REPORT ON ENCRYPT FUNCTION

The first task’s code is implementing the Vigenère cipher encryption scheme.

Function definition:

def encrypt_func(plaintext, cipher_key, ciphertext):

Parameters:

-**plaintext:** The original message to be encrypted.

-**cipher_key:** The key used for encryption.

-**ciphertext:** The resulting encrypted message.

Initializations:

plaintext_counter = 0

cipher_key_counter = 0

cipher_key_len = len(cipher_key)

Variables:

-**plaintext_counter:** Tracks the position in the plaintext.

-**cipher_key_counter:** Tracks the position in the cipher key.

-**cipher_key_len:** Stores the length of the cipher key.

Main Code:

-The code loops through the plaintext. (for char in plaintext:)

-I check if the character is a letter or not. (if plaintext[plaintext_counter].isalpha():)

-I calculate how much we must shift the plaintext letter to get the encrypted letter:

letter = plaintext[plaintext_counter]

shift_by = abs(ord('a') - ord(cipher_key[cipher_key_counter].lower()))

Variables:

-**letter:** The current letter from the plaintext.

-**shift_by:** The shift value calculated based on the cipher key.

-I apply the shift to the plaintext letter:

```
if (ord(letter.lower()) + shift_by) > ord('z'):
    new_shift = abs(ord('z') - (ord(letter.lower()) + shift_by))
    shift_by = new_shift - 1
    shifted_letter = chr(ord('a') + shift_by)
else:
    shifted_letter = chr(ord(letter.lower()) + shift_by)
```

-This applies the shift to the current letter, wrapping it around if necessary. (Wrapping around means if the letter + shift amount is greater than the numerical position in the alphabet of the letter ‘z’)

-I do necessary case preservation when putting the ciphertext letters into the ciphertext string:

```
if letter.isupper():
    ciphertext += shifted_letter.upper()
else:
    ciphertext += shifted_letter
```

-I update my counters that are counting through plaintext letters and cipher_key letters. I additionally, have a check that sees if the cipher_key_counter exceeds the length of the cipher key so we can reset the cipher_key_counter to 0 and keep encrypting the plaintext:

```
plaintext_counter = plaintext_counter + 1
cipher_key_counter = cipher_key_counter + 1


if cipher_key_counter >= cipher_key_len:
    cipher_key_counter = 0
```

-I then write the else to my if plaintext[plaintext_counter].isalpha(): statement, which handles the case if the plaintext letter is not a letter:

```
else:
    ciphertext += plaintext[plaintext_counter]
    plaintext_counter = plaintext_counter + 1
```

-We put the non-alphabetic character into the ciphertext and then shift the plaintext_counter to the next position.

Calling the Function:

Finally, we have the code that takes the user input and passes it to the function:

```
plaintext = input()
cipher_key = input()
ciphertext = ""

print(encrypt_func(plaintext, cipher_key, ciphertext))
```

REPORT ON DECRYPT FUNCTION

The second task's code is implementing the Vigenère cipher decryption scheme.

Function Definition:

def decrypt_func(ciphertext, cipher_key, plaintext, ciphertext_counter, cipher_key_counter, cipher_key_len):

Parameters:

-**ciphertext:** The encrypted text that needs to be decrypted.

-**cipher_key:** The key used for decryption.

-**plaintext:** The resulting decrypted text.

-**ciphertext_counter:** A counter to keep track of the current position in the ciphertext.

-**cipher_key_counter:** A counter to keep track of the current position cipher key.

-**cipher_key_len:** The length of the key.

Main Code:

-I iterate through each character in the ciphertext and do the following:

-I check if the character is alphabetic:

```
if ciphertext[ciphertext_counter].isalpha():
```

-I determine the shift value:

```
shift_by = abs(ord('a') - ord(cipher_key[cipher_key_counter].lower()))
```

-The shift value is calculated based on the corresponding character in the cipher key.

-I decrypt the character:

```
if (ord(letter.lower()) - shift_by) < ord('a'):
    new_shift = abs(ord('a') - (ord(letter.lower()) - shift_by))
    shift_by = new_shift - 1
    shifted_letter = chr(ord('z') - shift_by)
```

-I do this by subtracting the shift_by value from the character in the plaintext, allowing us to go back to the plaintext character.

-If the shift results in going past “a” then I wrap around to “z” as seen above. In regular cases the following code is used:

```
else:
    shifted_letter = chr(ord(letter.lower()) - shift_by)
```

-I preserve the case of the original character:

```
if letter.isupper():
    plaintext += shifted_letter.upper()
else:
    plaintext += shifted_letter
```

-I update the text counters for the ciphertext and cipher key. If the cipher key counter goes past its length, I reset the counter:

```
ciphertext_counter = ciphertext_counter + 1
cipher_key_counter = cipher_key_counter + 1


if cipher_key_counter >= cipher_key_len:
    cipher_key_counter = 0
```

-This else statement correlates to the if statement that tests if the character of the ciphertext is a letter, this else statement is entered if the ciphertext letter was not a letter. I just place the character into the plaintext as is and add 1 to the ciphertext counter :

```
else:
    plaintext += ciphertext[ciphertext_counter]
```

```
ciphertext_counter = ciphertext_counter + 1
```

Calling the Function:

Finally, we have the code that takes the user input and passes it to the function:

```
ciphertext = input()

cipher_key = input()

plaintext = ""


ciphertext_counter = 0

cipher_key_counter = 0

cipher_key_len = len(cipher_key)

print(decrypt_func(ciphertext, cipher_key, plaintext, ciphertext_counter,
cipher_key_counter, cipher_key_len))
```

Sources:

The algorithm for the code for the first two tasks was derived from the slides and the discussions on the Vigenère cipher in class.

REPORT ON DECRYPTION WITH KEY LENGTH

The main idea for this algorithm was derived from: https://inventwithpython.com/cracking/chapter20.html

The Vigenère cipher in practice is like a shift cipher. Each character of the plaintext was shifted by the numerical position in the alphabet of each character in the cipher key (a = 1, b = 2, c = 3, …). This shifting of the plaintext results in the ciphertext. So, I decided to divide and group up the characters of the ciphertext by the cipher key character that encrypted them. Before this I clean up the text removing any non-alphabetic characters. For example:

AEWJKHUPMSTUKS

Key length: 2

1st group (Encrypted by the first character of the key):

AJUSK

2nd group (Encrypted by the second character of the key):

EKPTS

What we know is that for both these groups of characters the cipher key character that encrypted them can be any character A – Z. Which means any character 1 – 26.

So, for each of these groups of letters I shift them back 1, 26 times to find which position of characters is the optimal one. For example:

AJUSK → ZITRJ → YHSQI → …

How do I determine the optimal value?

For each of these "shifted groupings" of letters I calculate a chi-squared test (https://en.wikipedia.org/wiki/Chi-squared_test):

$$x^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

To get the **expected value** in this formula I multiply the **length of the shifted group** by the **known frequency in the English language** of each letter in the group (https://en.wikipedia.org/wiki/Letter_frequency). The **observed value** is the frequency of the letter in the group. I do this for each letter in the group and in the end sum all the chi-squared values.

In the end for each of the original grouping of letters, in my example above there were 2 original grouping of letters. I will get **26 chi-squared values.**

Out of each of the sets of 26 chi-squared values I will find the **minimum (optimal value)** and mark the **index** at which they occur.

After this I will get **n indexes** (these indexes were where the minimum chi-squared values were found), which determine how much I must shift from the letter 'a' to get the cipher key character (Here n is the length of the key). For example:

GROUP 1:
Minimum chi-squared value was found to be at index 2.

GROUP 2:

Minimum chi-squared value was found to be at index 5

So,

CIPHER KEY:	__C__	__E__
	'a' + 2	'a' + 5

Now that I have the cipher key, I just reuse my code from the decrypt function in task 2.

Two Key Helper Functions:

get_group:

-This is the function that extracts letters in groups from the ciphertext at intervals equal to the length of the key.

rotationcaesarcipher:

-This decodes a group of letters using a Caesar cipher with a given shift.

REPORT ON CIPHERTEXT ONLY DECRYPTION

The idea for this algorithm was derived from the Kasiski examination: https://en.wikipedia.org/wiki/Kasiski_examination#:~:text=The%20number%20of%20%22coincidences%22%20goes,described%20above%20using%20frequency%20analysis.

The algorithm compares the original ciphertext to shifted versions of the ciphertext. As it compares the ciphertext and shifted versions the **number of matching pairs of letters** are recorded. Before this I clean up the text removing any non-alphabetic characters. For example:

AEWJKHUPMSTUKS

First iteration:

AEWJKHUPMSTUKS NO MATCHES

 AEWJKHUPMSTUK

Second iteration:

AEWJKHUPMSTUKS NO MATCHES

  AEWJKHUPMSTU

….

AEWJKHUPMSTUK**S** 1 MATCHING PAIR

    AEWJKHUPM**S**

….

In my case I iterate for $\frac{1}{2}$ **the length** of the ciphertext. This gives me enough data (matching pairs) to determine the key length.

I store all the counts of matching pairs collected from each iteration into an array.

I use the statistics module from python to calculate the mean and standard deviation of the data.

I find the peaks in the data by calculating if the values in each index of the matching pairs array are **1.5 standard deviations** above the mean. After multiple tests, 1.5 seemed to be the optimal value to get the **“spikes”** in the data. The index, in the matching pairs array, of these “spikes” are stored into another array. I add 1 to the indices to account for the fact that the indexes start from 0.

I iterate through the “spikes” array and calculate the differences of **consecutive numbers** in the array and store those differences in another array. Now I can use the statistics module to get the mode of the data in the differences array. Whatever the mode is this will be the likely key length.

Now that I have the key length, I reuse the code I have from the function in task 3 to get the cipher key and plaintext.

Key Helper Function:

count_matching_positions:

-This is the function that counts the number of matching pairs between the original ciphertext and a shifted version of it.