if plaintext[x]!= 'x':
            new_text = new_text+ plaintext[x]
        x+=1
    return new_text

def playfair_preprocess_image(plaintext: np.ndarray) -> np.ndarray: # 3.1.8
    flattened = plaintext.flatten()
    # replace 129 with 128
    flattened = np.where(flattened == 129, 128, flattened)

    if len(flattened) % 2 != 0:
        flattened = np.append(flattened, 129)

    return flattened

def playfair_remove_image_padding(plaintextWithPadding: np.ndarray) -> np.ndarray: # 3.1.9
    new_text = []
    for i in range(len(plaintextWithPadding)):
        if plaintextWithPadding[i] != 129:
            new_text.append(plaintextWithPadding[i])
    return np.array(new_text)

def playfair_encrypt_image(plaintext: np.ndarray, key: str) -> np.ndarray: # 3.1.10
    processed_image = playfair_preprocess_image(plaintext)
    key_matrix = playfair_get_key(False, key)

    ciphertext = np.zeros_like(processed_image)
    # Encrypt two pixels at a time
    for i in range(0, len(processed_image), 2):
        if i+1 < len(processed_image):
            x = processed_image[i]
            y = processed_image[i+1]

            pos = playfair_get_pos_in_key(x, y, key_matrix)

            new_pos = playfair_get_encryption_pos(pos, key_matrix)
            ciphertext[i] = key_matrix[new_pos[0]][new_pos[1]]
            ciphertext[i+1] = key_matrix[new_pos[2]][new_pos[3]]

    return ciphertext

def playfair_decrypt_image(removePadding: bool, ciphertext: np.ndarray, key: str) -> np.ndarray: # 3.1.11
    key_matrix = playfair_get_key(False, key)
    plaintext = np.zeros_like(ciphertext)

    # Decrypt two pixels at a time
    for i in range(0, len(ciphertext), 2):
        if i+1 < len(ciphertext):
            x = ciphertext[i]
            y = ciphertext[i+1]

            pos = playfair_get_pos_in_key(x, y, key_matrix)
            new_pos = playfair_get_decryption_pos(pos, key_matrix)
            plaintext[i] = key_matrix[new_pos[0]][new_pos[1]]
            plaintext[i+1] = key_matrix[new_pos[2]][new_pos[3]]

    if removePadding:
        plaintext = playfair_remove_image_padding(plaintext)

    return plaintext

def playfair_convert_to_image(imageData: np.ndarray, originalShape) -> np.ndarray: # 3.1.12
    # Dimentions of image
    required_size = originalShape[0] * originalShape[1] * originalShape[2]
    if len(imageData) < required_size:
        padding_needed = required_size - len(imageData)
        padded_data = np.pad(imageData, (0, padding_needed), 'constant', constant_values=0)
        reshaped_image = padded_data.reshape(originalShape)
    elif len(imageData) > required_size:
        reshaped_image = imageData[:required_size].reshape(originalShape)
    else:
        reshaped_image = imageData.reshape(originalShape)

    return reshaped_image

# ----------------------------------------------------------------------------------------------------

# 3.2 Hill Cipher
# ----------------------------------------------------------------------------------------------------

def hill_get_key(isText: bool, key: str) -> np.ndarray:
    if isText:
        key_list = list(key)
        for i in range(len(key_list)):
            key_list[i] = ord(key_list[i]) % 97
        key_list = np.array(key_list)
        if len(key_list) == 4:
            key_list = key_list.reshape(2, 2)
        if len(key_list) == 9:
            key_list = key_list.reshape(3, 3)
        det = np.linalg.det(key_list)
        if det == 0:
            for i in range(key_list.shape[0]):
                for j in range(key_list.shape[1]):
                    key_list[i][j] = -1
        return key_list
    else:
        # For images, similar process but with modulo 256
        key_list = list(key)
        for i in range(len(key_list)):
            key_list[i] = ord(key_list[i])  # ASCII value directly, no modulo 97
        key_list = np.array(key_list)
        if len(key_list) == 4:
            key_list = key_list.reshape(2, 2)
        elif len(key_list) == 9:
            key_list = key_list.reshape(3, 3)
        else:
            return np.array([[-1]])  # Invalid size

        det = np.linalg.det(key_list) % 256
        # Check if determinant has an inverse in modulo 256
        has_inverse = False
        for i in range(1, 256):
            if (int(det) * i) % 256 == 1:
                has_inverse = True
                break

        if det == 0 or not has_inverse:
            return np.full(key_list.shape, -1)

        return key_list

def hill_get_inv_key(isText: bool, keyMat: np.ndarray) -> np.ndarray:  # 3.2.2

    mod = 26 if isText else 256

    # Calculate the determinant of the key matrix
    det = int(round(np.linalg.det(keyMat)))

    # Ensure the determinant is positive
    det = det % mod

    # Find modular multiplicative inverse of the determinant
    det_inv = pow(det, -1, mod)

    # Calculate the adjugate matrix
    n = keyMat.shape[0]
    adj = np.zeros(keyMat.shape, dtype=int)

    for i in range(n):
        for j in range(n):
            minor = np.delete(np.delete(keyMat, i, axis=0), j, axis=1)
            cofactor = round(np.linalg.det(minor))
            cofactor *= (-1) ** (i + j)
            adj[j, i] = cofactor

    # Calculate the inverse key
    inv_key = (det_inv * adj) % mod

    return inv_key

def hill_process_group(isText: bool, group: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.2.3
    if isText:
        result = np.dot(keyMat, group) % 26
        return result
    else:
        result = np.dot(keyMat, group) % 256
        return result

def hill_pre_process_text(plaintext: str, keyLength: int) -> np.ndarray: # 3.2.4
    alph = list(string.ascii_lowercase)
    new_text =[]
    for s in plaintext:
        if s.lower() in alph:
            new_text.append(s.lower())
    if keyLength == 4:
        while len(new_text) % 2 !=0:
            new_text.append("x")
    if keyLength == 9:
        while len(new_text) % 3 !=0:
            new_text.append("x")

    for i in range(len(new_text)):
        new_text[i] = ord(new_text[i]) % 97

    return np.array(new_text)

def hill_encrypt_text(plaintext: str, key: str) -> str: # 3.2.5


    key_length = len(key)
    key_mat = hill_get_key(True, key)

    # Check if key is valid
    if -1 in key_mat:
        return "Invalid Key"

    processed_text = hill_pre_process_text(plaintext, key_length)

    # Determine the group size based on key length
    group_size = 2 if key_length == 4 else 3

    # Encrypt
    result = []
    for i in range(0, len(processed_text), group_size):
        group = processed_text[i:i+group_size]
        encrypted_group = hill_process_group(True, group, key_mat)

        # Convert numbers back to characters
        for num in encrypted_group:
            result.append(chr(num + 97))

    return "".join(result)

def hill_decrypt_text(ciphertext: str, key: str) -> str: # 3.2.6

    import numpy as np

    key_length = len(key)
    key_mat = hill_get_key(True, key)

    # Check if key is valid
    if -1 in key_mat:
        return "Invalid Key"

    inv_key_mat = hill_get_inv_key(True, key_mat)

    processed_text = []
    for char in ciphertext:
        processed_text.append(ord(char.lower()) - 97)
    processed_text = np.array(processed_text)

    # Determine the group size based on key length
    group_size = 2 if key_length == 4 else 3

    # Decrypt
    result = []
    for i in range(0, len(processed_text), group_size):
        group = processed_text[i:i+group_size]
        decrypted_group = hill_process_group(True, group, inv_key_mat)

        # Convert numbers back to characters
        for num in decrypted_group:
            result.append(chr(num + 97))

    # remove padding
    plaintext = "".join(result)
    while plaintext.endswith('x'):
        plaintext = plaintext[:-1]

    return plaintext

def hill_pre_process_image(plaintext: np.ndarray, keyLength: int) -> np.ndarray: # 3.2.7

    flattened = plaintext.flatten()

    flattened = np.where(flattened == 129, 128, flattened)

    # Determine the group size
    group_size = 2 if keyLength == 4 else 3

    # Add padding if necessary
    padding_needed = (group_size - (len(flattened) % group_size)) % group_size
    if padding_needed > 0:
        padding = np.full(padding_needed, 129)
        flattened = np.concatenate((flattened, padding))

    return flattened

def hill_encrypt_image(plaintext: np.ndarray, key: str) -> np.ndarray: # 3.2.8


    # Get the key matrix
    key_length = len(key)
    key_mat = hill_get_key(False, key)

    # Pre-process the image
    processed_image = hill_pre_process_image(plaintext, key_length)

    # Determine the group size based on key length
    group_size = 2 if key_length == 4 else 3

    # Encrypt
    result = np.zeros_like(processed_image)
    for i in range(0, len(processed_image), group_size):
        group = processed_image[i:i+group_size]
        encrypted_group = hill_process_group(False, group, key_mat)
        result[i:i+group_size] = encrypted_group

    return result

def hill_decrypt_image(ciphertext: np.ndarray, key: str) -> np.ndarray: # 3.2.9



    # Get the key matrix
    key_length = len(key)
    key_mat = hill_get_key(False, key)
```

```python

    # Get inverse key matrix
    inv_key_mat = hill_get_inv_key(False, key_mat)

    # Determine the group size based on key length
    group_size = 2 if key_length == 4 else 3

    # Decrypt
    result = np.zeros_like(ciphertext)
    for i in range(0, len(ciphertext), group_size):
        group = ciphertext[i:i+group_size]
        decrypted_group = hill_process_group(False, group, inv_key_mat)
        result[i:i+group_size] = decrypted_group

    # Remove padding
    result = result[result != 129]

    return result

def hill_convert_to_image(imageData: np.ndarray, originalShape: tuple) -> np.ndarray: # 3.2.10


    # Calculate the size needed for the original shape
    original_size = originalShape[0] * originalShape[1] * originalShape[2]

    # Pad where needed
    if len(imageData) < original_size:
        padding_needed = original_size - len(imageData)
        imageData = np.concatenate((imageData, np.zeros(padding_needed)))

    # In case the decoded data is longer than the original
    if len(imageData) > original_size:
        imageData = imageData[:original_size]

    # Reshape the data to the original shape
    reshaped_image = imageData.reshape(originalShape)

    return reshaped_image

# ----------------------------------------------------------------------------------------------------

# 3.3 Row Transposition Cipher
# ----------------------------------------------------------------------------------------------------

def row_gen_key(key: str) -> np.ndarray: # 3.3.1
    ascii_key = []
    new_key = []
    key_matrix = []

    # get rid of duplicates
    for x in key:
        if x not in new_key:
            new_key.append(x)
    # convert to ascii
    for i in new_key:
        ascii_key.append(ord(i))

    sorted_ascii = list(ascii_key)
    sorted_ascii.sort() # sorted ascendingly to determine order
    for i in sorted_ascii:
        key_matrix.append(ascii_key.index(i))

    return np.array(key_matrix)

def row_pad_text(plaintext: str, key: np.ndarray) -> str: # 3.3.2
    length = len(key)
    while len(plaintext) % length !=0:
        plaintext = plaintext+ 'x'
    return plaintext

def row_encrypt_single_stage(plaintext: str, key: np.ndarray) -> str:
    plaintext = row_pad_text(plaintext, key)
    # dimensions of block that I am seperating it into
    key_length = len(key)
    num_rows = len(plaintext) // key_length

    ciphertext = ""
    for k in key: # each number of the key
        for i in range(num_rows):
            position = i * key_length + k # i* key_length controls row count, k then indexes to
                                          # correct cipher
            ciphertext += plaintext[position]

    return ciphertext

def row_decrypt_single_stage(ciphertext: str, key: np.ndarray) -> str:
    key_length = len(key)
    num_rows = len(ciphertext) // key_length

    # create an empty matrix to house the decrypted text
    matrix = [[''] * key_length for _ in range(num_rows)]

    # go column by column for decrypion into the matrix
    char_index = 0
    for k_index, k_value in enumerate(key):
        for i in range(num_rows):
            matrix[i][k_value] = ciphertext[char_index]
            char_index += 1

    # move lists of characters into the plaintext
    plaintext = ""
    for row in matrix:
        plaintext += ''.join(row)

    return plaintext

def row_encrypt(plaintext: str, key: str, stage: int) -> str:
    key_array = row_gen_key(key)

    ciphertext = row_encrypt_single_stage(plaintext, key_array)

    # second stage if required
    if stage > 1:
        ciphertext = row_encrypt_single_stage(ciphertext, key_array)

    return ciphertext

def row_decrypt(ciphertext: str, key: str, stage: int) -> str:

    key_array = row_gen_key(key)

    plaintext = row_decrypt_single_stage(ciphertext, key_array)

    # Second stage if required
    if stage > 1:
        plaintext = row_decrypt_single_stage(plaintext, key_array)

    # Remove padding
    plaintext = plaintext.rstrip('x')

    return plaintext

# ----------------------------------------------------------------------------------------------------

```