**Information Security Practical File**

Name: Shrutika Shaw

Subject: Information Security

Course: B.Sc (Hons) Computer Science

College Roll No: 2019345

Examination Roll No: 19066570040

1. Implement the error correcting code.

Code:

def calcRedundantBits(m):

for i in range(m):

if(2 \*\* i >= m + i + 1):

return i

def posRedundantBits(data, r):

j = 0

k = 1

m = len(data)

res = ''

for i in range(1, m + r + 1):

if(i == 2 \*\* j):

res = res + '0'

j += 1

else:

res = res + data[-1 \* k]

k += 1

return res[::-1]

def calcParityBits(arr, r):

n = len(arr)

for i in range(r):

val = 0

for j in range(1, n + 1):

if(j & (2 \*\* i) == (2 \*\* i)):

val = val ^ int(arr[-1 \* j])

arr = arr[:n - (2 \*\* i)] + str(val) + arr[n - (2 \*\* i) + 1:]

return arr

def detectError(arr, nr):

n = len(arr)

res = 0

for i in range(nr):

val = 0

for j in range(1, n + 1):

if(j & (2 \*\* i) == (2 \*\* i)):

val = val ^ int(arr[-1 \* j])

res = res + val \* (10 \*\* i)

return int(str(res), 2)

data = input("Enter data: ")

m = len(data)

r = calcRedundantBits(m)

arr = posRedundantBits(data, r)

arr = calcParityBits(arr, r)

print("Data transfered is: " + arr)

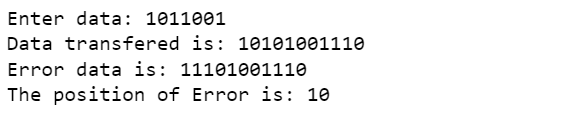
arr = '11101001110'

print("Error data is: " + arr)

correction = detectError(arr, r)

print("The position of Error is: " + str(correction))

Output:



1. Implement the error detecting code.

Code:

def findChecksum(SentMessage, k):

c1 = SentMessage[0:k]

c2 = SentMessage[k:2\*k]

c3 = SentMessage[2\*k:3\*k]

c4 = SentMessage[3\*k:4\*k]

Sum = bin(int(c1, 2)+int(c2, 2)+int(c3, 2)+int(c4, 2))[2:]

if(len(Sum) > k):

x = len(Sum)-k

Sum = bin(int(Sum[0:x], 2)+int(Sum[x:], 2))[2:]

if(len(Sum) < k):

Sum = '0'\*(k-len(Sum))+Sum

Checksum = ''

for i in Sum:

if(i == '1'):

Checksum += '0'

else:

Checksum += '1'

return Checksum

def checkReceiverChecksum(ReceivedMessage, k, Checksum):

c1 = ReceivedMessage[0:k]

c2 = ReceivedMessage[k:2\*k]

c3 = ReceivedMessage[2\*k:3\*k]

c4 = ReceivedMessage[3\*k:4\*k]

ReceiverSum = bin(int(c1, 2)+int(c2, 2)+int(Checksum, 2) + int(c3, 2)+int(c4, 2)+int(Checksum, 2))[2:]

if(len(ReceiverSum) > k):

x = len(ReceiverSum)-k

ReceiverSum = bin(int(ReceiverSum[0:x], 2)+int(ReceiverSum[x:], 2))[2:]

ReceiverChecksum = ''

for i in ReceiverSum:

if(i == '1'):

ReceiverChecksum += '0'

else:

ReceiverChecksum += '1'

return ReceiverChecksum

#SentMessage = "10010101011000111001010011101100"

SentMessage=input("Enter message sent: ")

k = 8

#ReceivedMessage = "10000101011000111001010011101101"

ReceivedMessage=input("Enter message received: ")

Checksum = findChecksum(SentMessage, k)

ReceiverChecksum = checkReceiverChecksum(ReceivedMessage, k, Checksum)

print("SENDER SIDE CHECKSUM: ", Checksum)

print("RECEIVER SIDE CHECKSUM: ", ReceiverChecksum)

if(int(ReceiverChecksum, 2) == 0):

print("Receiver Checksum is equal to 0. Therefore,")

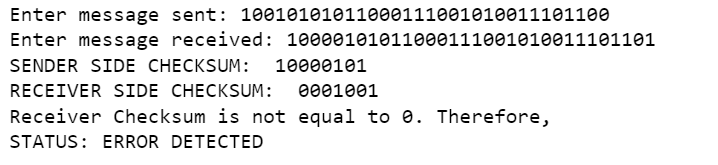
print("STATUS: ACCEPTED")

else:

print("Receiver Checksum is not equal to 0. Therefore,")

print("STATUS: ERROR DETECTED")

Output:



1. Implement caeser cipher substitution operation.

Code:

def encrypt(txt, key):

s = ""

for i in range(len(txt)):

ch = txt[i]

if ch.isupper():

ch = chr((ord(ch) + key - 65) % 26 + 65)

s += ch

else:

ch = chr((ord(ch) + key - 97) % 26 + 97)

s += ch

return(s)

def decrypt(txt, key):

s = ""

for i in range(len(txt)):

ch = txt[i]

if ch.isupper():

ch = chr((ord(ch) - key - 65) % 26 + 65)

s += ch

else:

ch = chr((ord(ch) - key - 97) % 26 + 97)

s += ch

return(s)

def main():

print("\t\t\t\*\*\*\*\*\*\*\*\*\*\*CAESER CIPHER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

print("Enter your choice\n")

print("1. Encryption\n")

print("2. Decryption\n")

choice = input()

msg = input("Enter your message")

key = ord(input("Enter key"))

if(choice == '1'):

print("Plain text: ", msg)

print("Cipher text: ", encrypt(msg, key))

elif(choice == "2"):

print("Cipher text: ", msg)

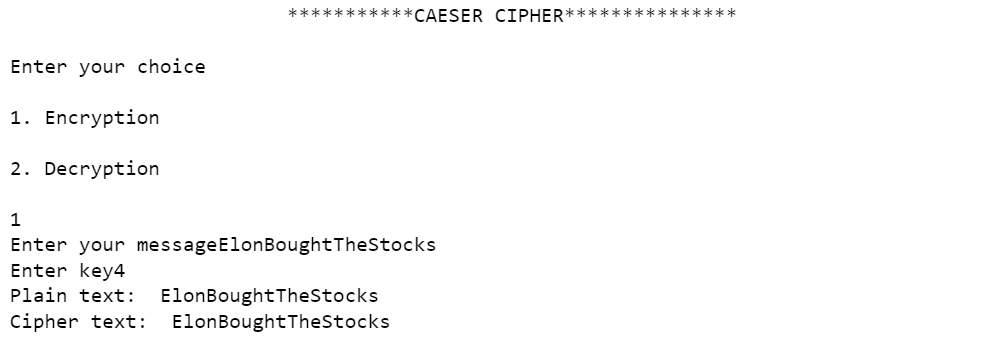
print("Plain text: ", decrypt(msg, key))

else:

print("Wrong choice \n TRY AGAIN")

main()

Output:



1. Implement monoalphabetic and polyalphabetic cipher substitution operation.

Code:

Monalphabetic (Multiplicative Cipher)

def encrypt(message,k):

cipher=""

for i in message:

if i.isupper():

cipher+=chr(((ord(i)-65)\*k)%26+65)

elif i.islower():

cipher+=chr(((ord(i)-97)\*k)%26+97)

else:

cipher+=" "

return cipher

def getCoeff(d):

for i in range(1,26):

j=int(1)

eqn=int(1)

while(eqn>=1):

eqn=26\*i-d\*j

if eqn==1:

return -j

j=j+1

def decrypt(cipher,k):

message=""

k=getCoeff(k)

for i in cipher:

if i.isupper():

message+=chr(((ord(i)-65)\*k)%26+65)

elif i.islower():

message+=chr(((ord(i)-97)\*k)%26+97)

else:

message+=" "

return message

print("Press \n 1. For encryption \n 2. For decryption")

choice=int(input("Enter your choice: "))

if choice==1:

key=int(input("Enter a valid numerical key:"))

plainText=input("Enter your message: ")

print("Cipher Text:",encrypt(plainText,key))

elif choice==2:

key=int(input("Enter a valid numerical key:"))

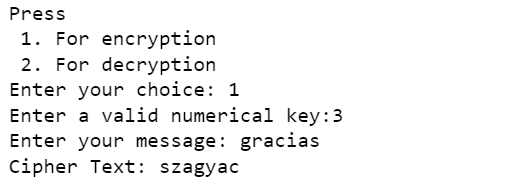
cipherText=input("Enter your message to be decrypted: ")

print("Plain Text: ",decrypt(cipherText,key))

else:

print("Invalid Choice!!")

Output:



Polyalphabetic (Vignere Cipher)

Code:

def encrypt(p, k):

Key\_length = len(k)

key\_as\_int = [ord(i) for i in k]

plaintext\_int = [ord(i) for i in p]

ciphertext = " "

for i in range(len(plaintext\_int)):

value = (plaintext\_int[i] + key\_as\_int[i % Key\_length]) % 26

ciphertext += chr(value +65)

return ciphertext

def decrypt(c, k):

Key\_length = len(k)

key\_as\_int = [ord(i) for i in k]

ciphertext\_int = [ord(i) for i in c]

plaintext = " "

for i in range(len(ciphertext\_int)):

value = (ciphertext\_int[i] - key\_as\_int[i % Key\_length]) % 26

plaintext += chr(value +65)

return plaintext

while True:

print("-------------------Welcome to vigenere cipher---------------.\n\n"

"The text message should contain only characters and the key should be one character \n"

"1. Enrypt a message \n2. Decrypt a message \n3. Exit\n " )

choose = input("Choice: ")

if choose == '1':

p = input("Enter the plain text: ")

p = p.replace(" ", "") # this will make sure that there is no space in the message

if (p.isupper()) == False:

p = p.upper()

if p.isalpha():

k = input("Enter the key: ")

k = k.strip() # remove the white spaces from both sides

if (k.isupper()) == False:

k = k.upper()

if k.isalpha():

c = encrypt(p, k)

print("The cipher text is: ", c)

else:

print(k)

print("Enter valid key, key is only one character word!")

else:

print("only letters are allowed !!")

elif choose == '2':

c = input("Enter the cipher text: ")

c = c.replace(" ", "")

if (c.isupper()) == False:

c = c.upper()

if c.isalpha():

k = input("Enter the key: ")

if (k.isupper()) == False:

k = k.upper()

if not k.isalpha():

print("Enter valid key, key is only one character word!")

else:

p = decrypt(c, k)

print("The plain text is: ", p)

else:

print("only letters are allowed!")

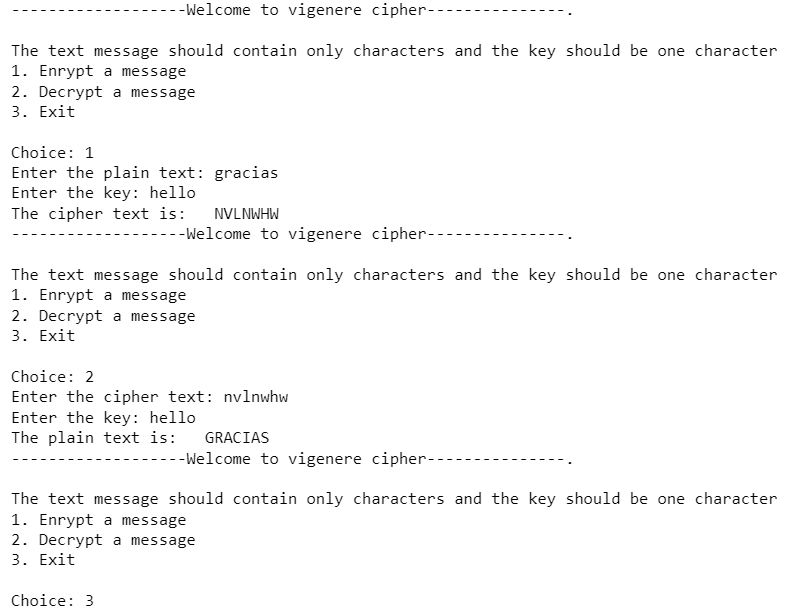
elif choose == '3':

break

else:

print("Please enter a valid choice!")

Output:



1. Implement playfair cipher substitution operation.

Code:

key=input("Enter key: ")

key=key.replace(" ", "")

key=key.upper()

def matrix(x,y,initial):

return [[initial for i in range(x)] for j in range(y)]

result=list()

for c in key: #storing key

if c not in result:

if c=='J':

result.append('I')

else:

result.append(c)

flag=0

for i in range(65,91): #storing other character

if chr(i) not in result:

if i==73 and chr(74) not in result:

result.append("I")

flag=1

elif flag==0 and i==73 or i==74:

pass

else:

result.append(chr(i))

k=0

my\_matrix=matrix(5,5,0) #initialize matrix

for i in range(0,5): #making matrix

for j in range(0,5):

my\_matrix[i][j]=result[k]

k+=1

def locindex(c): #get location of each character

loc=list()

if c=='J':

c='I'

for i ,j in enumerate(my\_matrix):

for k,l in enumerate(j):

if c==l:

loc.append(i)

loc.append(k)

return loc

def encrypt(): #Encryption

msg=str(input("ENTER MSG:"))

msg=msg.upper()

msg=msg.replace(" ", "")

i=0

for s in range(0,len(msg)+1,2):

if s<len(msg)-1:

if msg[s]==msg[s+1]:

msg=msg[:s+1]+'X'+msg[s+1:]

if len(msg)%2!=0:

msg=msg[:]+'X'

print("CIPHER TEXT:",end=' ')

while i<len(msg):

loc=list()

loc=locindex(msg[i])

loc1=list()

loc1=locindex(msg[i+1])

if loc[1]==loc1[1]:

print("{}{}".format(my\_matrix[(loc[0]+1)%5][loc[1]],my\_matrix[(loc1[0]+1)%5][loc1[1]]),end=' ')

elif loc[0]==loc1[0]:

print("{}{}".format(my\_matrix[loc[0]][(loc[1]+1)%5],my\_matrix[loc1[0]][(loc1[1]+1)%5]),end=' ')

else:

print("{}{}".format(my\_matrix[loc[0]][loc1[1]],my\_matrix[loc1[0]][loc[1]]),end=' ')

i=i+2

def decrypt(): #decryption

msg=str(input("ENTER CIPHER TEXT:"))

msg=msg.upper()

msg=msg.replace(" ", "")

print("PLAIN TEXT:",end=' ')

i=0

while i<len(msg):

loc=list()

loc=locindex(msg[i])

loc1=list()

loc1=locindex(msg[i+1])

if loc[1]==loc1[1]:

print("{}{}".format(my\_matrix[(loc[0]-1)%5][loc[1]],my\_matrix[(loc1[0]-1)%5][loc1[1]]),end=' ')

elif loc[0]==loc1[0]:

print("{}{}".format(my\_matrix[loc[0]][(loc[1]-1)%5],my\_matrix[loc1[0]][(loc1[1]-1)%5]),end=' ')

else:

print("{}{}".format(my\_matrix[loc[0]][loc1[1]],my\_matrix[loc1[0]][loc[1]]),end=' ')

i=i+2

while(1):

choice=int(input("\n 1.Encryption \n 2.Decryption: \n 3.EXIT"))

if choice==1:

encrypt()

elif choice==2:

decrypt()

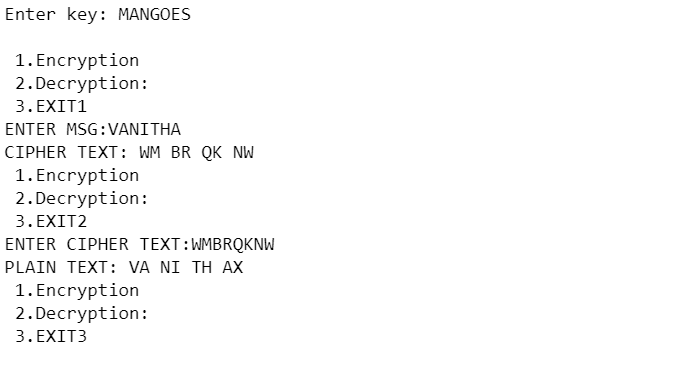
elif choice==3:

exit()

else:

print("Choose correct choice")

Output:



1. Implement hill cipher substitution operation.

Code:

import math

import string

import sys

import numpy as np

from sympy import Matrix

def menu():

while True:

print("---- Hill Cipher ----\n")

print("1) Encrypt a Message.")

print("2) Decipher a Message.")

print("3) Quit.\n")

try:

choice = int(input("Select a function to run: "))

if 1 <= choice <= 3:

return choice

else:

print("\nYou must enter a number from 1 to 3\n")

except ValueError:

print("\nYou must enter a number from 1 to 3\n")

input("Press Enter to continue.\n")

# Create two dictionaries, english alphabet to numbers and numbers to english alphabet, and returns them

def get\_alphabet():

alphabet = {}

for character in string.ascii\_uppercase:

alphabet[character] = string.ascii\_uppercase.index(character)

reverse\_alphabet = {}

for key, value in alphabet.items():

reverse\_alphabet[value] = key

return alphabet, reverse\_alphabet

# Get input from the user and checks if respects the alphabet

def get\_text\_input(message, alphabet):

while True:

text = input(message)

text = text.upper()

if all(keys in alphabet for keys in text):

return text

else:

print("\nThe text must contain only characters from the english alphabet ([A to Z] or [a to z]).")

# Check if the key is a square in length

def is\_square(key):

key\_length = len(key)

if 2 <= key\_length == int(math.sqrt(key\_length)) \*\* 2:

return True

else:

return False

# Create the matrix k for the key

def get\_key\_matrix(key, alphabet):

k = list(key)

m = int(math.sqrt(len(k)))

for (i, character) in enumerate(k):

k[i] = alphabet[character]

return np.reshape(k, (m, m))

# Create the matrix of m-grams of a text, if needed, complete the last m-gram with the last letter of the alphabet

def get\_text\_matrix(text, m, alphabet):

matrix = list(text)

remainder = len(text) % m

for (i, character) in enumerate(matrix):

matrix[i] = alphabet[character]

if remainder != 0:

for i in range(m - remainder):

matrix.append(25)

return np.reshape(matrix, (int(len(matrix) / m), m)).transpose()

# Encrypt a Message and returns the ciphertext matrix

def encrypt(key, plaintext, alphabet):

m = key.shape[0]

m\_grams = plaintext.shape[1]

# Encrypt the plaintext with the key provided k, calculate matrix c of ciphertext

ciphertext = np.zeros((m, m\_grams)).astype(int)

for i in range(m\_grams):

ciphertext[:, i] = np.reshape(np.dot(key, plaintext[:, i]) % len(alphabet), m)

return ciphertext

# Transform a matrix to a text, according to the alphabet

def matrix\_to\_text(matrix, order, alphabet):

if order == 't':

text\_array = np.ravel(matrix, order='F')

else:

text\_array = np.ravel(matrix)

text = ""

for i in range(len(text\_array)):

text = text + alphabet[text\_array[i]]

return text

# Check if the key is invertible and in that case returns the inverse of the matrix

def get\_inverse(matrix, alphabet):

alphabet\_len = len(alphabet)

if math.gcd(int(round(np.linalg.det(matrix))), alphabet\_len) == 1:

matrix = Matrix(matrix)

return np.matrix(matrix.inv\_mod(alphabet\_len))

else:

return None

# Decrypt a Message and returns the plaintext matrix

def decrypt(k\_inverse, c, alphabet):

return encrypt(k\_inverse, c, alphabet)

def get\_m():

while True:

try:

m = int(input("Insert the length of the grams (m): "))

if m >= 2:

return m

else:

print("\nYou must enter a number m >= 2\n")

except ValueError:

print("\nYou must enter a number m >= 2\n")

# Force a Ciphertext (Known Plaintext Attack)

def plaintext\_attack(c, p\_inverse, alphabet):

return encrypt(c, p\_inverse, alphabet)

def main():

while True:

# Ask the user what function wants to run

choice = menu()

# Get two dictionaries, english alphabet to numbers and numbers to english alphabet

alphabet, reverse\_alphabet = get\_alphabet()

# Run the function selected by the user

if choice == 1:

# Asks the user the plaintext and the key for the encryption and checks the input

plaintext = get\_text\_input("\nInsert the text to be encrypted: ", alphabet)

key = get\_text\_input("Insert the key for encryption: ", alphabet)

if is\_square(key):

# Get the key matrix k

k = get\_key\_matrix(key, alphabet)

print("\nKey Matrix:\n", k)

# Get the m-grams matrix p of the plaintext

p = get\_text\_matrix(plaintext, k.shape[0], alphabet)

print("Plaintext Matrix:\n", p)

input("\nPress Enter to begin te encryption.")

# Encrypt the plaintext

c = encrypt(k, p, alphabet)

# Transform the ciphertext matrix to a text of the alphabet

ciphertext = matrix\_to\_text(c, "t", reverse\_alphabet)

print("\nThe message has been encrypted.\n")

print("Generated Ciphertext: ", ciphertext)

print("Generated Ciphertext Matrix:\n", c, "\n")

else:

print("\nThe length of the key must be a square and >= 2.\n")

elif choice == 2:

# Asks the user the ciphertext and the key for the encryption and checks the input

ciphertext = get\_text\_input("\nInsert the ciphertext to be decrypted: ", alphabet)

key = get\_text\_input("Insert the key for decryption: ", alphabet)

if is\_square(key):

# Get the key matrix k

k = get\_key\_matrix(key, alphabet)

# Check if the key is invertible and in that case returns the inverse of the matrix

k\_inverse = get\_inverse(k, alphabet)

if k\_inverse is not None:

# Get the m-grams matrix c of the ciphertext

c = get\_text\_matrix(ciphertext, k\_inverse.shape[0], alphabet)

print("\nKey Matrix:\n", k)

print("Ciphertext Matrix:\n", c)

input("\nPress Enter to begin the decryption.")

# Decrypt the ciphertext

p = decrypt(k\_inverse, c, alphabet)

# Transform the ciphertext matrix to a text of the alphabet

plaintext = matrix\_to\_text(p, "t", reverse\_alphabet)

print("\nThe message has been decrypted.\n")

print("Generated Plaintext: ", plaintext)

print("Generated Plaintext Matrix:\n", p, "\n")

else:

print("\nThe matrix of the key provided is not invertible.\n")

else:

print("\nThe key must be a square and size >= 2.\n")

elif choice == 3:

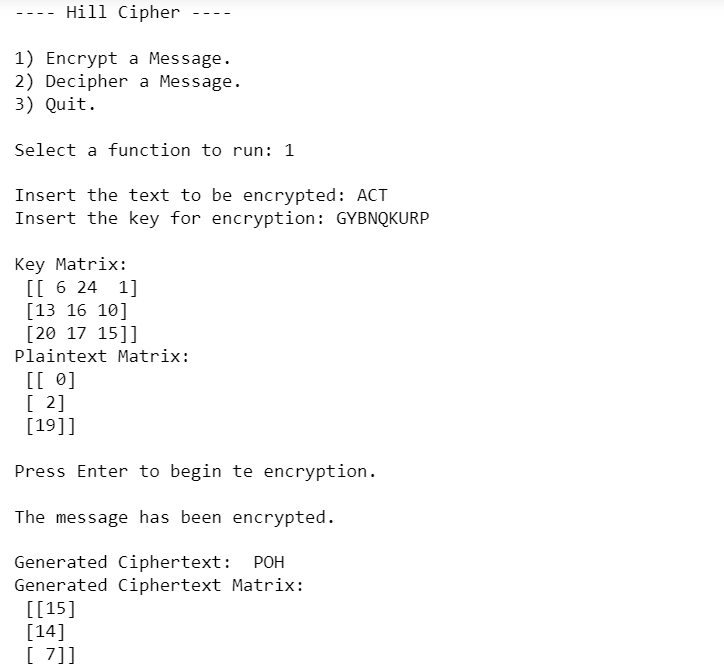
sys.exit(0)

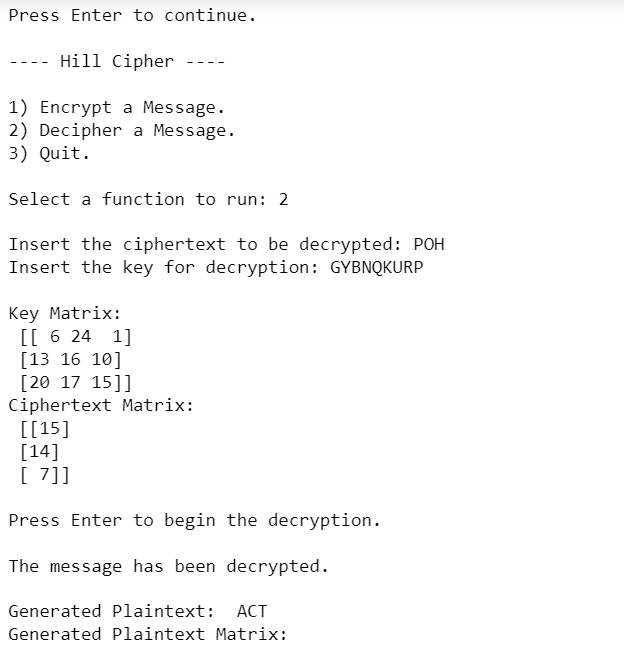
input("Press Enter to continue.\n")

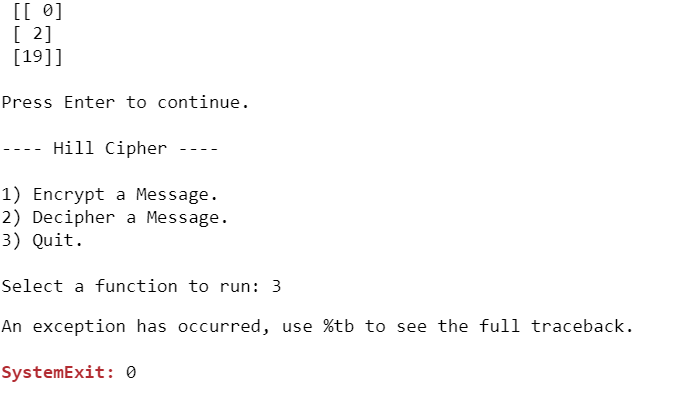
if \_\_name\_\_ == '\_\_main\_\_':

main()

Output:







1. Implement rail fence cipher transposition operation.

Code:

def encryptRailFence(text, key):

# create the matrix to cipher

# plain text key = rows ,

# length(text) = columns

# filling the rail matrix

# to distinguish filled

# spaces from blank ones

rail = [['\n' for i in range(len(text))]

for j in range(key)]

# to find the direction

dir\_down = False

row, col = 0, 0

for i in range(len(text)):

# check the direction of flow

# reverse the direction if we've just

# filled the top or bottom rail

if (row == 0) or (row == key - 1):

dir\_down = not dir\_down

# fill the corresponding alphabet

rail[row][col] = text[i]

col += 1

# find the next row using

# direction flag

if dir\_down:

row += 1

else:

row -= 1

# now we can construct the cipher

# using the rail matrix

result = []

for i in range(key):

for j in range(len(text)):

if rail[i][j] != '\n':

result.append(rail[i][j])

return("" . join(result))

# This function receives cipher-text

# and key and returns the original

# text after decryption

def decryptRailFence(cipher, key):

# create the matrix to cipher

# plain text key = rows ,

# length(text) = columns

# filling the rail matrix to

# distinguish filled spaces

# from blank ones

rail = [['\n' for i in range(len(cipher))]

for j in range(key)]

# to find the direction

dir\_down = None

row, col = 0, 0

# mark the places with '\*'

for i in range(len(cipher)):

if row == 0:

dir\_down = True

if row == key - 1:

dir\_down = False

# place the marker

rail[row][col] = '\*'

col += 1

# find the next row

# using direction flag

if dir\_down:

row += 1

else:

row -= 1

# now we can construct the

# fill the rail matrix

index = 0

for i in range(key):

for j in range(len(cipher)):

if ((rail[i][j] == '\*') and (index < len (cipher))):

rail[i][j] = cipher[index]

index += 1

# now read the matrix in

# zig-zag manner to construct

# the resultant text

result = []

row, col = 0, 0

for i in range(len(cipher)):

# check the direction of flow

if row == 0:

dir\_down = True

if row == key-1:

dir\_down = False

# place the marker

if (rail[row][col] != '\*'):

result.append(rail[row][col])

col += 1

# find the next row using

# direction flag

if dir\_down:

row += 1

else:

row -= 1

return("".join(result))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Reil Fence cipher\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

key= int(input("Enter your key"))

while(1):

choice=int(input("\n 1.Encryption \n 2.Decryption \n 3. Quit \n"))

if choice==1:

msg=input("Enter the message")

print(encryptRailFence(msg,key))

elif choice==2:

cipher=input("Enter the cipher text")

print(decryptRailFence(cipher,key))

elif choice==3:

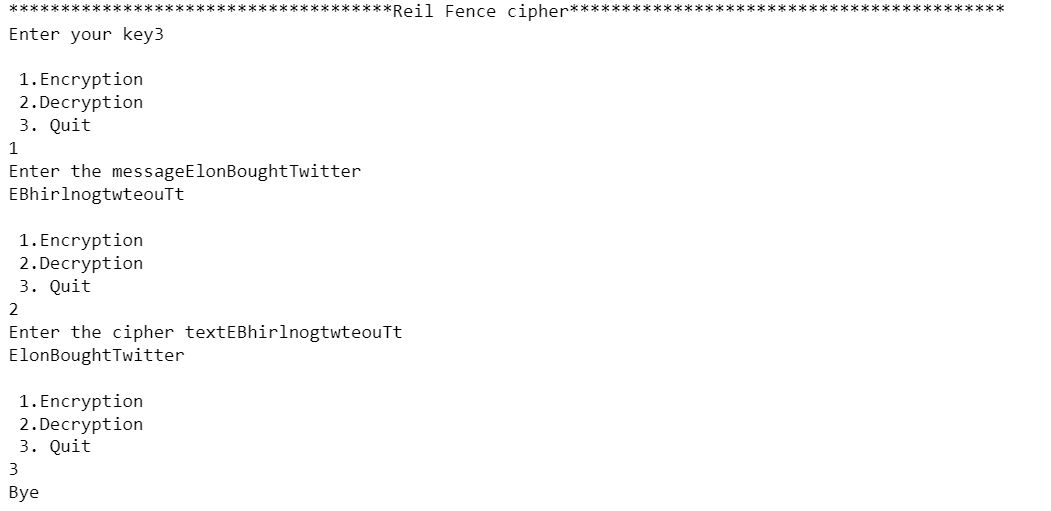
print("Bye")

exit()

else:

print("Choose correct choice")

Output:



1. Implement row transposition cipher transposition operation.

Code:

import math

key = input("enter key: ")

# Encryption

def encryptMessage(msg):

cipher = ""

# track key indices

k\_indx = 0

msg\_len = float(len(msg))

msg\_lst = list(msg)

key\_lst = sorted(list(key))

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# add the padding character '\_' in empty

# the empty cell of the matix

fill\_null = int((row \* col) - msg\_len)

msg\_lst.extend('\_' \* fill\_null)

# create Matrix and insert message and

# padding characters row-wise

matrix = [msg\_lst[i: i + col]

for i in range(0, len(msg\_lst), col)]

# read matrix column-wise using key

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

cipher += ''.join([row[curr\_idx]

for row in matrix])

k\_indx += 1

return cipher

# Decryption

def decryptMessage(cipher):

msg = ""

# track key indices

k\_indx = 0

# track msg indices

msg\_indx = 0

msg\_len = float(len(cipher))

msg\_lst = list(cipher)

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# convert key into list and sort

# alphabetically so we can access

# each character by its alphabetical position.

key\_lst = sorted(list(key))

# create an empty matrix to

# store deciphered message

dec\_cipher = []

for \_ in range(row):

dec\_cipher += [[None] \* col]

# Arrange the matrix column wise according

# to permutation order by adding into new matrix

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

for j in range(row):

dec\_cipher[j][curr\_idx] = msg\_lst[msg\_indx]

msg\_indx += 1

k\_indx += 1

# convert decrypted msg matrix into a string

try:

msg = ''.join(sum(dec\_cipher, []))

except TypeError:

raise TypeError("This program cannot",

"handle repeating words.")

null\_count = msg.count('\_')

if null\_count > 0:

return msg[: -null\_count]

return msg

# Driver Code

msg = input("Enter your message: ")

cipher = encryptMessage(msg)

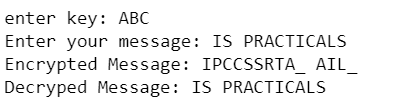
print("Encrypted Message: {}".

format(cipher))

print("Decryped Message: {}".

format(decryptMessage(cipher)))

Output:



1. Implement product cipher transposition operation.

Code:

k = [3,1,4,5,2]

ki = [2,5,1,3,4]

kc = 3

alpha = 'abcdefghijklmnopqrstuvwxyz'

print("-----------Product Cipher------------")

msg = input("Enter the message: ")

msg = "".join(msg.split())

enc = ""

dec = ""

while len(msg)%5 != 0 :

msg = msg + "x"

for i in msg :

enc = enc + alpha[(alpha.find(i)+kc)%26]

print("After encryption with Caesar Cipher:",enc)

msg = enc

enc = ""

mat = [["x" for i in range(5)] for j in range(int(len(msg)/5))]

print("Transposition Matrix: ")

for i in range(int(len(msg)/5)) :

for j in range(5):

print(msg[i\*5+j], end=" ")

print()

for i in range(5) :

for j in range(int(len(msg)/5)) :

if j\*5+k[i]-1 < len(msg) :

mat[j][i] = msg[j\*5+k[i]-1]

enc = ""

for i in range(5) :

for j in range(int(len(msg)/5)) :

enc = enc + mat[j][i]

print("Final Encrypted Message:",enc.upper())

for i in range(5) :

for j in range(int(len(enc)/5)) :

mat[j][i] = enc[i\*(int(len(enc)/5))+j]

enc= ""

for i in range(int(len(msg)/5)) :

for j in range(5) :

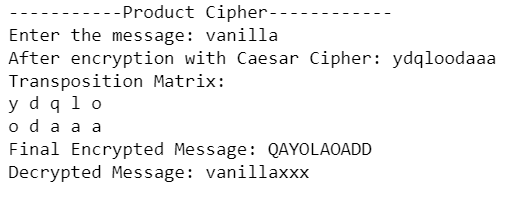
enc = enc + mat[i][ki[j]-1]

for i in enc :

dec = dec + alpha[(alpha.find(i)-kc)%26]

print("Decrypted Message:",dec)

Output:



11.Implement a stream cipher technique

Code:

def encrypt(p, k):

Key\_length = len(k)

key\_as\_int = [ord(i) for i in k]

plaintext\_int = [ord(i) for i in p]

ciphertext = " "

for i in range(len(plaintext\_int)):

value = (plaintext\_int[i] + key\_as\_int[i % Key\_length]) % 26

ciphertext += chr(value +65)

return ciphertext

def decrypt(c, k):

Key\_length = len(k)

key\_as\_int = [ord(i) for i in k]

ciphertext\_int = [ord(i) for i in c]

plaintext = " "

for i in range(len(ciphertext\_int)):

value = (ciphertext\_int[i] - key\_as\_int[i % Key\_length]) % 26

plaintext += chr(value +65)

return plaintext

while True:

print("-------------------Welcome to vigenere cipher---------------.\n\n"

"The text message should contain only characters and the key should be one character \n"

"1. Enrypt a message \n2. Decrypt a message \n3. Exit\n " )

choose = input("Choice: ")

if choose == '1':

p = input("Enter the plain text: ")

p = p.replace(" ", "") # this will make sure that there is no space in the message

if (p.isupper()) == False:

p = p.upper()

if p.isalpha():

k = input("Enter the key: ")

k = k.strip() # remove the white spaces from both sides

if (k.isupper()) == False:

k = k.upper()

if k.isalpha():

c = encrypt(p, k)

print("The cipher text is: ", c)

else:

print(k)

print("Enter valid key, key is only one character word!")

else:

print("only letters are allowed !!")

elif choose == '2':

c = input("Enter the cipher text: ")

c = c.replace(" ", "")

if (c.isupper()) == False:

c = c.upper()

if c.isalpha():

k = input("Enter the key: ")

if (k.isupper()) == False:

k = k.upper()

if not k.isalpha():

print("Enter valid key, key is only one character word!")

else:

p = decrypt(c, k)

print("The plain text is: ", p)

else:

print("only letters are allowed!")

elif choose == '3':

break

else:

print("Please enter a valid choice!")

Output:

