INDEX

S.No TITLE Pg. No

1. Caesar cipher 1

2. Monoalphabetic 2

3. Playfair 3

4. Polyalphabetic 4

5. Affine Caesar cipher 5

6. Cipher text 6

7. Simple substitution 7

8. Monoalphabetic Cipher 8

9. Playfair code 9

10. Playfair matrix 10

11. Unique keys 11

12. Hill cipher 12

13. Hill cipher 13

14. Vigenere cipher 14

15. Additive cipher 15

16. Monoalphabetic substitution 16

17. Des algorithm 17

18. Des algorithm 18

19. Cbc algorithm 19

20. Ebc mode 20

21. Counter mode 21

22. Cbc mode 22

23. Counter mode 23

24. Rsa system 24

25. Rsa algorithm 25

26. Rsa public-key encryption 26

27. Rsa cryptosystem 27

28. Diffie-hellman protocol 28

29. Sha-3 option 29

30. Cbc mac 30

31. Cmac 31

32. Dsa,Rsa difference 32

33. Des algorithm 33

34. Ecb,Cbc and Cfc modes 34

35. Vigenere cipher 35

36. Affine Caesar cipher 36

37. Monoalphabetic substitution 37

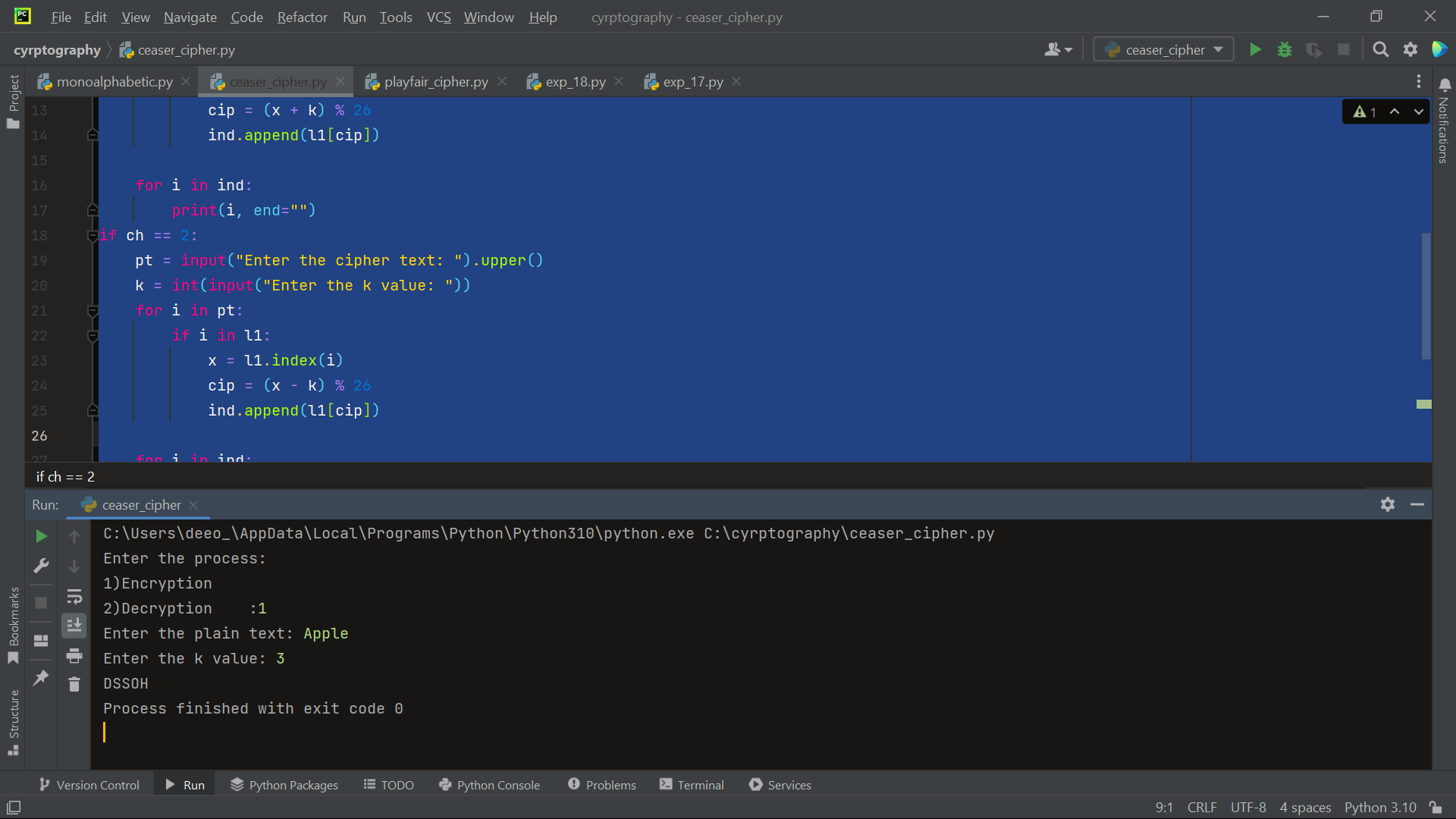
38. Hill cipher 38

39. Additive cipher 39

40. Monoalphabetic substitution 40

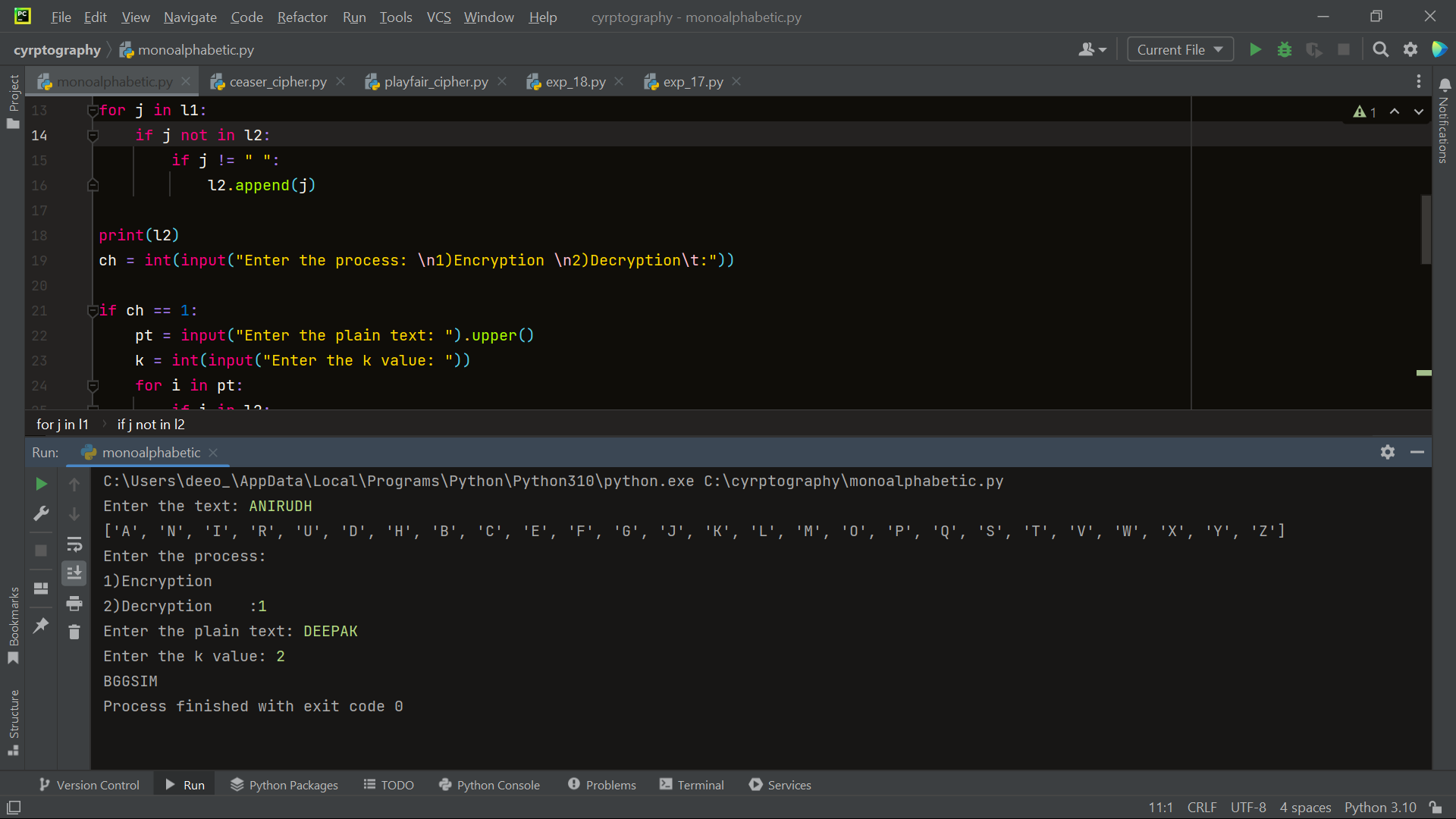
1. Write a Python program for Caesar cipher involves replacing each letter of the alphabet with the letter standing k places further down the alphabet, for k in the range 1 through 25.

l1 = ['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']  
ind = []  
  
ch = int(input("Enter the process: \n1)Encryption \n2)Decryption\t:"))  
  
if ch == 1:  
 pt = input("Enter the plain text: ").upper()  
 k = int(input("Enter the k value: "))  
 for i in pt:  
 if i in l1:  
 x = l1.index(i)  
 cip = (x + k) % 26  
 ind.append(l1[cip])  
  
 for i in ind:  
 print(i, end="")  
if ch == 2:  
 pt = input("Enter the cipher text: ").upper()  
 k = int(input("Enter the k value: "))  
 for i in pt:  
 if i in l1:  
 x = l1.index(i)  
 cip = (x - k) % 26  
 ind.append(l1[cip])  
  
 for i in ind:  
 print(i, end="")



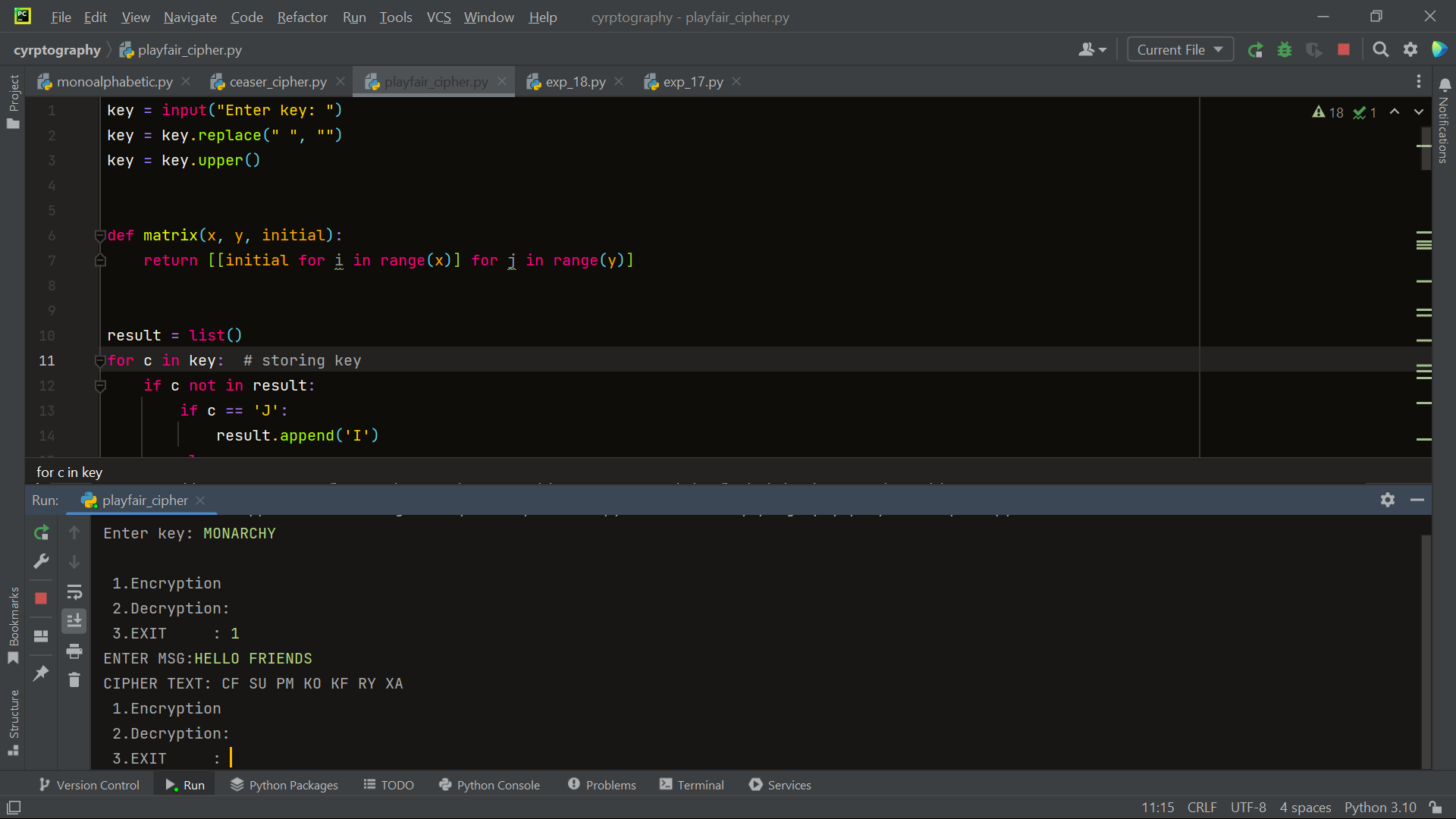
1. Write a Python program for monoalphabetic substitution cipher maps a plaintext alphabet to a ciphertext alphabet, so that each letter of the plaintext alphabet maps to a single unique letter of the ciphertext alphabet.

l1 = ['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']  
  
keyword = input("Enter the text: ").upper()  
ind = []  
  
l2 = []  
for i in keyword:  
 if i not in l2:  
 if i != " ":  
 l2.append(i)  
  
for j in l1:  
 if j not in l2:  
 if j != " ":  
 l2.append(j)  
  
print(l2)  
ch = int(input("Enter the process: \n1)Encryption \n2)Decryption\t:"))  
  
if ch == 1:  
 pt = input("Enter the plain text: ").upper()  
 k = int(input("Enter the k value: "))  
 for i in pt:  
 if i in l2:  
 x = l2.index(i)  
 cip = (x + k) % 26  
 ind.append(l2[cip])  
  
 for i in ind:  
 print(i, end="")  
if ch == 2:  
 pt = input("Enter the cipher text: ").upper()  
 k = int(input("Enter the k value: "))  
 for i in pt:  
 if i in l2:  
 x = l2.index(i)  
 cip = (x - k) % 26  
 ind.append(l2[cip])  
  
 for i in ind:  
 print(i, end="")



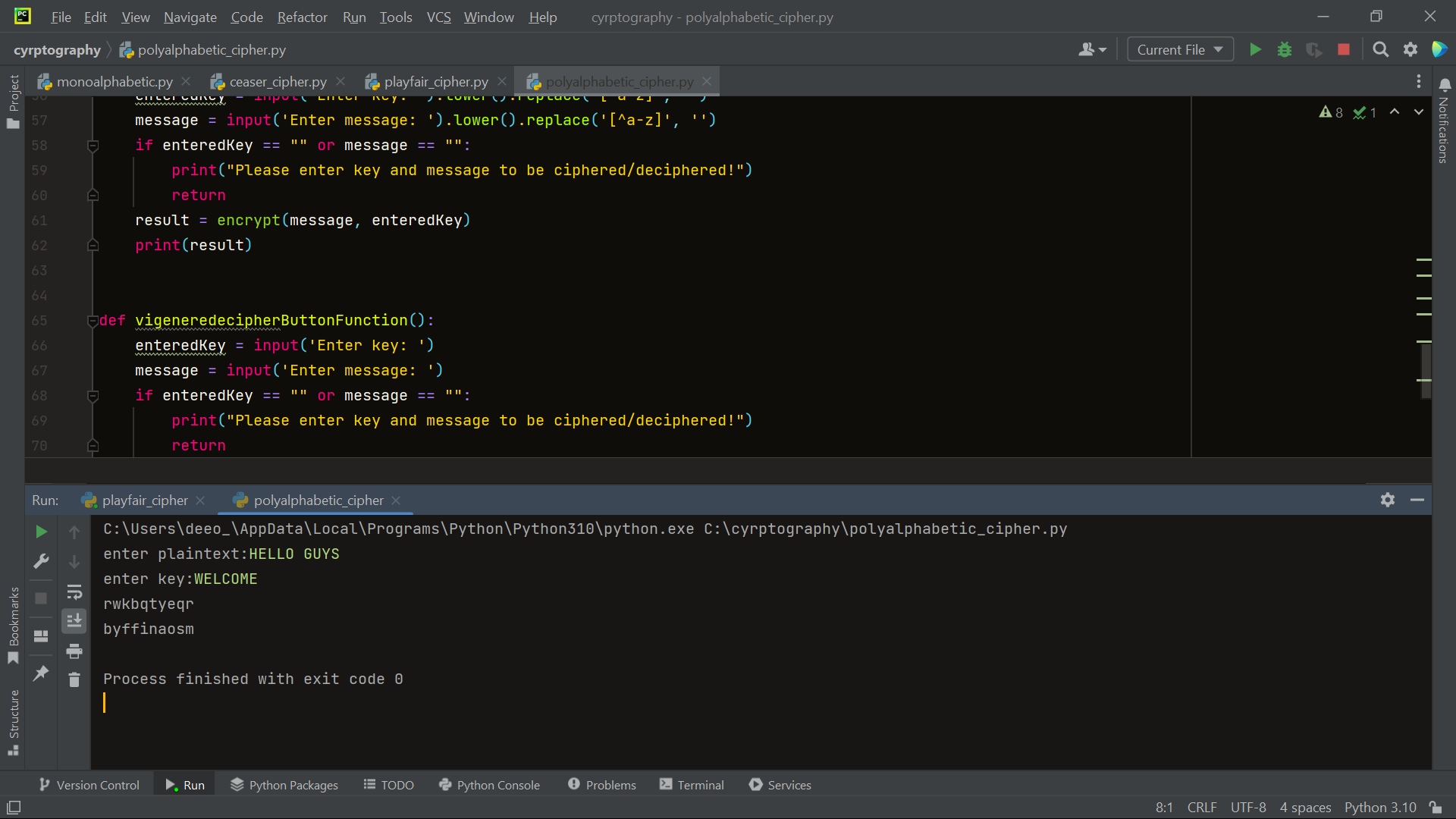
1. Write a Python program for Playfair algorithm is based on the use of a 5 X 5 matrix of letters constructed using a keyword. Plaintext is encrypted two letters at a time using this matrix.

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 \t: "))  
 if choice == 1:  
 encrypt()  
 elif choice == 2:  
 decrypt()  
 elif choice == 3:  
 exit()  
 else:  
 print("Choose correct choice")



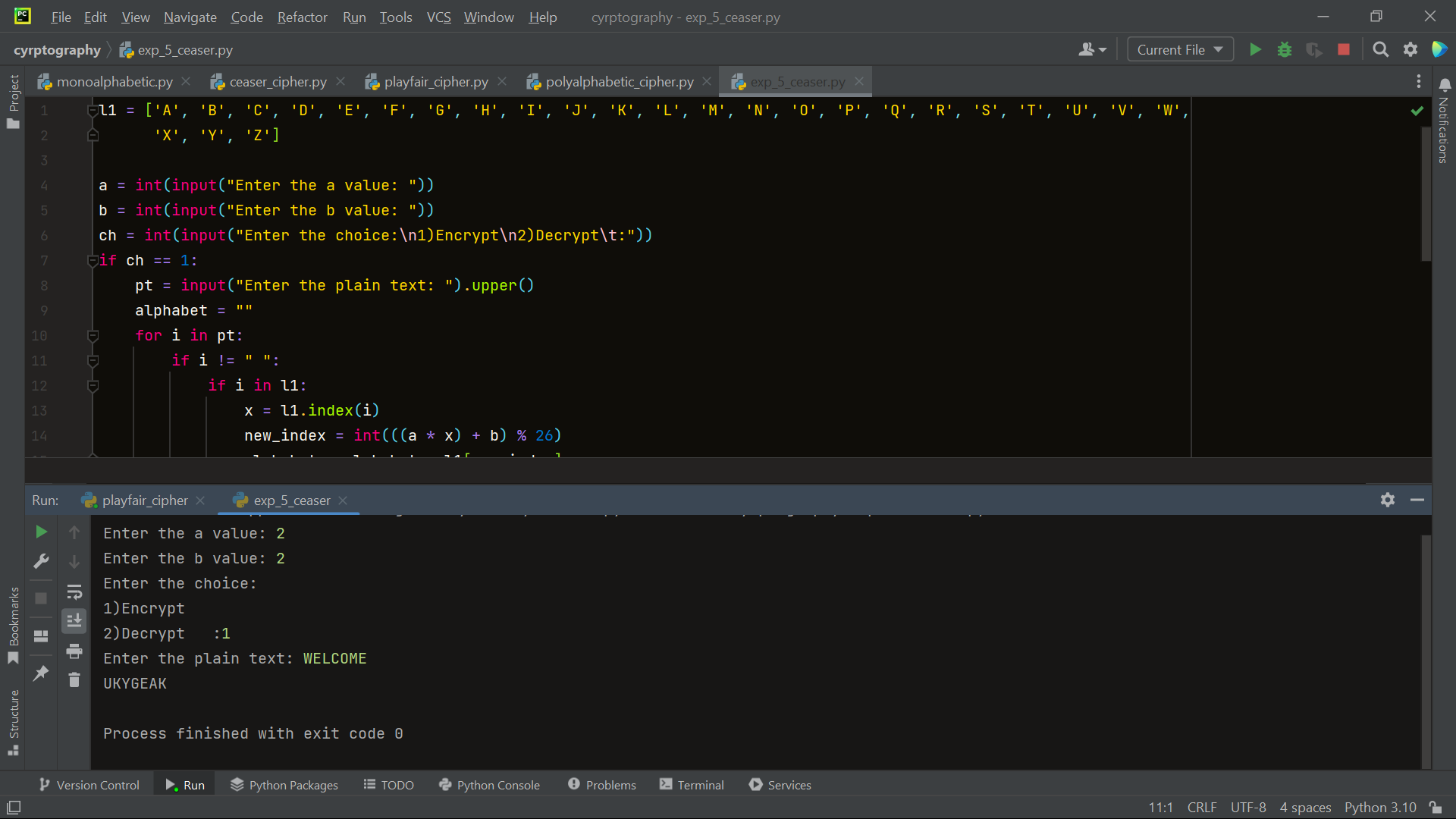
1. Write a Python program for polyalphabetic substitution cipher uses a separate monoalphabetic substitution cipher for each successive letter of plaintext, depending on a key

def encrypt(plaintext, k):  
 if len(k) <= 1:  
 print("keyword should be at least 2 characters long")  
 return  
 ciphertext = ""  
 for i in range(len(plaintext)):  
 ciphertext += chr((((ord(plaintext[i]) - 97) + (ord(k[i % len(k)]) - 97) + 26) % 26) + 97)  
 return ciphertext  
  
  
def decrypt(ciphertext, k):  
 if len(k) <= 1:  
 print("keyword should be at least 2 characters long")  
 return  
 plaintext = ""  
 for i in range(len(ciphertext)):  
 plaintext += chr((((ord(ciphertext[i]) - 97) - (ord(k[i % len(k)]) - 97) + 26) % 26) + 97)  
 return plaintext  
  
  
def vigenerecipherButtonFunction():  
 enteredKey = input('Enter key: ').lower().replace('[^a-z]', '')  
 message = input('Enter message: ').lower().replace('[^a-z]', '')  
 if enteredKey == "" or message == "":  
 print("Please enter key and message to be ciphered/deciphered!")  
 return  
 result = encrypt(message, enteredKey)  
 print(result)  
  
  
def vigeneredecipherButtonFunction():  
 enteredKey = input('Enter key: ')  
 message = input('Enter message: ')  
 if enteredKey == "" or message == "":  
 print("Please enter key and message to be ciphered/deciphered!")  
 return  
 result = decrypt(message, enteredKey)  
 print(result)  
  
  
plaintext = input("enter plaintext:")  
k = input("enter key:")  
print(encrypt(plaintext, k))  
ciphertext = encrypt(plaintext, k)  
print(decrypt(ciphertext, k))



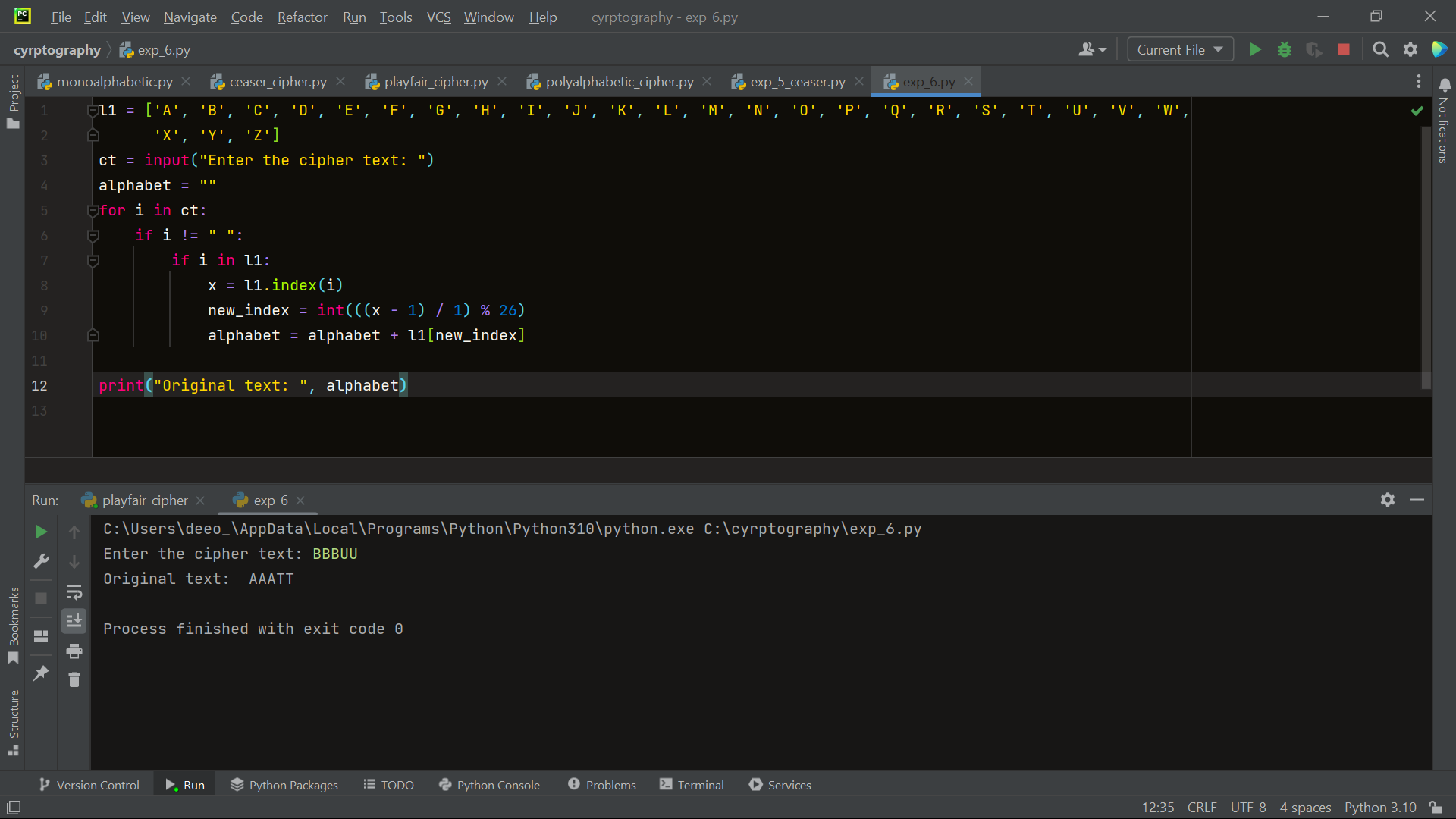
1. Write a Python program for generalization of the Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3. a. Are there any limitations on the value of b? b. Determine which values of a are not allowed.

l1 = ['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']  
  
a = int(input("Enter the a value: "))  
b = int(input("Enter the b value: "))  
ch = int(input("Enter the choice:\n1)Encrypt\n2)Decrypt\t:"))  
if ch == 1:  
 pt = input("Enter the plain text: ").upper()  
 alphabet = ""  
 for i in pt:  
 if i != " ":  
 if i in l1:  
 x = l1.index(i)  
 new\_index = int(((a \* x) + b) % 26)  
 alphabet = alphabet + l1[new\_index]  
  
 print(alphabet)  
  
if ch == 2:  
 ct = input("Enter the cipher text: ").upper()  
 alphabet = ""  
 for i in ct:  
 if i != " ":  
 if i in l1:  
 x = l1.index(i)  
 new\_index = int(((x - b) / a) % 26)  
 alphabet = alphabet + l1[new\_index]  
  
 print(alphabet)



1. Write a Python program for ciphertext has been generated with an affine cipher. The most frequent letter of the ciphertext is “B,” and the second most frequent letter of the ciphertext is “U.” Break this code.

l1 = ['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']  
ct = input("Enter the cipher text: ")  
alphabet = ""  
for i in ct:  
 if i != " ":  
 if i in l1:  
 x = l1.index(i)  
 new\_index = int(((x - 1) / 1) % 26)  
 alphabet = alphabet + l1[new\_index]  
  
print("Original text: ", alphabet)



1. Write a Python program for the following ciphertext was generated using a simple substitution algorithm. 53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\* ;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81 (‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;

Decrypt this message.

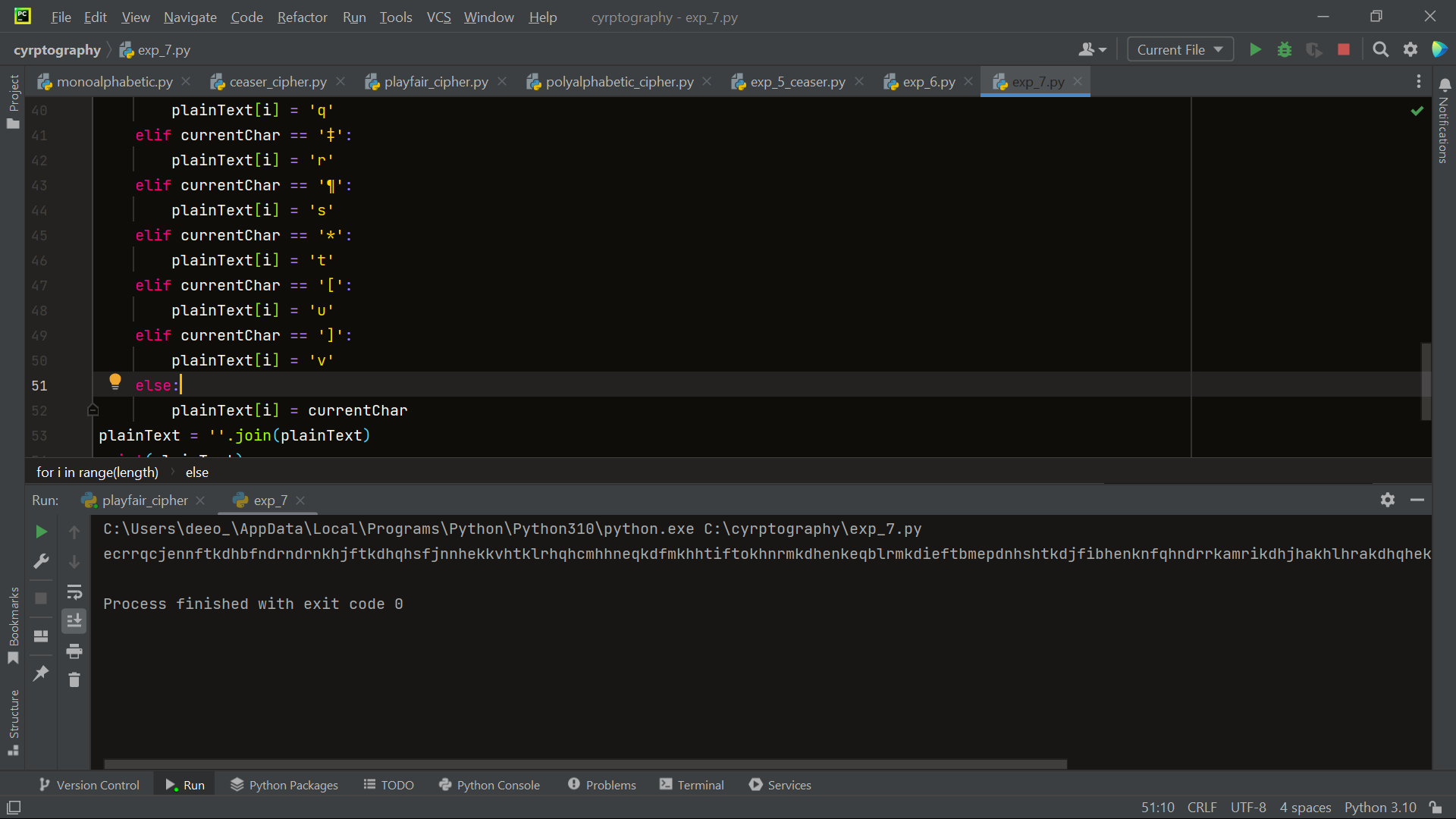
1. As you know, the most frequently occurring letter in English is e. Therefore, the first or second (or perhaps third?) most common character in the message is likely to stand for e. Also, e is often seen in pairs (e.g., meet, fleet, speed, seen, been, agree, etc.). Try to find a character in the ciphertext that

decodes to e.

2. The most common word in English is “the.” Use this fact to guess the characters that stand for t and h.

3. Decipher the rest of the message by deducing additional words.

cipherText = "53‡‡†305))6\*;4826)4‡)4‡);806\*;48†8¶60))85;;]8\*;:‡8†83(88)5†;46(;88\*96\*?;8)‡(;485);5†2:‡(;4956\*2(" \  
 "5—4)8¶8\*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81(‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;"  
length = len(cipherText)  
plainText = [''] \* length  
for i in range(length):  
 currentChar = cipherText[i]  
 if currentChar == '1':  
 plainText[i] = 'a'  
 elif currentChar == '2':  
 plainText[i] = 'b'  
 elif currentChar == '3':  
 plainText[i] = 'c'  
 elif currentChar == '4':  
 plainText[i] = 'd'  
 elif currentChar == '5':  
 plainText[i] = 'e'  
 elif currentChar == '6':  
 plainText[i] = 'f'  
 elif currentChar == '7':  
 plainText[i] = 'g'  
 elif currentChar == '8':  
 plainText[i] = 'h'  
 elif currentChar == '9':  
 plainText[i] = 'i'  
 elif currentChar == '0':  
 plainText[i] = 'j'  
 elif currentChar == ';':  
 plainText[i] = 'k'  
 elif currentChar == ':':  
 plainText[i] = 'l'  
 elif currentChar == '(':  
 plainText[i] = 'm'  
 elif currentChar == ')':  
 plainText[i] = 'n'  
 elif currentChar == '?':  
 plainText[i] = 'o'  
 elif currentChar == '—':  
 plainText[i] = 'p'  
 elif currentChar == '†':  
 plainText[i] = 'q'  
 elif currentChar == '‡':  
 plainText[i] = 'r'  
 elif currentChar == '¶':  
 plainText[i] = 's'  
 elif currentChar == '\*':  
 plainText[i] = 't'  
 elif currentChar == '[':  
 plainText[i] = 'u'  
 elif currentChar == ']':  
 plainText[i] = 'v'  
 else:  
 plainText[i] = currentChar  
plainText = ''.join(plainText)  
print(plainText)



1. Write a Python program for monoalphabetic cipher is that both sender and receiver must commit the permuted cipher sequence to memory. A common technique for avoiding this is to use a keyword from which the cipher sequence can be generated. For example, using the keyword CIPHER, write out the keyword followed by unused letters in normal order and match this against the plaintext letters: plain: 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 cipher:

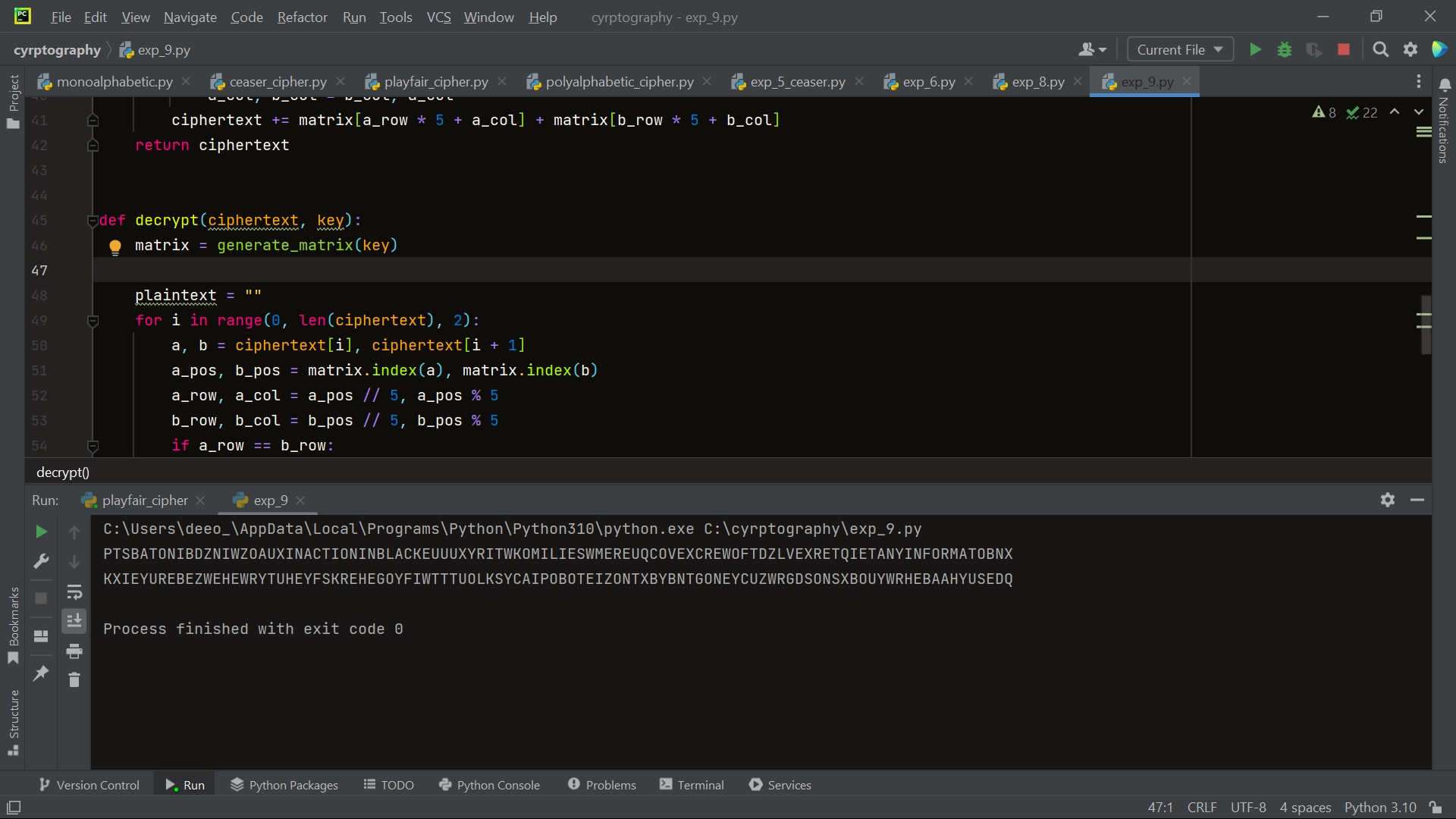
C I P H E R A B D F G J K L M N O Q S T U V W X Y Z

import random  
  
plaintext\_alphabet = "abcdefghijklmnopqrstuvwxyz"  
ciphertext\_alphabet = "CIPHERABDFGJKLNOQSTUVWXYZ"  
  
cipher\_dict = dict(zip(plaintext\_alphabet, ciphertext\_alphabet))  
  
message = input("enter the message: ")  
  
encrypted\_message = ""  
  
for ch in message:  
 if ch.isalpha():  
  
 if ch.islower():  
 encrypted\_ch = cipher\_dict[ch]  
 else:  
 encrypted\_ch = cipher\_dict[ch.lower()].upper()  
  
 else:  
 encrypted\_ch = ch  
  
 encrypted\_message += encrypted\_ch  
  
print("Encrypted message:", encrypted\_message)  
print("Cipher key:", cipher\_dict)



9. Write a C program for PT-109 American patrol boat, under the command of Lieutenant John F. Kennedy, was sunk by a Japanese destroyer, a message was received at an Australian wireless station in Playfair code: KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBNT GONEY CUZWR GDSON SXBOU YWRHE BAAHY USEDQ

def generate\_matrix(key):  
 key = key.replace(" ", "").upper()  
 key = "".join(sorted(set(key), key=key.index))  
 alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  
 matrix = []  
 for letter in key + alphabet:  
 if letter not in matrix:  
 matrix.append(letter)  
 return matrix  
  
  
def prepare\_text(text):  
 text = "".join(filter(str.isalpha, text)).upper()  
  
 text = text.replace("J", "I")  
  
 if len(text) % 2 != 0:  
 text += "X"  
 return text  
  
  
def encrypt(text, key):  
 matrix = generate\_matrix(key)  
  
 text = prepare\_text(text)  
  
 ciphertext = ""  
 for i in range(0, len(text), 2):  
 a, b = text[i], text[i + 1]  
 a\_pos, b\_pos = matrix.index(a), matrix.index(b)  
 a\_row, a\_col = a\_pos // 5, a\_pos % 5  
 b\_row, b\_col = b\_pos // 5, b\_pos % 5  
 if a\_row == b\_row:  
 a\_col = (a\_col + 1) % 5  
 b\_col = (b\_col + 1) % 5  
 elif a\_col == b\_col:  
 a\_row = (a\_row + 1) % 5  
 b\_row = (b\_row + 1) % 5  
 else:  
 a\_col, b\_col = b\_col, a\_col  
 ciphertext += matrix[a\_row \* 5 + a\_col] + matrix[b\_row \* 5 + b\_col]  
 return ciphertext  
  
  
def decrypt(ciphertext, key):  
 matrix = generate\_matrix(key)  
  
 plaintext = ""  
 for i in range(0, len(ciphertext), 2):  
 a, b = ciphertext[i], ciphertext[i + 1]  
 a\_pos, b\_pos = matrix.index(a), matrix.index(b)  
 a\_row, a\_col = a\_pos // 5, a\_pos % 5  
 b\_row, b\_col = b\_pos // 5, b\_pos % 5  
 if a\_row == b\_row:  
 a\_col = (a\_col - 1) % 5  
 b\_col = (b\_col - 1) % 5  
 elif a\_col == b\_col:  
 a\_row = (a\_row - 1) % 5  
 b\_row = (b\_row - 1) % 5  
 else:  
 a\_col, b\_col = b\_col, a\_col  
 plaintext += matrix[a\_row \* 5 + a\_col] + matrix[b\_row \* 5 + b\_col]  
 return plaintext  
  
  
plaintext = "KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBNT GONEY CUZWR GDSON SXBOU " \  
 "YWRHE BAAHY USEDQ"  
key = "royal new zeland navy"  
ciphertext = encrypt(plaintext, key)  
print(ciphertext)  
decrypted\_plaintext = decrypt(ciphertext, key)  
print(decrypted\_plaintext)



1. Write a Python program for Playfair matrix:

M F H I/J K

U N O P Q

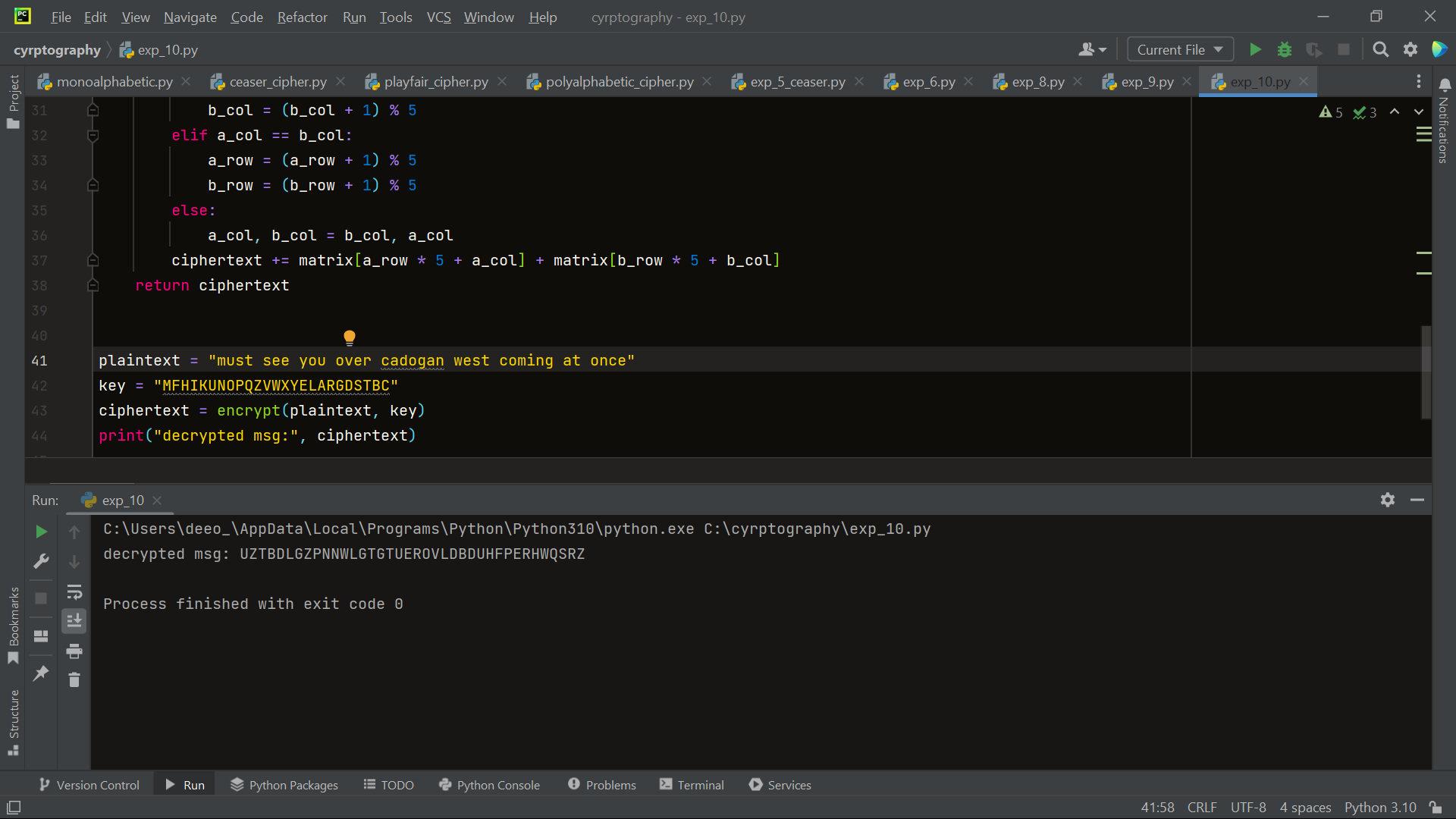
Z V W X Y

E L A R G

D S T B C

Encrypt this message: Must see you over Cadogan West. Coming at once.

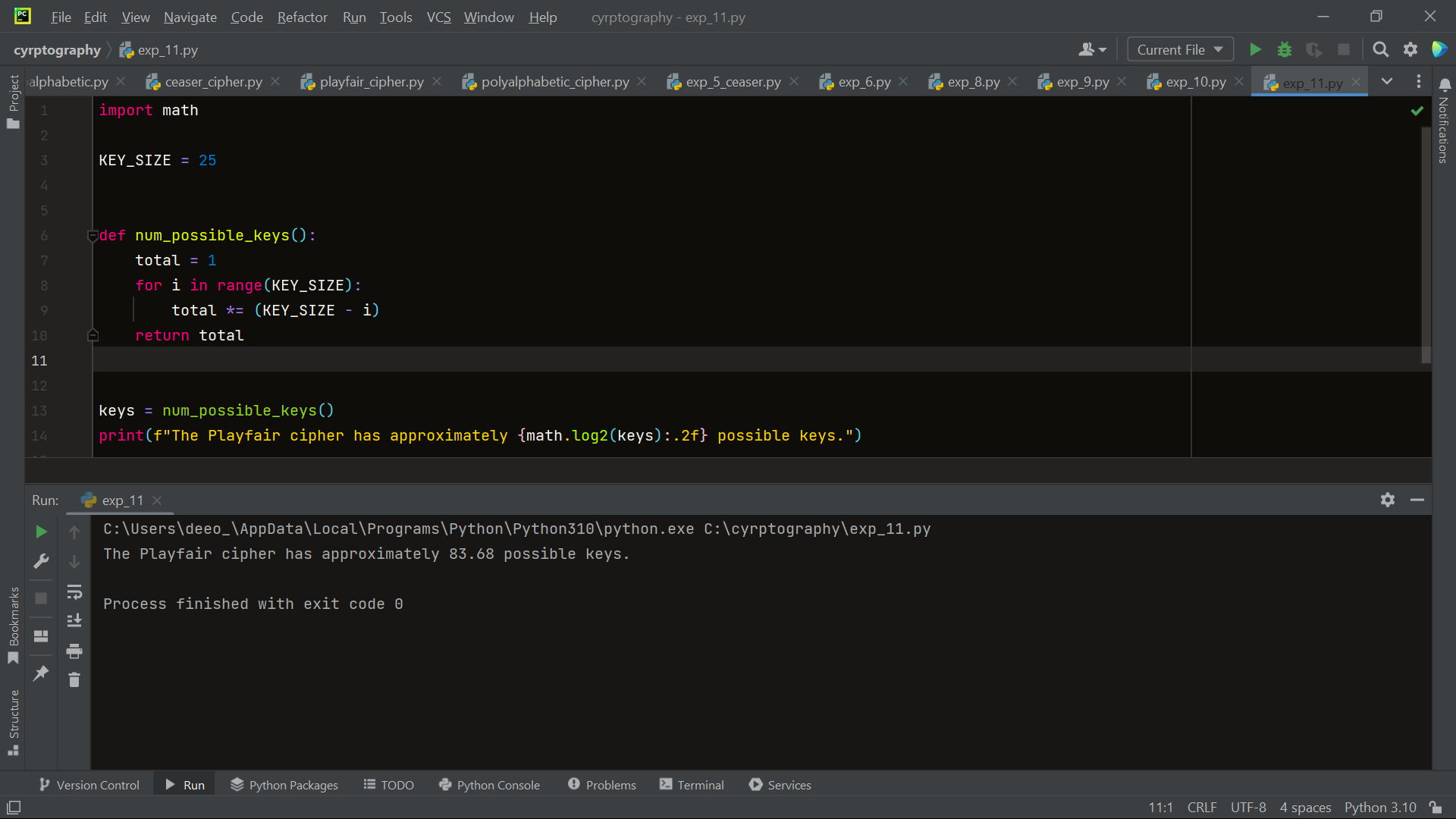
def generate\_matrix(key):  
 key = key.replace(" ", "").upper()  
 key = "".join(sorted(set(key), key=key.index))  
 alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  
 matrix = []  
 for letter in key + alphabet:  
 if letter not in matrix:  
 matrix.append(letter)  
 return matrix  
  
  
def prepare\_text(text):  
 text = "".join(filter(str.isalpha, text)).upper()  
 text = text.replace("J", "I")  
 if len(text) % 2 != 0:  
 text += "X"  
 return text  
  
  
def encrypt(text, key):  
 matrix = generate\_matrix(key)  
 text = prepare\_text(text)  
 ciphertext = ""  
 for i in range(0, len(text), 2):  
 a, b = text[i], text[i + 1]  
 a\_pos, b\_pos = matrix.index(a), matrix.index(b)  
 a\_row, a\_col = a\_pos // 5, a\_pos % 5  
 b\_row, b\_col = b\_pos // 5, b\_pos % 5  
 if a\_row == b\_row:  
 a\_col = (a\_col + 1) % 5  
 b\_col = (b\_col + 1) % 5  
 elif a\_col == b\_col:  
 a\_row = (a\_row + 1) % 5  
 b\_row = (b\_row + 1) % 5  
 else:  
 a\_col, b\_col = b\_col, a\_col  
 ciphertext += matrix[a\_row \* 5 + a\_col] + matrix[b\_row \* 5 + b\_col]  
 return ciphertext  
  
  
plaintext = "must see you over cadogan west coming at once"  
key = "MFHIKUNOPQZVWXYELARGDSTBC"  
ciphertext = encrypt(plaintext, key)  
print("decrypted msg:", ciphertext)



1. Write a Python program for possible keys does the Playfair cipher have? Ignore the fact that some keys might produce identical encryption results. Express your answer as an approximate power of 2.

a. Now take into account the fact that some Playfair keys produce the same encryption results. How many effectively unique keys does the Playfair cipher have?

import math  
  
KEY\_SIZE = 25  
  
  
def num\_possible\_keys():  
 total = 1  
 for i in range(KEY\_SIZE):  
 total \*= (KEY\_SIZE - i)  
 return total  
  
  
keys = num\_possible\_keys()  
print(f"The Playfair cipher has approximately {math.log2(keys):.2f} possible keys.")



1. a. Write a Python program to Encrypt the message “meet me at the usual place at ten rather than eight o’ clock” using the Hill cipher with the key

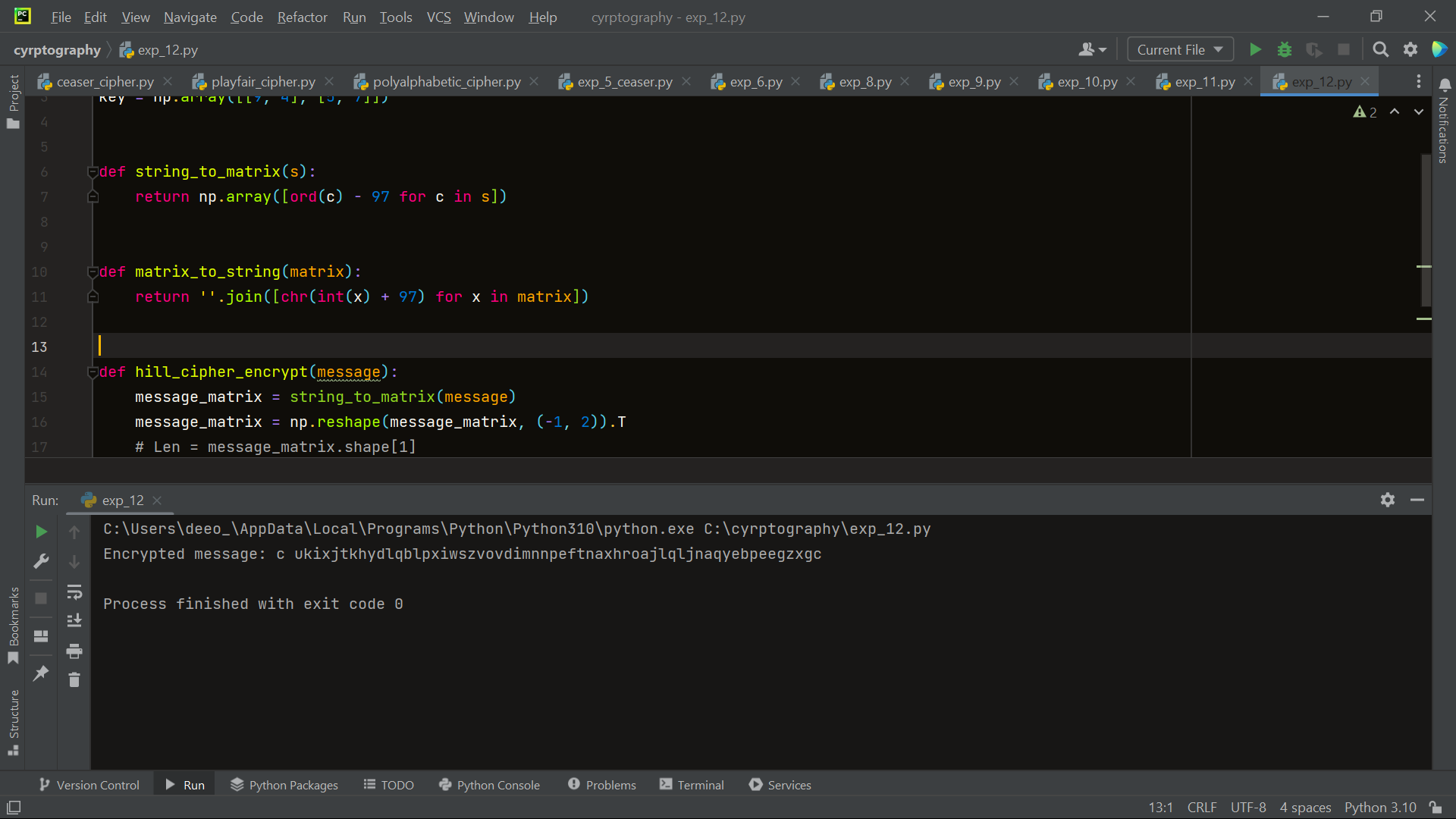
[9 4

5 7]

a. Show your calculations and the result.

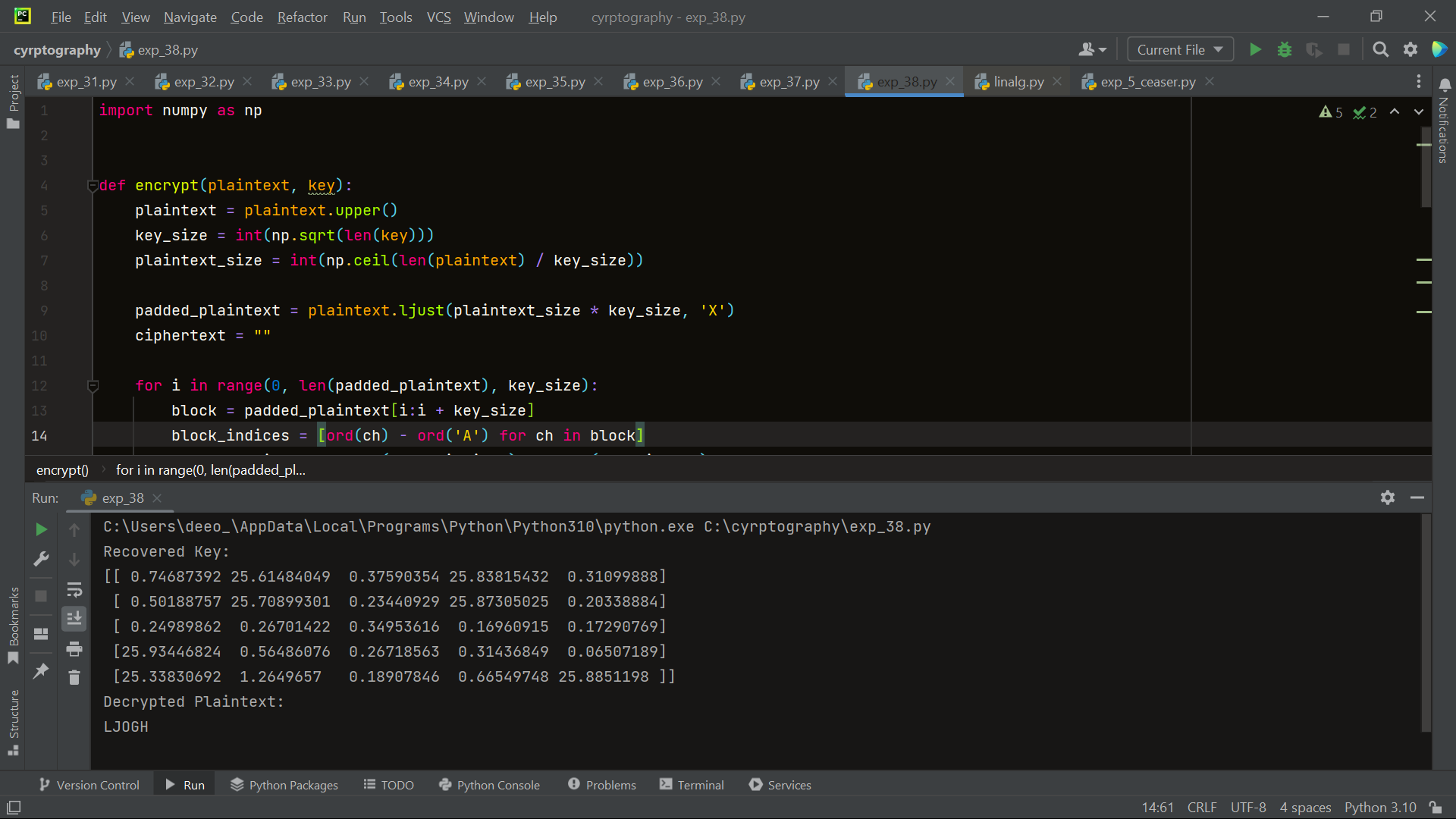
b. Show the calculations for the corresponding decryption of the ciphertext to recover the original plaintext.

import numpy as np  
  
key = np.array([[9, 4], [5, 7]])  
  
  
def string\_to\_matrix(s):  
 return np.array([ord(c) - 97 for c in s])  
  
  
def matrix\_to\_string(matrix):  
 return ''.join([chr(int(x) + 97) for x in matrix])  
  
  
def hill\_cipher\_encrypt(message):  
 message\_matrix = string\_to\_matrix(message)  
 message\_matrix = np.reshape(message\_matrix, (-1, 2)).T  
 # Len = message\_matrix.shape[1]  
 encrypted\_matrix = np.mod(np.dot(key, message\_matrix), 26)  
 encrypted\_message = matrix\_to\_string(encrypted\_matrix.T.flatten())  
 return encrypted\_message  
  
  
message = "meet me at the usual place at ten rather than eight oclock"  
encrypted\_message = hill\_cipher\_encrypt(message)  
print("Encrypted message: c", encrypted\_message)



1. Write a Python program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext– ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted.

import numpy as np  
  
  
def encrypt(plaintext, key):  
 plaintext = plaintext.upper()  
 key\_size = int(np.sqrt(len(key)))  
 plaintext\_size = int(np.ceil(len(plaintext) / key\_size))  
  
 padded\_plaintext = plaintext.ljust(plaintext\_size \* key\_size, 'X')  
 ciphertext = ""  
  
 for i in range(0, len(padded\_plaintext), key\_size):  
 block = padded\_plaintext[i:i + key\_size]  
 block\_indices = [ord(ch) - ord('A') for ch in block]  
 block\_matrix = np.array(block\_indices).reshape(key\_size, 1)  
  
 encrypted\_block\_matrix = np.dot(key, block\_matrix) % 26  
 encrypted\_block = "".join(chr(index + ord('A')) for index in encrypted\_block\_matrix.flatten())  
 ciphertext += encrypted\_block  
  
 return ciphertext  
  
  
def known\_plaintext\_attack(plaintexts, ciphertexts):  
 plaintext\_matrix = np.array([list(map(lambda x: ord(x) - ord('A'), plaintext)) for plaintext in plaintexts])  
 ciphertext\_matrix = np.array([list(map(lambda x: ord(x) - ord('A'), ciphertext)) for ciphertext in ciphertexts])  
  
 key = np.dot(np.linalg.pinv(plaintext\_matrix), ciphertext\_matrix) % 26  
  
 return key  
  
  
def decrypt(ciphertext, key):  
 ciphertext\_matrix = np.array(list(map(lambda x: ord(x) - ord('A'), ciphertext)))  
 decrypted\_matrix = np.dot(ciphertext\_matrix, key) % 26  
 decrypted\_text = ''.join(chr(int(val) + ord('A')) for val in decrypted\_matrix)  
 return decrypted\_text  
  
  
# Example usage  
plaintexts = ["HELLO", "WORLD"]  
ciphertexts = ["AXNNE", "ZCVDN"]  
key = known\_plaintext\_attack(plaintexts, ciphertexts)  
  
print("Recovered Key:")  
print(key)  
  
# Test the decryption with the recovered key  
recovered\_plaintext = decrypt(ciphertexts[0], key)  
print("Decrypted Plaintext:")  
print(recovered\_plaintext)

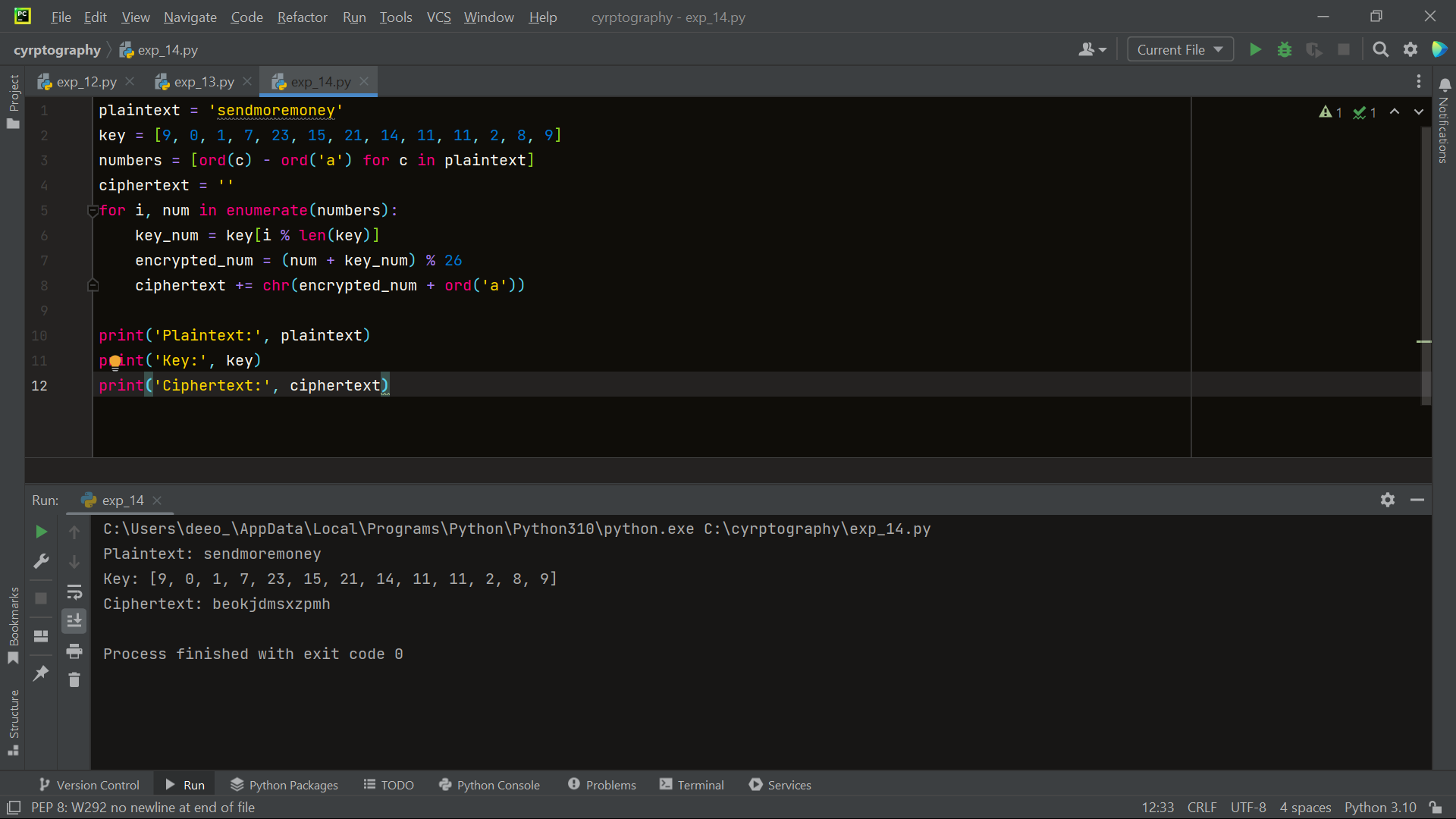


1. Write a Python program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.

a. Encrypt the plaintext send more money with the key stream 9 0 1 7 23 15 21 14 11 11 2 8 9

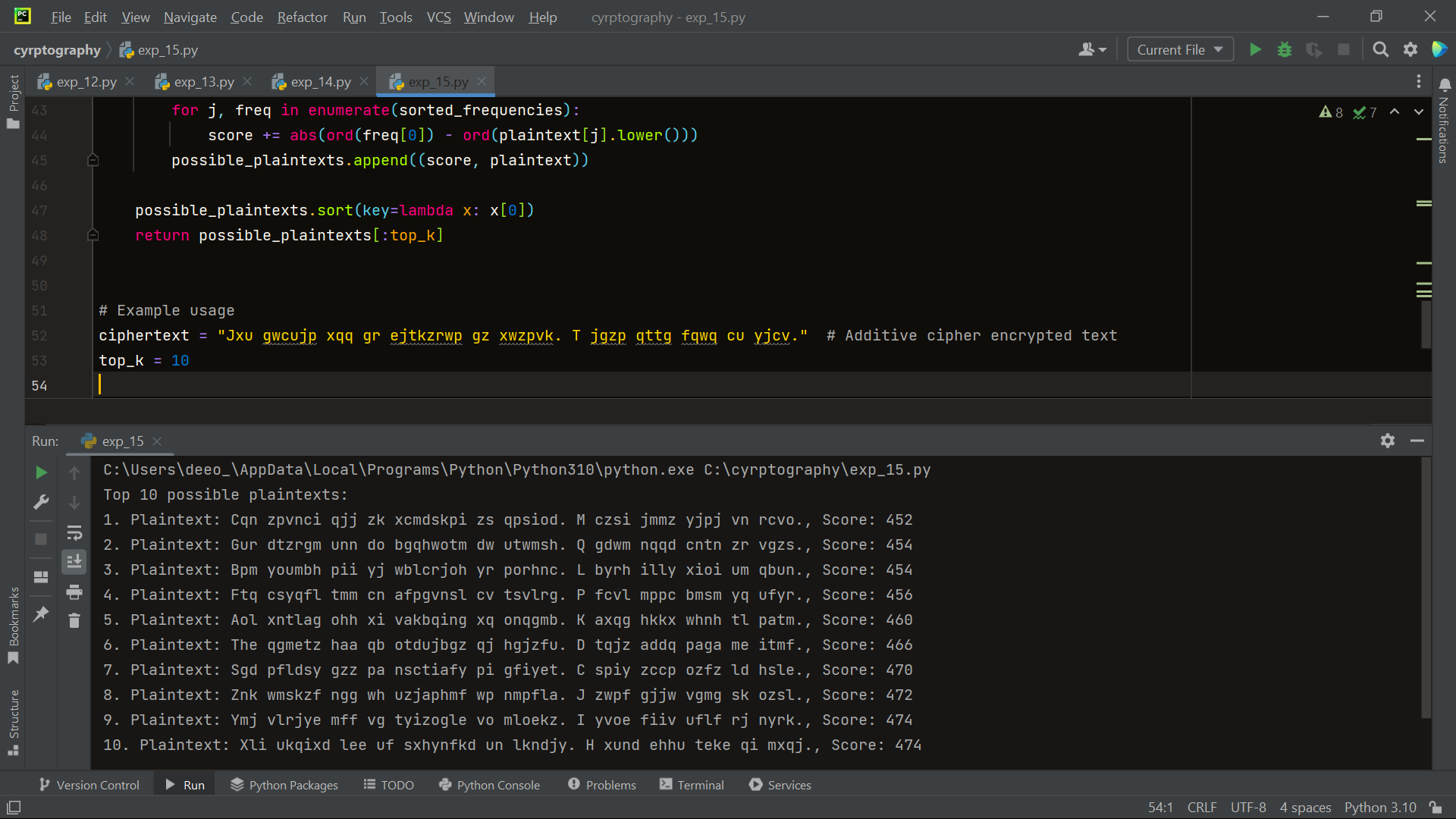
b. Using the ciphertext produced in part (a), find a key so that the cipher text decrypts to the plaintext cash not needed

plaintext = 'sendmoremoney'  
key = [9, 0, 1, 7, 23, 15, 21, 14, 11, 11, 2, 8, 9]  
numbers = [ord(c) - ord('a') for c in plaintext]  
ciphertext = ''  
for i, num in enumerate(numbers):  
 key\_num = key[i % len(key)]  
 encrypted\_num = (num + key\_num) % 26  
 ciphertext += chr(encrypted\_num + ord('a'))  
  
print('Plaintext:', plaintext)  
print('Key:', key)  
print('Ciphertext:', ciphertext)



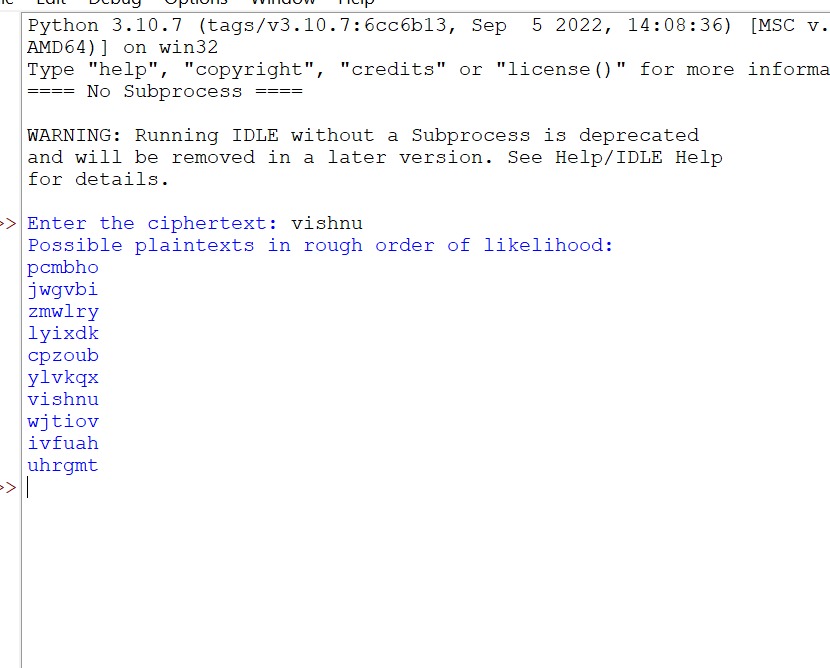
1. Write a Python program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

import string  
  
  
def get\_letter\_frequencies(ciphertext):  
 frequencies = {letter: 0 for letter in string.ascii\_lowercase}  
 total\_letters = 0  
  
 for letter in ciphertext:  
 if letter.isalpha():  
 frequencies[letter.lower()] += 1  
 total\_letters += 1  
  
 for letter in frequencies:  
 frequencies[letter] /= total\_letters  
  
 return frequencies  
  
  
def decrypt(ciphertext, key):  
 plaintext = ""  
 for letter in ciphertext:  
 if letter.isalpha():  
 is\_upper = letter.isupper()  
 letter = letter.lower()  
 decrypted\_letter = chr((ord(letter) - ord('a') - key) % 26 + ord('a'))  
 if is\_upper:  
 decrypted\_letter = decrypted\_letter.upper()  
 plaintext += decrypted\_letter  
 else:  
 plaintext += letter  
 return plaintext  
  
  
def letter\_frequency\_attack(ciphertext, top\_k=10):  
 frequencies = get\_letter\_frequencies(ciphertext)  
 sorted\_frequencies = sorted(frequencies.items(), key=lambda x: x[1], reverse=True)  
 possible\_plaintexts = []  
  
 for i in range(26):  
 key = i  
 plaintext = decrypt(ciphertext, key)  
 score = 0  
 for j, freq in enumerate(sorted\_frequencies):  
 score += abs(ord(freq[0]) - ord(plaintext[j].lower()))  
 possible\_plaintexts.append((score, plaintext))  
  
 possible\_plaintexts.sort(key=lambda x: x[0])  
 return possible\_plaintexts[:top\_k]  
  
  
# Example usage  
ciphertext = "Jxu gwcujp xqq gr ejtkzrwp gz xwzpvk. T jgzp qttg fqwq cu yjcv." # Additive cipher encrypted text  
top\_k = 10  
  
results = letter\_frequency\_attack(ciphertext, top\_k)  
  
print(f"Top {top\_k} possible plaintexts:")  
for i, result in enumerate(results, 1):  
 score, plaintext = result  
 print(f"{i}. Plaintext: {plaintext}, Score: {score}")



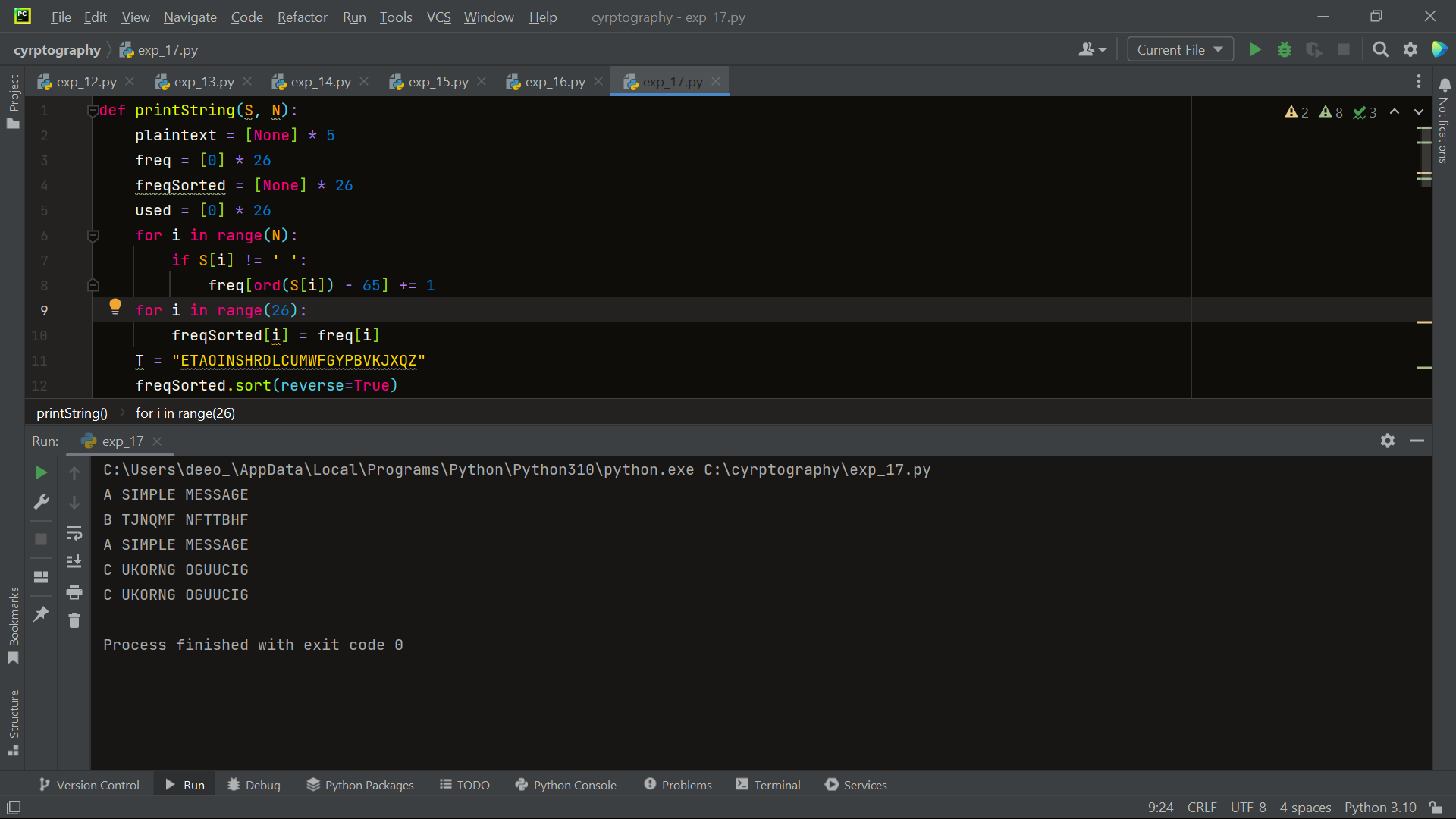
1. Write a Python program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

import string  
import operator  
  
  
def get\_cipher\_text():  
 cipher\_text = input("Enter the ciphertext: ")  
 return cipher\_text.lower()  
  
  
def letter\_frequency\_analysis(cipher\_text, num\_results=10):  
 freq = {'a': 8.167, 'b': 1.492, 'c': 2.782, 'd': 4.253, 'e': 12.702,  
 'f': 2.228, 'g': 2.015, 'h': 6.094, 'i': 6.966, 'j': 0.153,  
 'k': 0.772, 'l': 4.025, 'm': 2.406, 'n': 6.749, 'o': 7.507,  
 'p': 1.929, 'q': 0.095, 'r': 5.987, 's': 6.327, 't': 9.056,  
 'u': 2.758, 'v': 0.978, 'w': 2.360, 'x': 0.150, 'y': 1.974,  
 'z': 0.074}  
  
 letter\_count = {}  
 for letter in string.ascii\_lowercase:  
 letter\_count[letter] = cipher\_text.count(letter)  
  
 sorted\_letters = sorted(letter\_count.items(), key=operator.itemgetter(1), reverse=True)  
  
 total\_count = sum(letter\_count.values())  
 letter\_freq = {}  
 for letter, count in sorted\_letters:  
 letter\_freq[letter] = (count / total\_count) \* 100  
  
 distances = {}  
 for i in range(26):  
 distance = 0  
 for letter in freq:  
 index = (i + ord(letter) - ord('a')) % 26  
 distance += abs(freq[letter] - letter\_freq[chr(index + ord('a'))])  
 distances[i] = distance  
  
 sorted\_distances = sorted(distances.items(), key=operator.itemgetter(1))  
  
 print("Possible plaintexts in rough order of likelihood:")  
 for i in range(num\_results):  
 offset = sorted\_distances[i][0]  
 plain\_text = ""  
 for letter in cipher\_text:  
 if letter.isalpha():  
 plain\_text += chr((ord(letter) - ord('a') + offset) % 26 + ord('a'))  
 else:  
 plain\_text += letter  
 print(plain\_text)  
  
  
def main():  
 cipher\_text = get\_cipher\_text()  
 letter\_frequency\_analysis(cipher\_text, 10)  
  
  
if \_\_name\_\_ == '\_main\_':  
 main()



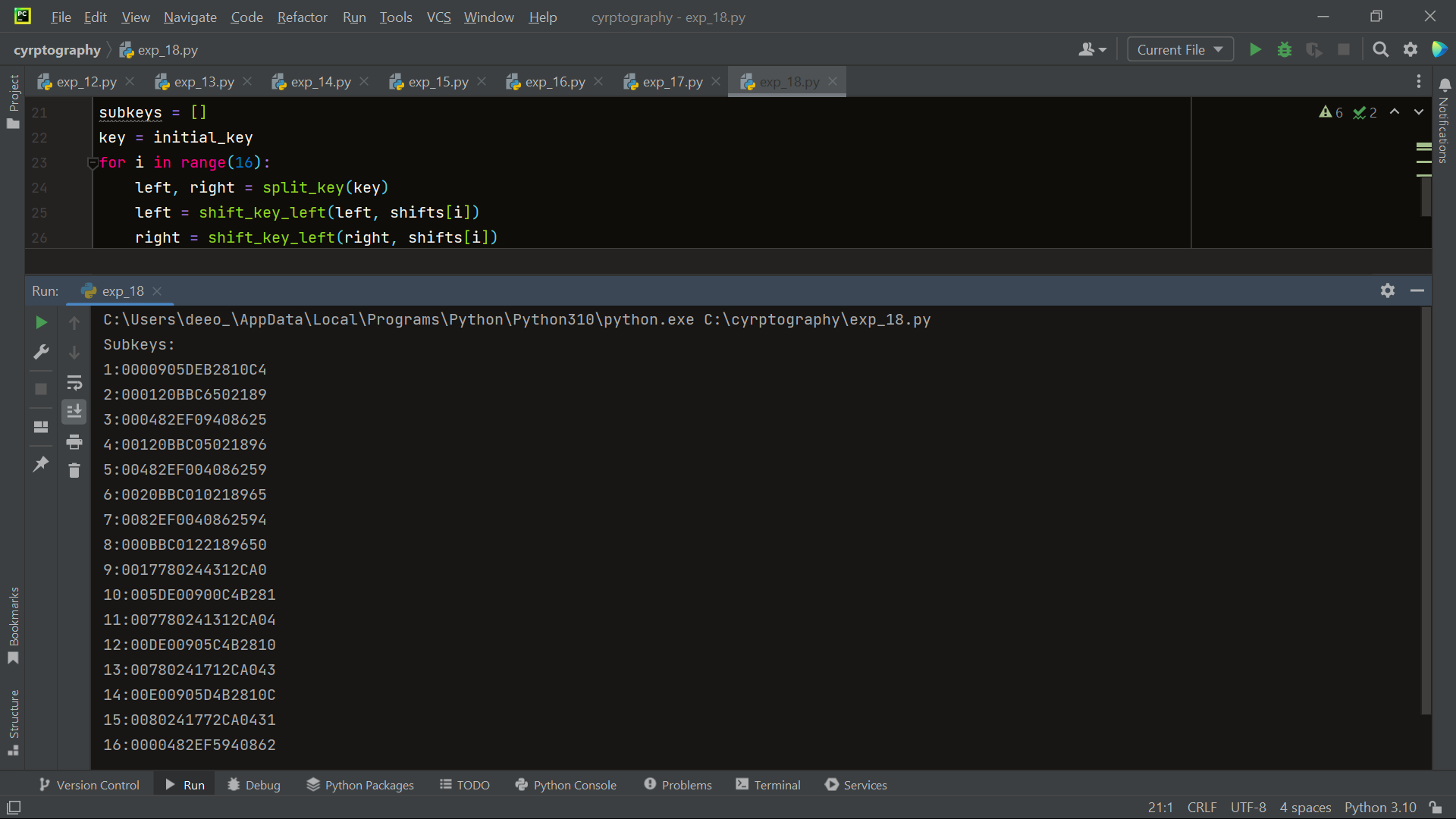
1. Write a Python program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used in reverse order. Design a key-generation scheme with the appropriate shift schedule for the decryption process

def printString(S, N):  
 plaintext = [None] \* 5  
 freq = [0] \* 26  
 freqSorted = [None] \* 26  
 used = [0] \* 26  
 for i in range(N):  
 if S[i] != ' ':  
 freq[ord(S[i]) - 65] += 1  
 for i in range(26):  
 freqSorted[i] = freq[i]  
 T = "ETAOINSHRDLCUMWFGYPBVKJXQZ"  
 freqSorted.sort(reverse=True)  
 for i in range(5):  
 ch = -1  
 for j in range(26):  
 if freqSorted[i] == freq[j] and used[j] == 0:  
 used[j] = 1  
 ch = j  
 break  
  
 if ch == -1:  
 break  
 x = ord(T[i]) - 65  
 x = x - ch  
 curr = ""  
 for k in range(N):  
 if S[k] == ' ':  
 curr += " "  
 continue  
 y = ord(S[k]) - 65  
 y += x  
  
 if y < 0:  
 y += 26  
 if y > 25:  
 y -= 26  
 curr += chr(y + 65)  
  
 plaintext[i] = curr  
  
 for i in range(5):  
 print(plaintext[i])  
  
  
S = "B TJNQMF NFTTBHF"  
N = len(S)  
  
printString(S,N)



1. Write a Python program for DES the first 24 bits of each subkey come from the same subset of 28 bits of the initial key and that the second 24 bits of each subkey come from a disjoint subset of 28 bits of the initial key.

import random  
  
initial\_key = random.getrandbits(64)  
  
  
def split\_key(key):  
 left = key >> 28  
 right = key & 0xFFFFFFF  
 return left, right  
  
  
def combine\_key(left, right):  
 return (left << 28) | right  
  
  
def shift\_key\_left(key, n):  
 return ((key << n) & 0xFFFFFFF) | (key >> (28 - n))  
  
  
shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]  
subkeys = []  
key = initial\_key  
for i in range(16):  
 left, right = split\_key(key)  
 left = shift\_key\_left(left, shifts[i])  
 right = shift\_key\_left(right, shifts[i])  
 subkey = combine\_key(left, right)  
 subkeys.append(subkey)  
 key = subkey  
print("Subkeys:")  
for i, subkey in enumerate(subkeys):  
 print(f"{i + 1}:{subkey:016X}")

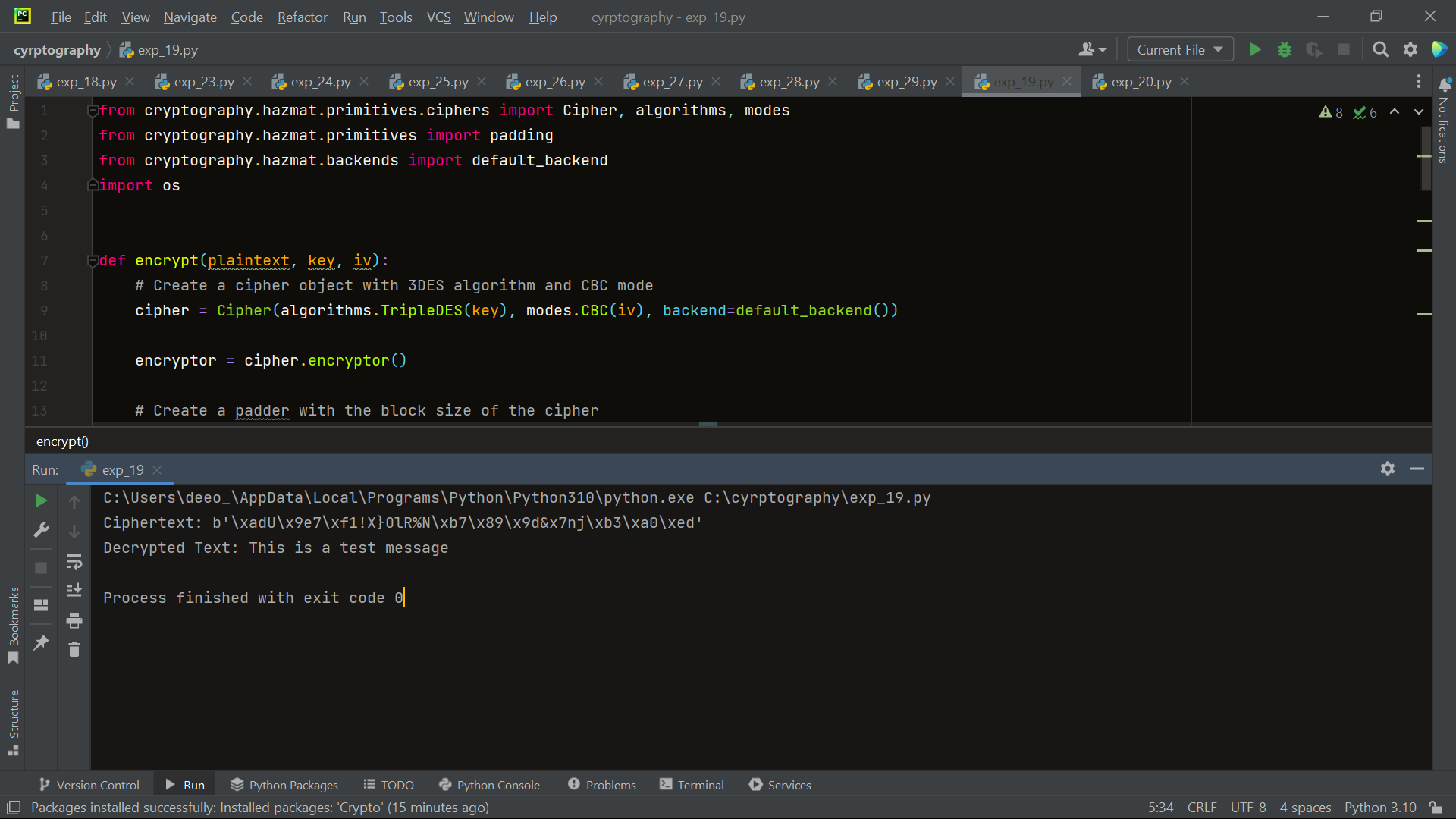


19. Write a Python program for encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the definition of CBC. Which of the two would you choose:

a. For security?

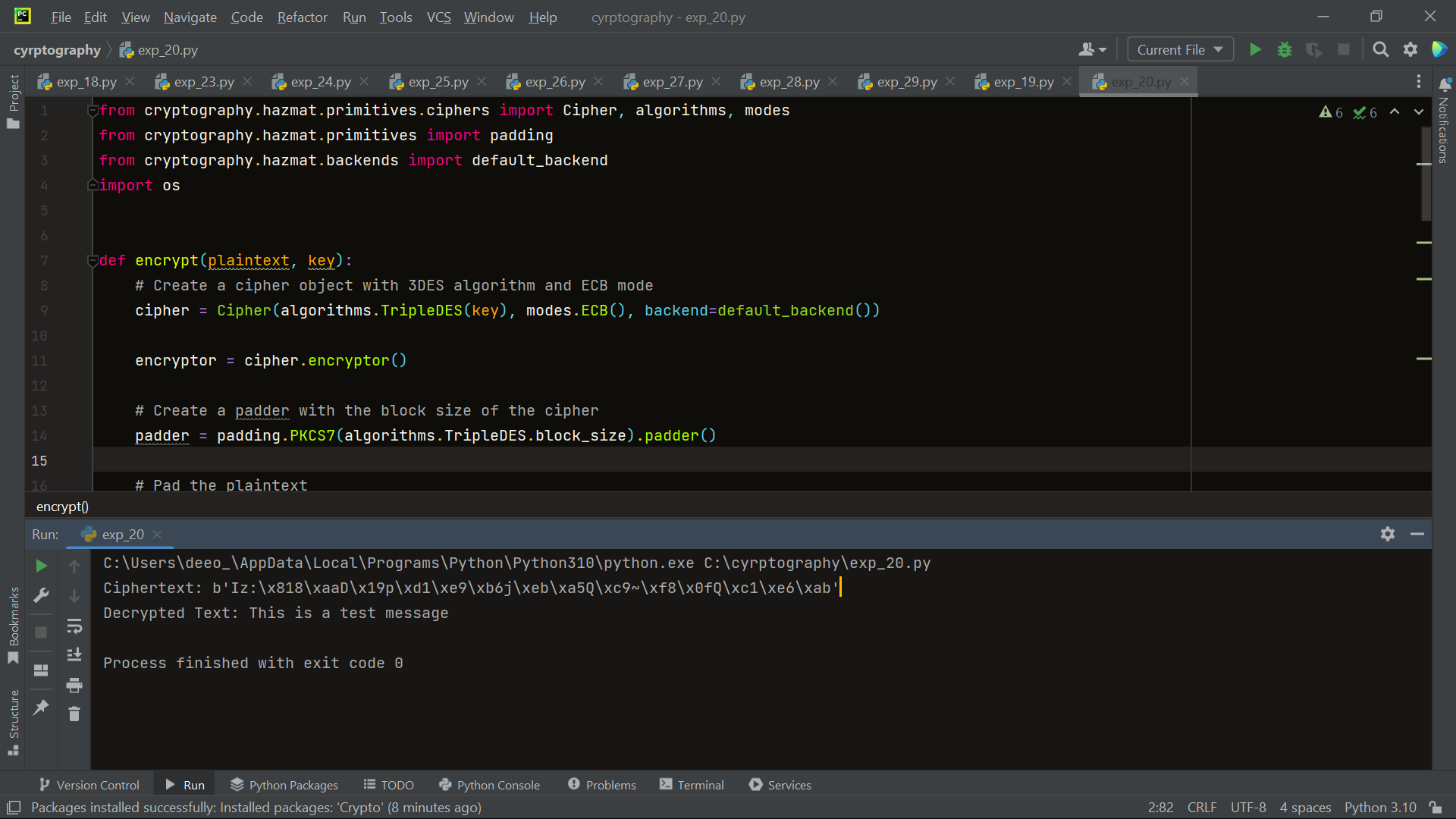
b. For performance?

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.primitives import padding  
from cryptography.hazmat.backends import default\_backend  
import os  
  
  
def encrypt(plaintext, key, iv):  
 # Create a cipher object with 3DES algorithm and CBC mode  
 cipher = Cipher(algorithms.TripleDES(key), modes.CBC(iv), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.TripleDES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def decrypt(ciphertext, key, iv):  
 # Create a cipher object with 3DES algorithm and CBC mode  
 cipher = Cipher(algorithms.TripleDES(key), modes.CBC(iv), backend=default\_backend())  
  
 decryptor = cipher.decryptor()  
  
 # Decrypt the ciphertext  
 decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()  
  
 # Create an unpadder with the block size of the cipher  
 unpadder = padding.PKCS7(algorithms.TripleDES.block\_size).unpadder()  
  
 # Unpad the decrypted data  
 plaintext = unpadder.update(decrypted\_data) + unpadder.finalize()  
  
 return plaintext  
  
  
# Example usage  
key = os.urandom(24) # Generate a random 24-byte 3DES key  
iv = os.urandom(algorithms.TripleDES.block\_size // 8) # Generate a random IV with the block size of 3DES  
plaintext = b'This is a test message'  
  
# Encryption  
ciphertext = encrypt(plaintext, key, iv)  
print("Ciphertext:", ciphertext)  
  
# Decryption  
decrypted\_text = decrypt(ciphertext, key, iv)  
print("Decrypted Text:", decrypted\_text.decode())



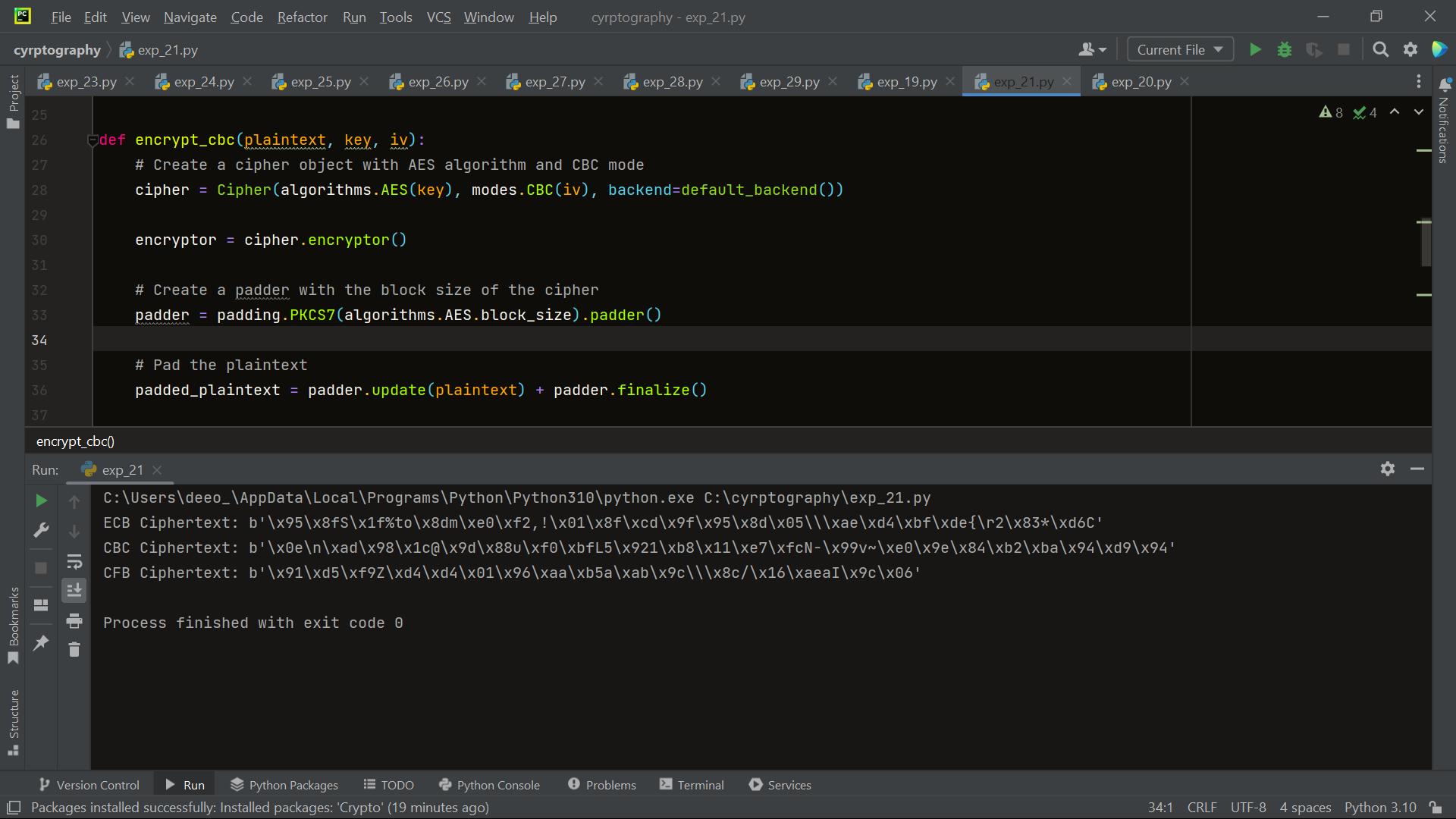
1. Write a Python program for ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error in the transmitted C1 obviously corrupts P1 and P2. a. Are any blocks beyond P2 affected?

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.primitives import padding  
from cryptography.hazmat.backends import default\_backend  
import os  
  
  
def encrypt(plaintext, key):  
 # Create a cipher object with 3DES algorithm and ECB mode  
 cipher = Cipher(algorithms.TripleDES(key), modes.ECB(), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.TripleDES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def decrypt(ciphertext, key):  
 # Create a cipher object with 3DES algorithm and ECB mode  
 cipher = Cipher(algorithms.TripleDES(key), modes.ECB(), backend=default\_backend())  
  
 decryptor = cipher.decryptor()  
  
 # Decrypt the ciphertext  
 decrypted\_data = decryptor.update(ciphertext) + decryptor.finalize()  
  
 # Create an unpadder with the block size of the cipher  
 unpadder = padding.PKCS7(algorithms.TripleDES.block\_size).unpadder()  
  
 # Unpad the decrypted data  
 plaintext = unpadder.update(decrypted\_data) + unpadder.finalize()  
  
 return plaintext  
  
  
# Example usage  
key = os.urandom(24) # Generate a random 24-byte 3DES key  
plaintext = b'This is a test message'  
  
# Encryption  
ciphertext = encrypt(plaintext, key)  
print("Ciphertext:", ciphertext)  
  
# Decryption  
decrypted\_text = decrypt(ciphertext, key)  
print("Decrypted Text:", decrypted\_text.decode())



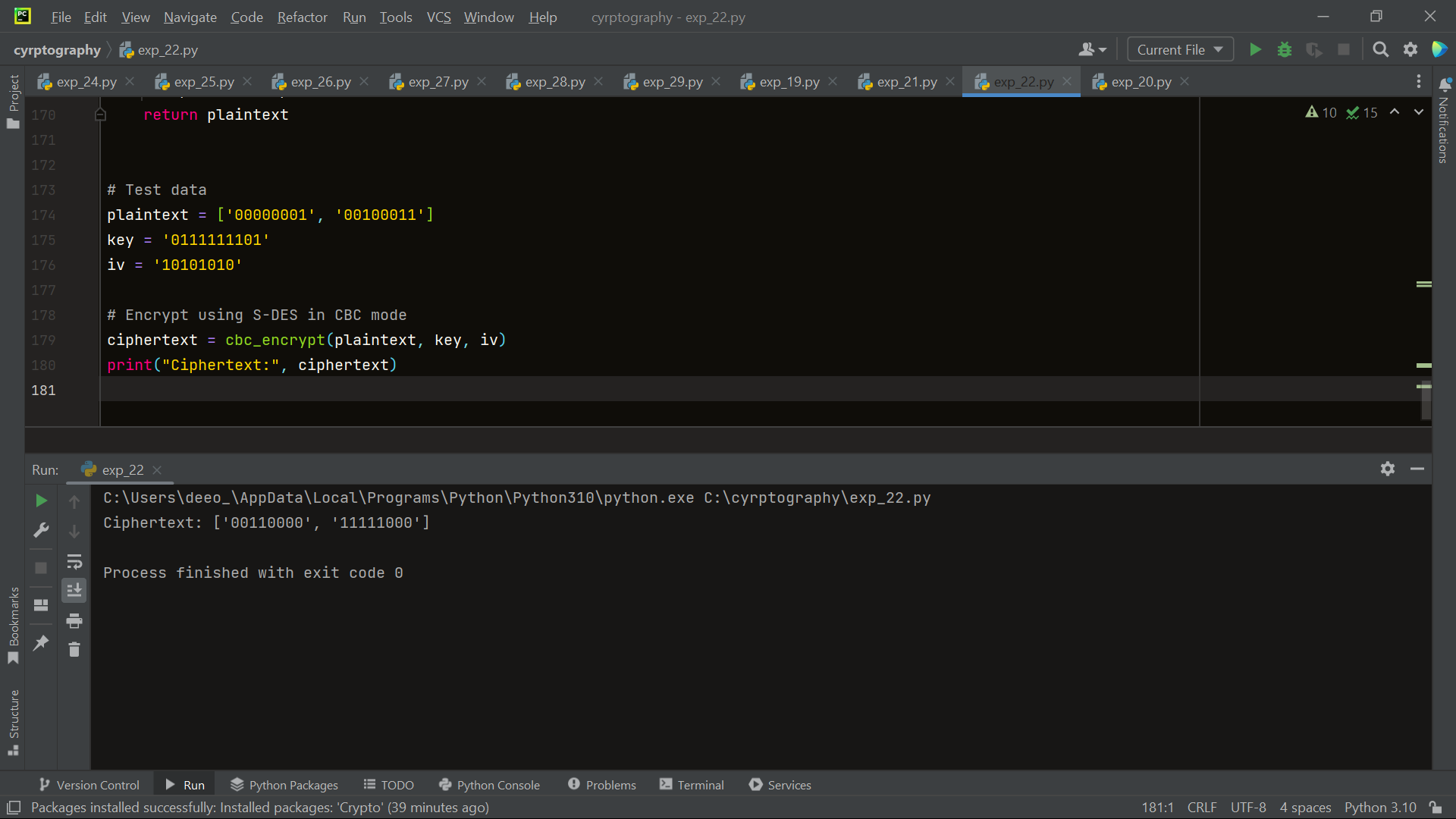
1. Write a Python program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.primitives import padding  
from cryptography.hazmat.backends import default\_backend  
import os  
  
  
def encrypt\_ecb(plaintext, key):  
 # Create a cipher object with AES algorithm and ECB mode  
 cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.AES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def encrypt\_cbc(plaintext, key, iv):  
 # Create a cipher object with AES algorithm and CBC mode  
 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.AES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def encrypt\_cfb(plaintext, key, iv):  
 # Create a cipher object with AES algorithm and CFB mode  
 cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Encrypt the plaintext  
 ciphertext = encryptor.update(plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
# Example usage  
key = os.urandom(16) # Generate a random 16-byte AES key  
iv = os.urandom(algorithms.AES.block\_size // 8) # Generate a random IV with the block size of AES  
plaintext = b'This is a test message'  
  
# Encryption in ECB mode  
ciphertext\_ecb = encrypt\_ecb(plaintext, key)  
print("ECB Ciphertext:", ciphertext\_ecb)  
  
# Encryption in CBC mode  
ciphertext\_cbc = encrypt\_cbc(plaintext, key, iv)  
print("CBC Ciphertext:", ciphertext\_cbc)  
  
# Encryption in CFB mode  
ciphertext\_cfb = encrypt\_cfb(plaintext, key, iv)  
print("CFB Ciphertext:", ciphertext\_cfb)

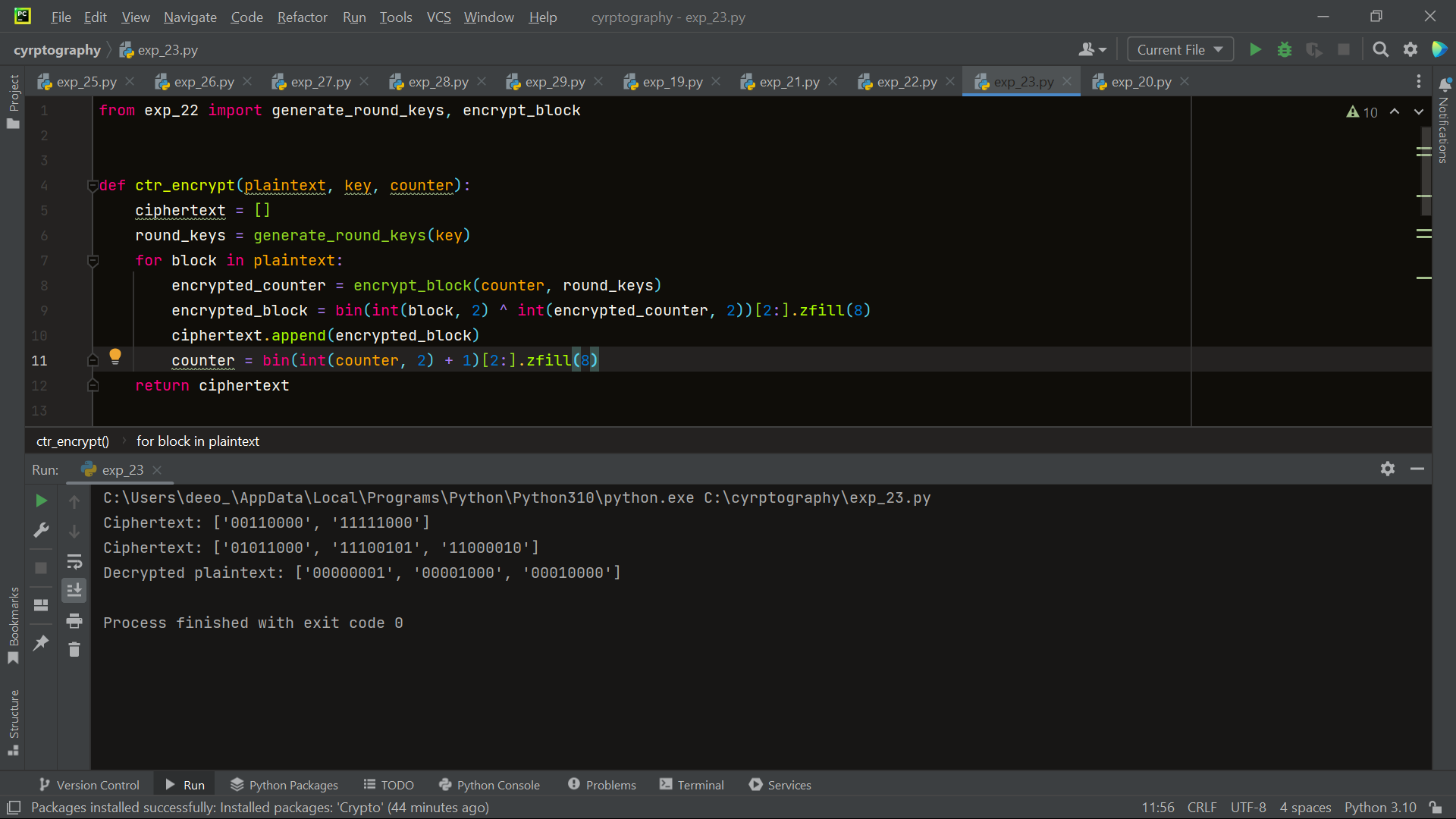


1. Write a Python program for Encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.

P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]  
P8 = [6, 3, 7, 4, 8, 5, 10, 9]  
EP = [4, 1, 2, 3, 2, 3, 4, 1]  
P4 = [2, 4, 3, 1]  
IP = [2, 6, 3, 1, 4, 8, 5, 7]  
IP\_INV = [4, 1, 3, 5, 7, 2, 8, 6]  
S0 = [[1, 0, 3, 2], [3, 2, 1, 0], [0, 2, 1, 3], [3, 1, 3, 2]]  
S1 = [[0, 1, 2, 3], [2, 0, 1, 3], [3, 0, 1, 0], [2, 1, 0, 3]]  
  
  
def apply\_permutation(bits, permutation):  
 return ''.join(bits[i - 1] for i in permutation)  
  
  
def generate\_round\_keys(key):  
 round\_keys = []  
 key = apply\_permutation(key, P10)  
 left, right = key[:5], key[5:]  
 for i in range(2):  
 left = left[1:] + left[0]  
 right = right[1:] + right[0]  
 round\_key = apply\_permutation(left + right, P8)  
 round\_keys.append(round\_key)  
 return round\_keys  
  
  
def f\_function(bits, round\_key):  
 expanded = apply\_permutation(bits, EP)  
 xored = bin(int(expanded, 2) ^ int(round\_key, 2))[2:].zfill(8)  
 left, right = xored[:4], xored[4:]  
 sbox\_output = apply\_sbox(left, S0) + apply\_sbox(right, S1)  
 return apply\_permutation(sbox\_output, P4)  
  
  
def apply\_sbox(bits, sbox):  
 row = int(bits[0] + bits[3], 2)  
 col = int(bits[1:3], 2)  
 return format(sbox[row][col], '02b')  
  
  
def encrypt\_block(block, round\_keys):  
 block = apply\_permutation(block, IP)  
 left, right = block[:4], block[4:]  
 for i in range(2):  
 f\_result = f\_function(right, round\_keys[i])  
 new\_right = bin(int(left, 2) ^ int(f\_result, 2))[2:].zfill(4)  
 left = right  
 right = new\_right  
 cipher\_text = apply\_permutation(right + left, IP\_INV)  
 return cipher\_text  
  
  
def decrypt\_block(block, round\_keys):  
 block = apply\_permutation(block, IP)  
 left, right = block[:4], block[4:]  
 for i in range(2):  
 f\_result = f\_function(right, round\_keys[1 - i])  
 new\_right = bin(int(left, 2) ^ int(f\_result, 2))[2:].zfill(4)  
 left = right  
 right = new\_right  
 plain\_text = apply\_permutation(right + left, IP\_INV)  
 return plain\_text  
  
  
# CBC encryption  
def cbc\_encrypt(plaintext, key, iv):  
 ciphertext = []  
 round\_keys = generate\_round\_keys(key)  
 previous\_cipher\_block = iv  
 for block in plaintext:  
 block = bin(int(block, 2) ^ int(previous\_cipher\_block, 2))[2:].zfill(8)  
 encrypted\_block = encrypt\_block(block, round\_keys)  
 ciphertext.append(encrypted\_block)  
 previous\_cipher\_block = encrypted\_block  
 return ciphertext  
  
  
# CBC decryption  
def cbc\_decrypt(ciphertext, key, iv):  
 plaintext = []  
 round\_keys = generate\_round\_keys(key)  
 previous\_cipher\_block = iv  
 for block in ciphertext:  
 decrypted\_block = decrypt\_block(block, round\_keys)  
 decrypted\_block = bin(int(decrypted\_block, 2) ^ int(previous\_cipher\_block, 2))[2:].zfill(8)  
 plaintext.append(decrypted\_block)  
 previous\_cipher\_block = block  
 return plaintext  
  
  
# Test data  
plaintext = ['00000001', '00100011']  
key = '0111111101'  
iv = '10101010'  
  
# Encrypt using S-DES in CBC mode  
ciphertext = cbc\_encrypt(plaintext, key, iv)  
print("Ciphertext:", ciphertext)

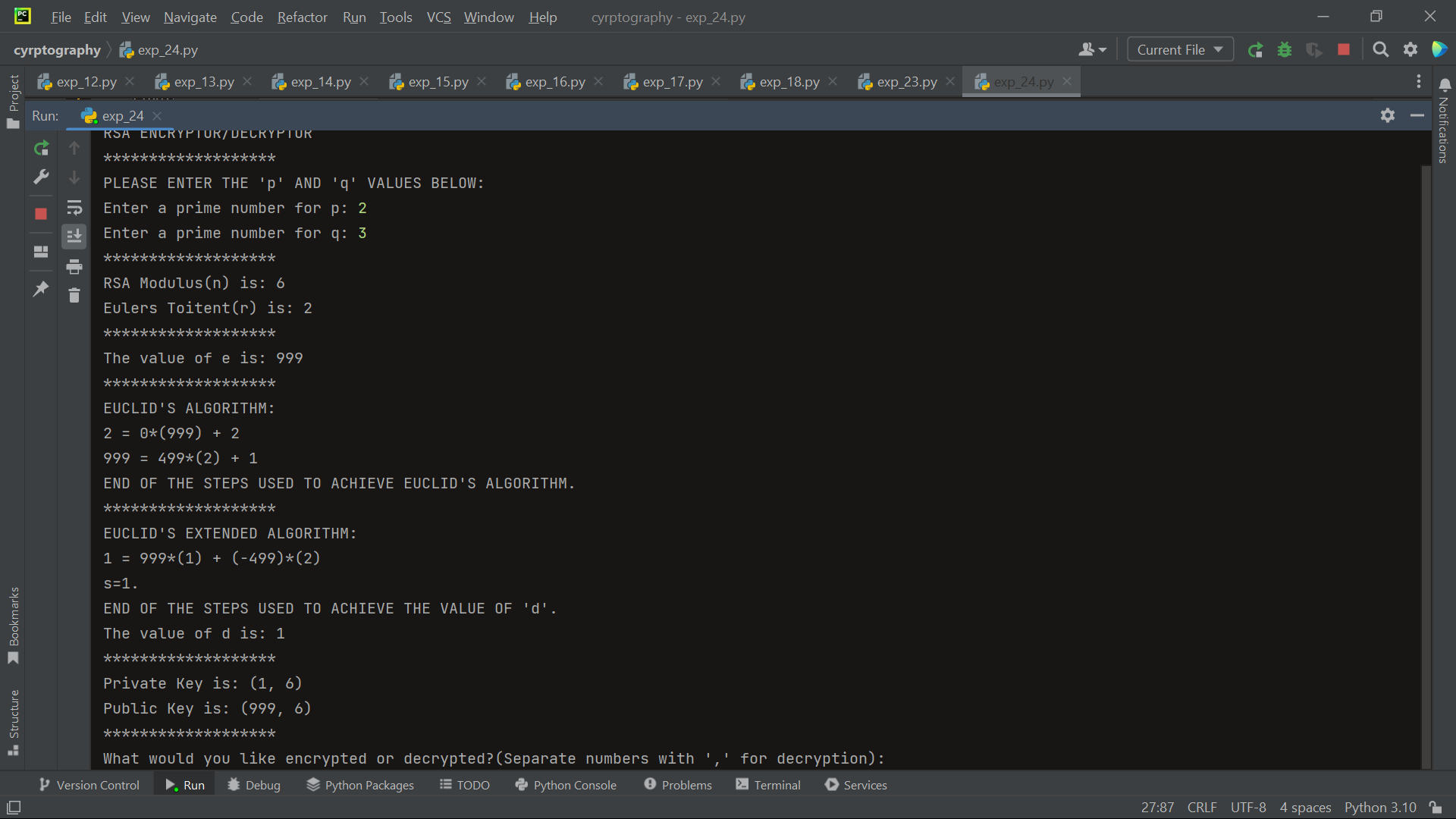


1. Write a Python program for Encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES. Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.
2. from exp\_22 import generate\_round\_keys, encrypt\_block  
     
     
   def ctr\_encrypt(plaintext, key, counter):  
    ciphertext = []  
    round\_keys = generate\_round\_keys(key)  
    for block in plaintext:  
    encrypted\_counter = encrypt\_block(counter, round\_keys)  
    encrypted\_block = bin(int(block, 2) ^ int(encrypted\_counter, 2))[2:].zfill(8)  
    ciphertext.append(encrypted\_block)  
    counter = bin(int(counter, 2) + 1)[2:].zfill(8)  
    return ciphertext  
     
     
   # Counter mode decryption  
   def ctr\_decrypt(ciphertext, key, counter):  
    plaintext = []  
    round\_keys = generate\_round\_keys(key)  
    for block in ciphertext:  
    decrypted\_counter = encrypt\_block(counter, round\_keys)  
    decrypted\_block = bin(int(block, 2) ^ int(decrypted\_counter, 2))[2:].zfill(8)  
    plaintext.append(decrypted\_block)  
    counter = bin(int(counter, 2) + 1)[2:].zfill(8)  
    return plaintext  
     
     
   # Test data  
   plaintext = ['00000001', '00001000', '00010000']  
   key = '0111111101'  
   counter = '00000000'  
     
   # Encrypt using S-DES in counter mode  
   ciphertext = ctr\_encrypt(plaintext, key, counter)  
   print("Ciphertext:", ciphertext)  
     
   # Decrypt using S-DES in counter mode  
   decrypted\_plaintext = ctr\_decrypt(ciphertext, key, counter)  
   print("Decrypted plaintext:", decrypted\_plaintext)



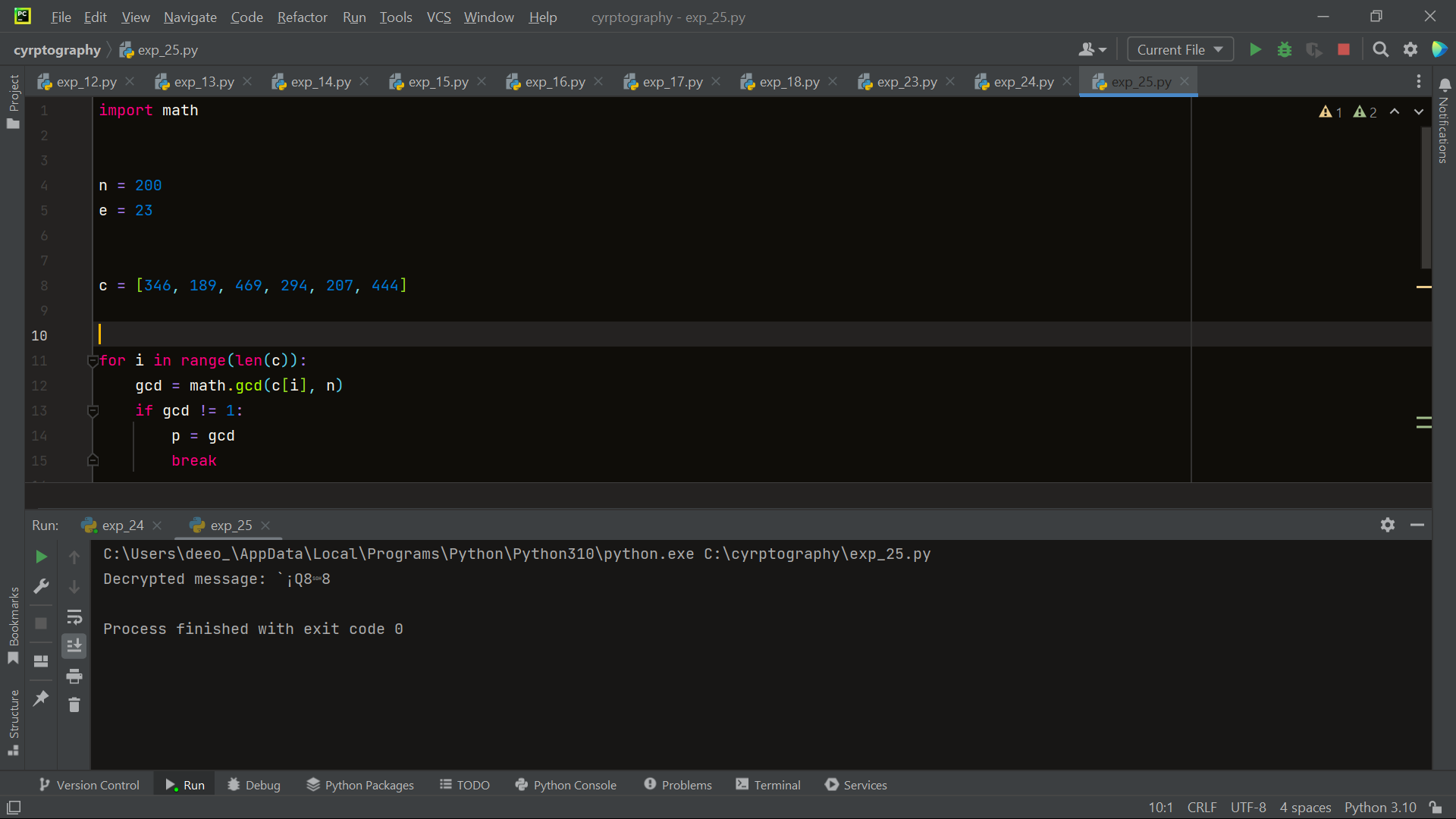
24. Write a Python program for RSA system, the public key of a given user is e = 31, n = 3599. What is the private key of this user? Hint: First use trial-and-error to determine p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).

import math  
  
print("RSA ENCRYPTOR/DECRYPTOR")  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
# Input Prime Numbers  
print("PLEASE ENTER THE 'p' AND 'q' VALUES BELOW:")  
p = int(input("Enter a prime number for p: "))  
q = int(input("Enter a prime number for q: "))  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
# Check if Input's are Prime  
'''THIS FUNCTION AND THE CODE IMMEDIATELY BELOW THE FUNCTION CHECKS WHETHER THE INPUTS ARE PRIME OR NOT.'''  
  
  
def prime\_check(a):  
 if (a == 2):  
 return True  
 elif ((a < 2) or ((a % 2) == 0)):  
 return False  
 elif (a > 2):  
 for i in range(2, a):  
 if not (a % i):  
 return false  
 return True  
  
  
check\_p = prime\_check(p)  
check\_q = prime\_check(q)  
while (check\_p == False) or (check\_q == False):  
 p = int(input("Enter a prime number for p: "))  
 q = int(input("Enter a prime number for q: "))  
 check\_p = prime\_check(p)  
 check\_q = prime\_check(q)  
  
# RSA Modulus  
'''CALCULATION OF RSA MODULUS 'n'.'''  
n = p \* q  
print("RSA Modulus(n) is:", n)  
  
# Eulers Toitent  
'''CALCULATION OF EULERS TOITENT 'r'.'''  
r = (p - 1) \* (q - 1)  
print("Eulers Toitent(r) is:", r)  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
# GCD  
'''CALCULATION OF GCD FOR 'e' CALCULATION.'''  
  
  
def egcd(e, r):  
 while (r != 0):  
 e, r = r, e % r  
 return e  
  
  
# Euclid's Algorithm  
def eugcd(e, r):  
 for i in range(1, r):  
 while (e != 0):  
 a, b = r // e, r % e  
 if (b != 0):  
 print("%d = %d\*(%d) + %d" % (r, a, e, b))  
 r = e  
 e = b  
  
  
# Extended Euclidean Algorithm  
def eea(a, b):  
 if (a % b == 0):  
 return (b, 0, 1)  
 else:  
 gcd, s, t = eea(b, a % b)  
 s = s - ((a // b) \* t)  
 print("%d = %d\*(%d) + (%d)\*(%d)" % (gcd, a, t, s, b))  
 return (gcd, t, s)  
  
  
# Multiplicative Inverse  
def mult\_inv(e, r):  
 gcd, s, \_ = eea(e, r)  
 if (gcd != 1):  
 return None  
 else:  
 if (s < 0):  
 print("s=%d. Since %d is less than 0, s = s(modr), i.e., s=%d." % (s, s, s % r))  
 elif (s > 0):  
 print("s=%d." % (s))  
 return s % r  
  
  
# e Value Calculation  
'''FINDS THE HIGHEST POSSIBLE VALUE OF 'e' BETWEEN 1 and 1000 THAT MAKES (e,r) COPRIME.'''  
for i in range(1, 1000):  
 if (egcd(i, r) == 1):  
 e = i  
print("The value of e is:", e)  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
# d, Private and Public Keys  
'''CALCULATION OF 'd', PRIVATE KEY, AND PUBLIC KEY.'''  
print("EUCLID'S ALGORITHM:")  
eugcd(e, r)  
print("END OF THE STEPS USED TO ACHIEVE EUCLID'S ALGORITHM.")  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
print("EUCLID'S EXTENDED ALGORITHM:")  
d = mult\_inv(e, r)  
print("END OF THE STEPS USED TO ACHIEVE THE VALUE OF 'd'.")  
print("The value of d is:", d)  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
public = (e, n)  
private = (d, n)  
print("Private Key is:", private)  
print("Public Key is:", public)  
print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
# Encryption  
'''ENCRYPTION ALGORITHM.'''  
  
  
def encrypt(pub\_key, n\_text):  
 e, n = pub\_key  
 x = []  
 m = 0  
 for i in n\_text:  
 if (i.isupper()):  
 m = ord(i) - 65  
 c = (m \*\* e) % n  
 x.append(c)  
 elif (i.islower()):  
 m = ord(i) - 97  
 c = (m \*\* e) % n  
 x.append(c)  
 elif (i.isspace()):  
 spc = 400  
 x.append(400)  
 return x  
  
  
# Decryption  
'''DECRYPTION ALGORITHM'''  
  
  
def decrypt(priv\_key, c\_text):  
 d, n = priv\_key  
 txt = c\_text.split(',')  
 x = ''  
 m = 0  
 for i in txt:  
 if (i == '400'):  
 x += ' '  
 else:  
 m = (int(i) \*\* d) % n  
 m += 65  
 c = chr(m)  
 x += c  
 return x  
  
  
# Message  
message = input("What would you like encrypted or decrypted?(Separate numbers with ',' for decryption):")  
print("Your message is:", message)  
  
# Choose Encrypt or Decrypt and Print  
choose = input("Type '1' for encryption and '2' for decrytion.")  
if (choose == '1'):  
 enc\_msg = encrypt(public, message)  
 print("Your encrypted message is:", enc\_msg)  
 print("Thank you for using the RSA Encryptor. Goodbye!")  
elif (choose == '2'):  
 print("Your decrypted message is:", decrypt(private, message))  
 print("Thank you for using the RSA Encryptor. Goodbye!")  
else:  
 print("You entered the wrong option.")  
 print("Thank you for using the RSA Encryptor. Goodbye!")



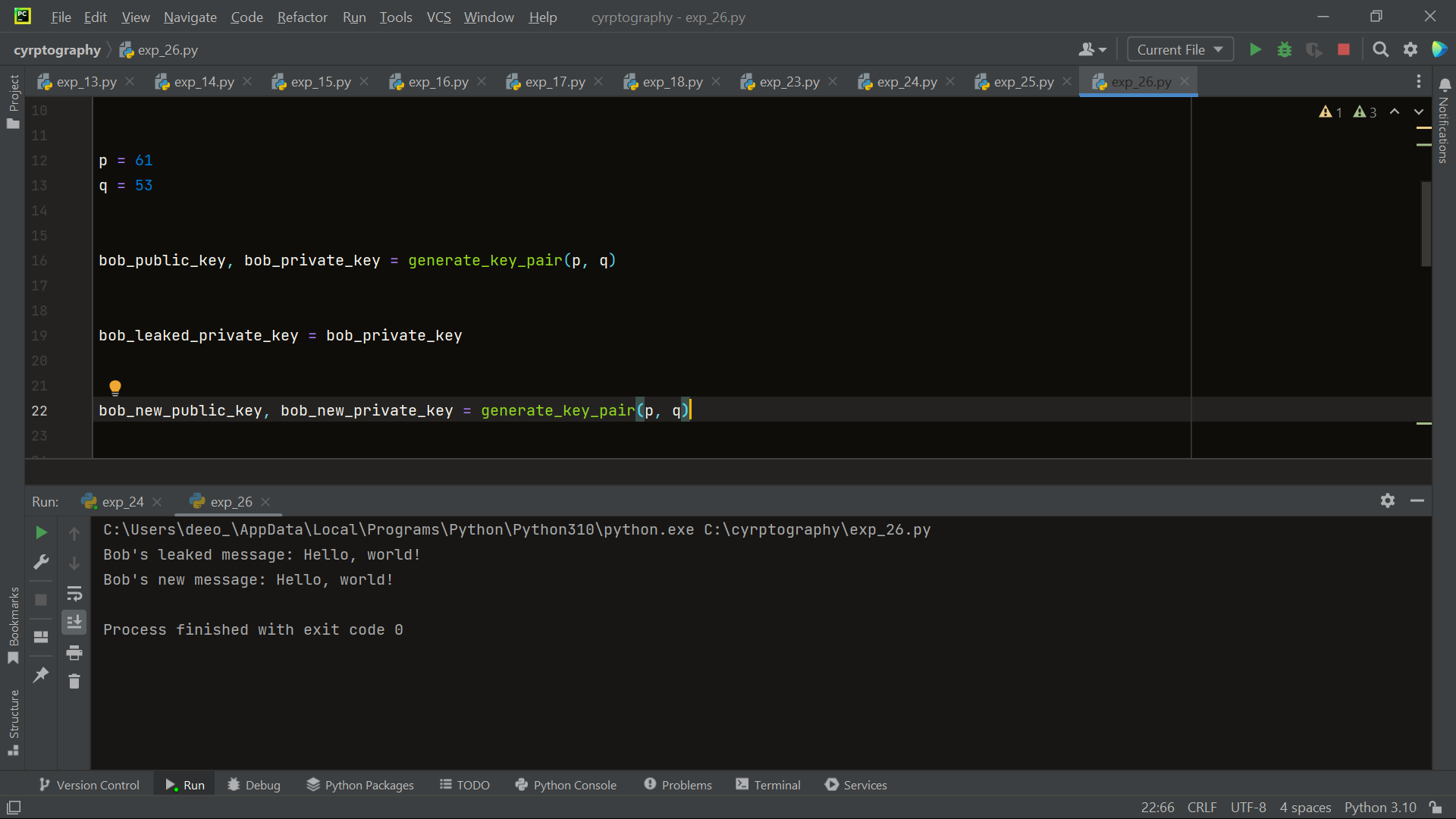
25. Write a Python program for set of blocks encoded with the RSA algorithm and we don’t have the private key. Assume n = pq, e is the public key. Suppose also someone tells us they know one of the plaintext blocks has a common factor with n. Does this help us in any way?

import math  
  
  
n = 200  
e = 23  
  
  
c = [346, 189, 469, 294, 207, 444]  
  
  
for i in range(len(c)):  
 gcd = math.gcd(c[i], n)  
 if gcd != 1:  
 p = gcd  
 break  
  
  
q = n // p  
  
  
phi = (p-1) \* (q-1)  
d = pow(e, -1, phi)  
  
  
m = []  
for i in range(len(c)):  
 m.append(pow(c[i], d, n))  
  
  
print("Decrypted message: ", end="")  
for i in range(len(m)):  
 print(chr(m[i]),end="")  
print()



26. Write a Python program for RSA public-key encryption scheme, each user has a public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than generating a new modulus, he decides to generate a new public and a new private key. Is this safe?

import math  
  
  
def generate\_key\_pair(p, q):  
 n = p \* q  
 phi = (p-1) \* (q-1)  
 e = 65537  
 d = pow(e, -1, phi)  
 return (e, n), (d, n)  
  
  
p = 61  
q = 53  
  
  
bob\_public\_key, bob\_private\_key = generate\_key\_pair(p, q)  
  
  
bob\_leaked\_private\_key = bob\_private\_key  
  
  
bob\_new\_public\_key, bob\_new\_private\_key = generate\_key\_pair(p, q)  
  
  
message = "Hello, world!"  
  
  
ciphertext = ""  
for char in message:  
 m = ord(char)  
 c = pow(m, bob\_public\_key[0], bob\_public\_key[1])  
 ciphertext += str(c) + " "  
  
  
plaintext = ""  
for c in ciphertext.split():  
 c = int(c)  
 m = pow(c, bob\_leaked\_private\_key[0], bob\_public\_key[1])  
 plaintext += chr(m)  
  
print("Bob's leaked message:", plaintext)  
  
  
plaintext = ""  
for c in ciphertext.split():  
 c = int(c)  
 m = pow(c, bob\_new\_private\_key[0], bob\_new\_public\_key[1])  
 plaintext += chr(m)  
  
print("Bob's new message:", plaintext)



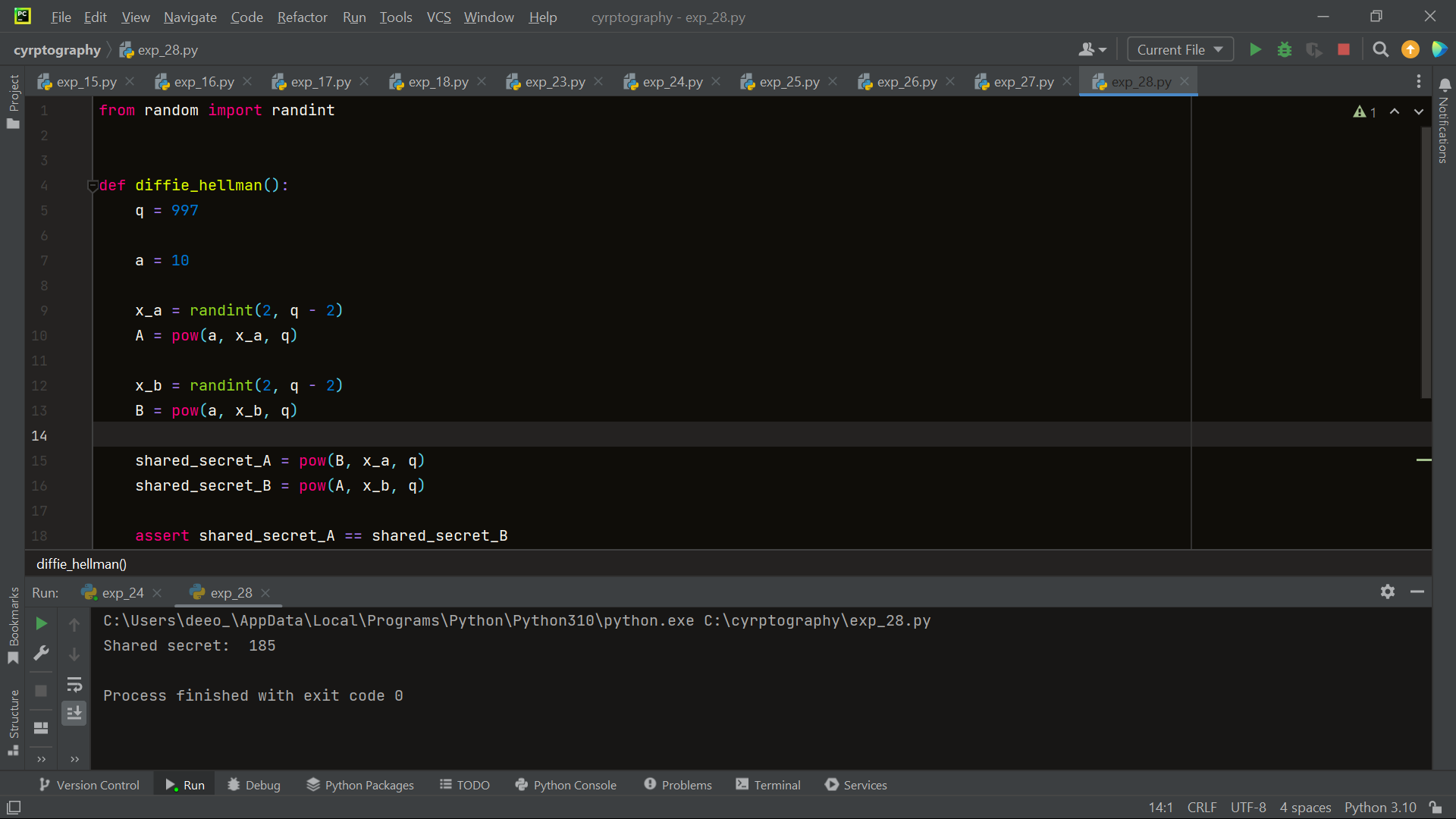
27. Write a Python program for Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 (A S 0, c, Z S 25) and then encrypting each number separately using RSA with large e and large n. Is this method secure? If not, describe the most efficient attack against this encryption method.

import math  
  
  
def generate\_key\_pair(p, q):  
 n = p \* q  
 phi = (p-1) \* (q-1)  
 e = 65537  
 d = pow(e, -1, phi)  
 return (e, n), (d, n)  
  
  
p = 61  
q = 53  
  
  
bob\_public\_key, bob\_private\_key = generate\_key\_pair(p, q)  
  
  
def encrypt\_message(message, public\_key):  
 ciphertext = ""  
 for char in message:  
 m = ord(char) - ord('A')  
 c = pow(m, public\_key[0], public\_key[1])  
 ciphertext += str(c) + " "  
 return ciphertext  
  
  
def decrypt\_message(ciphertext, private\_key):  
 plaintext = ""  
 for c in ciphertext.split():  
 c = int(c)  
 m = pow(c, private\_key[0], private\_key[1])  
 plaintext += chr(m + ord('A'))  
 return plaintext  
  
  
message = "HELLO WORLD"  
  
  
ciphertext = encrypt\_message(message, bob\_public\_key)  
  
  
def frequency\_analysis(ciphertext):  
 frequencies = [0] \* 26  
 for c in ciphertext.split():  
 c = int(c)  
 for i in range(26):  
 if pow(i, bob\_public\_key[0], bob\_public\_key[1]) == c:  
 frequencies[i] += 1  
 return frequencies  
  
  
frequencies = frequency\_analysis(ciphertext)  
  
  
for i in range(26):  
 print(chr(i + ord('A')), ":", frequencies[i])  
  
  
plaintext = decrypt\_message(ciphertext, bob\_private\_key)  
  
print("Decrypted message:", plaintext)



28. Write a Python program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?

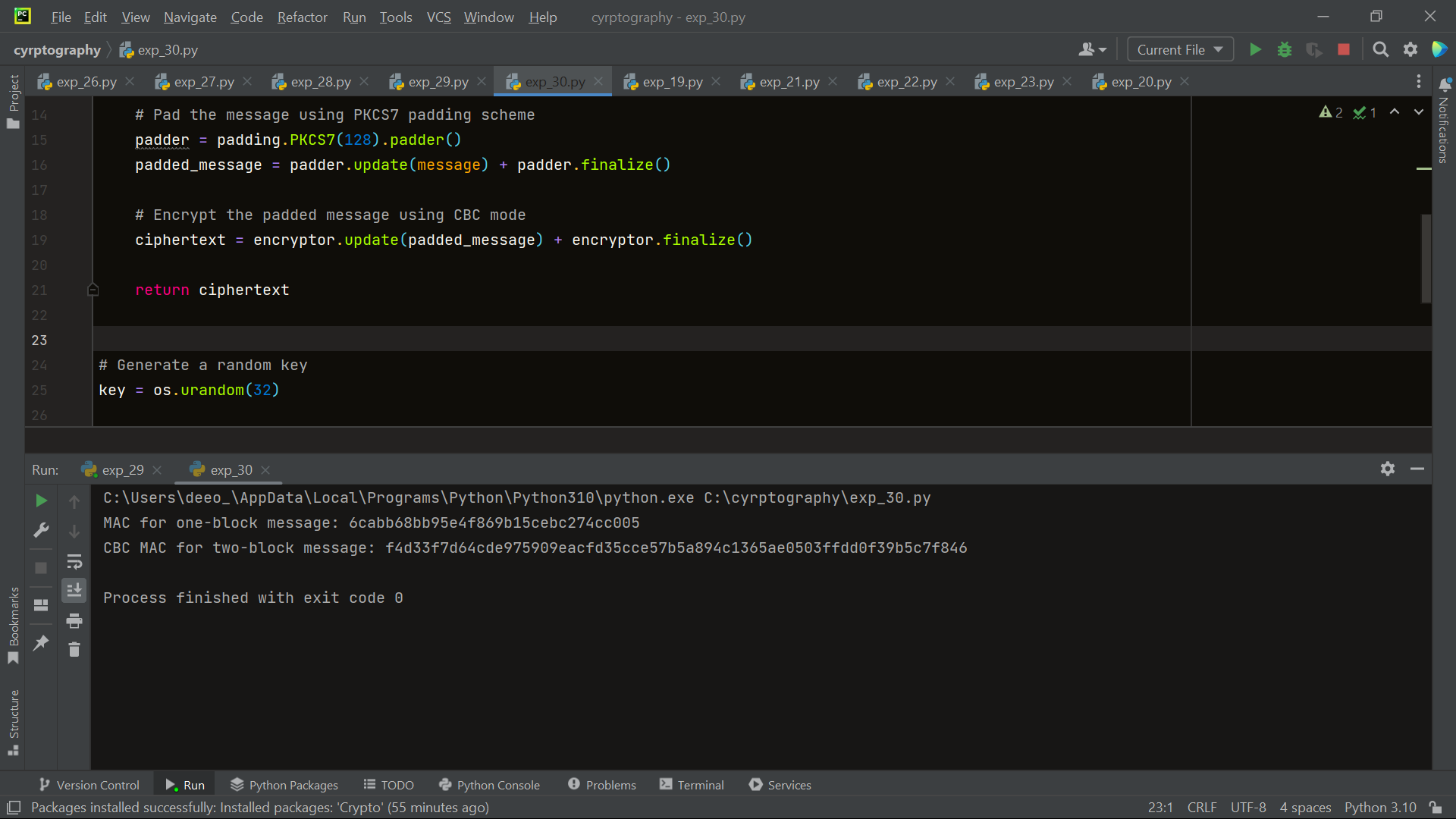
from random import randint  
  
  
def diffie\_hellman():  
 q = 997  
  
 a = 10  
  
 x\_a = randint(2, q - 2)  
 A = pow(a, x\_a, q)  
  
 x\_b = randint(2, q - 2)  
 B = pow(a, x\_b, q)  
  
 shared\_secret\_A = pow(B, x\_a, q)  
 shared\_secret\_B = pow(A, x\_b, q)  
  
 assert shared\_secret\_A == shared\_secret\_B  
  
 return shared\_secret\_A  
  
  
print("Shared secret: ", diffie\_hellman())



29. Write a Python program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix.

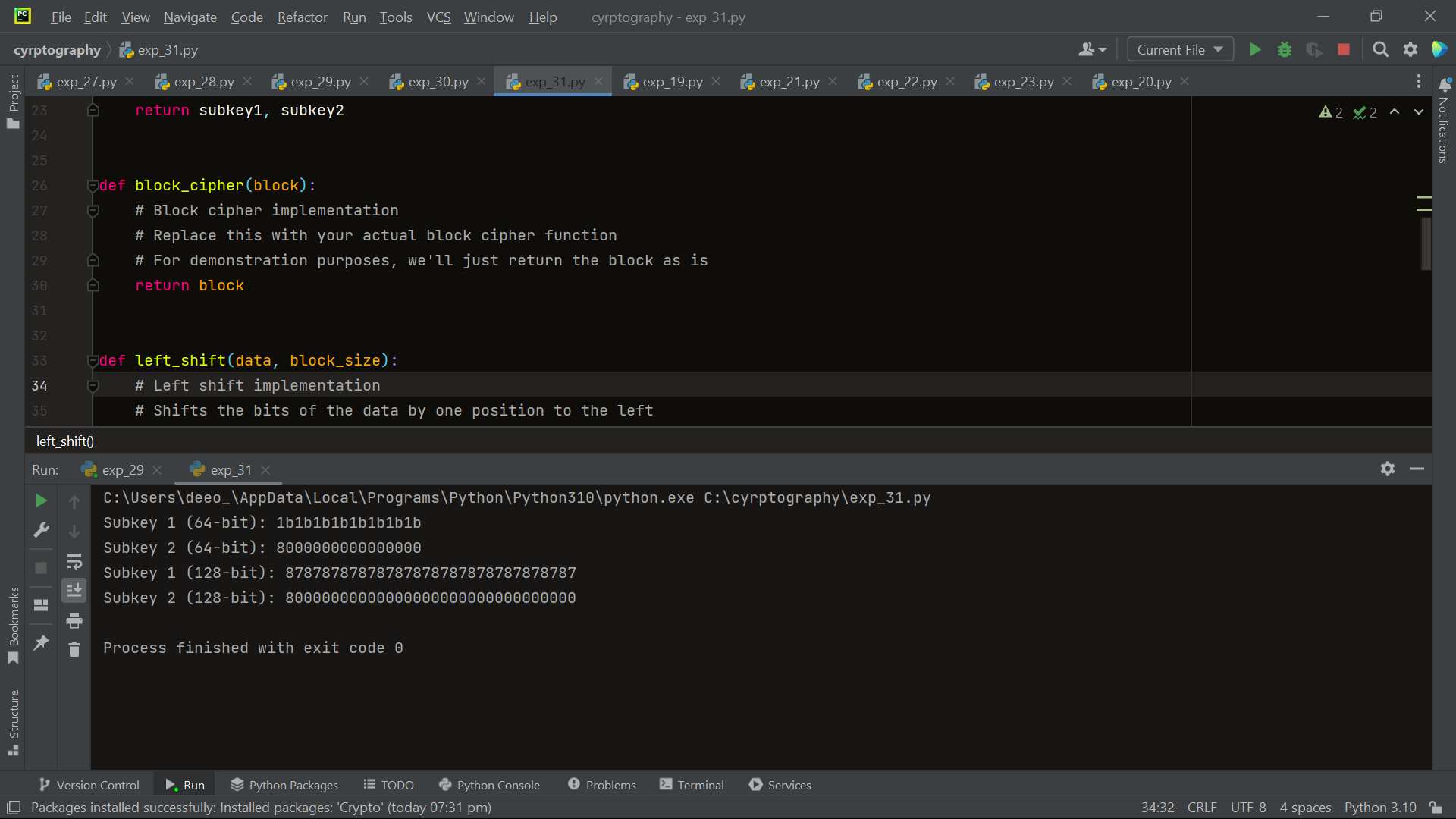
30. Write a Python program for CBC MAC of a one block message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import padding  
import os  
  
  
def cbc\_mac(key, message):  
 iv = os.urandom(16) # Generate a random initialization vector (IV)  
  
 # Create a Cipher object using AES algorithm and CBC mode  
 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())  
 encryptor = cipher.encryptor()  
  
 # Pad the message using PKCS7 padding scheme  
 padder = padding.PKCS7(128).padder()  
 padded\_message = padder.update(message) + padder.finalize()  
  
 # Encrypt the padded message using CBC mode  
 ciphertext = encryptor.update(padded\_message) + encryptor.finalize()  
  
 return ciphertext  
  
  
# Generate a random key  
key = os.urandom(32)  
  
# Define the message  
message = b"Hello, CBC-MAC!"  
  
# Calculate the CBC MAC for the one-block message  
mac = cbc\_mac(key, message)  
print("MAC for one-block message:", mac.hex())  
  
# Construct the two-block message X || (X ⊕ T)  
two\_block\_message = message + bytes(x ^ y for x, y in zip(message, mac))  
  
# Show that the adversary knows the CBC MAC for the two-block message  
print("CBC MAC for two-block message:", cbc\_mac(key, two\_block\_message).hex())



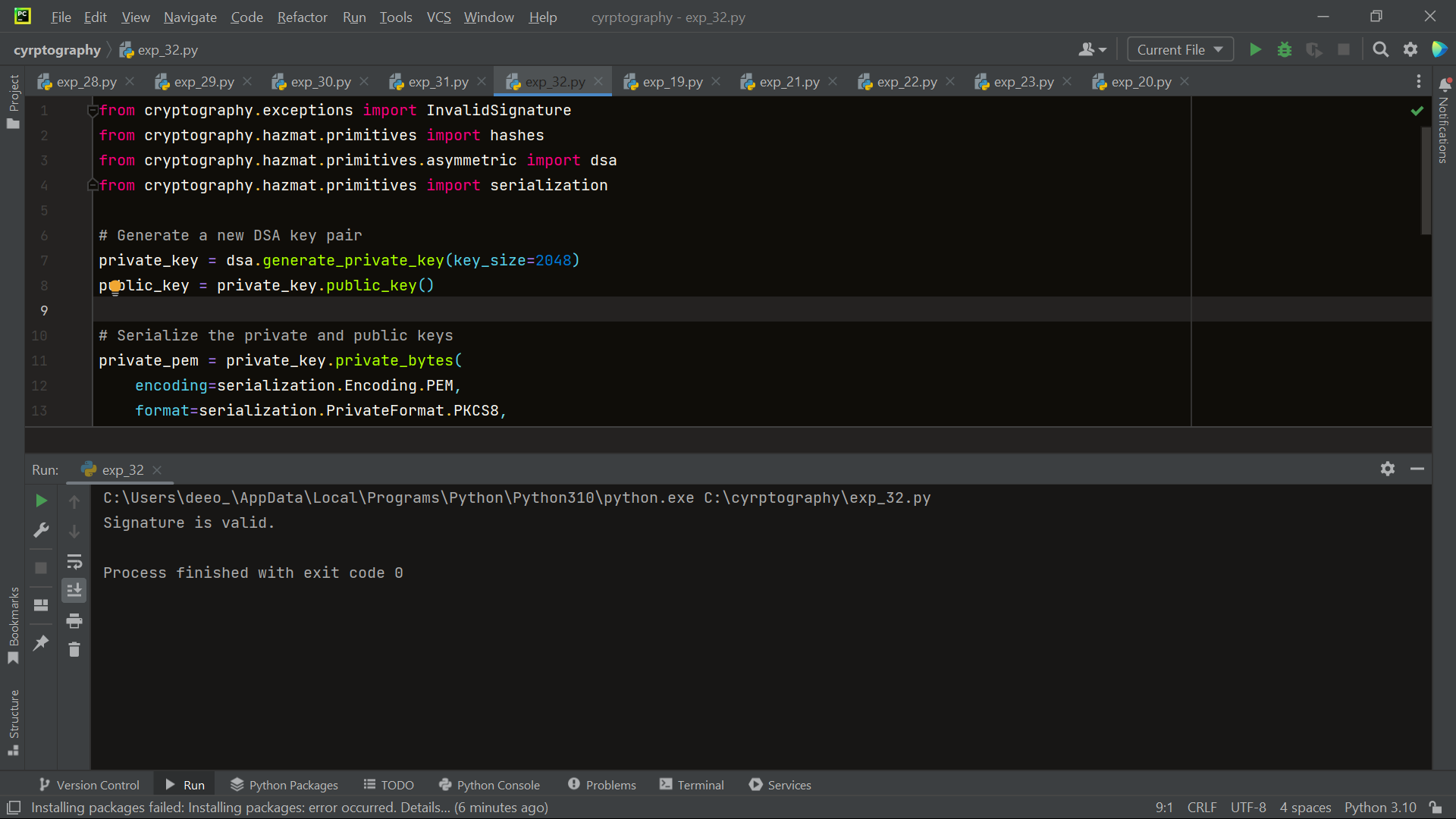
31. Write a Python program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. a. What constants are needed for block sizes of 64 and 128 bits? b. How the left shift and XOR accomplishes the desired result.

def generate\_subkeys(block\_size):  
 if block\_size == 64:  
 constant = 0x1B  
 elif block\_size == 128:  
 constant = 0x87  
 else:  
 raise ValueError("Invalid block size")  
  
 zero\_block = bytes([0] \* (block\_size // 8))  
  
 # Apply the block cipher to the zero block  
 ciphertext = block\_cipher(zero\_block)  
  
 # Left shift the ciphertext by one bit  
 left\_shifted = left\_shift(ciphertext, block\_size)  
  
 # XOR the left shifted ciphertext with the constant  
 subkey1 = xor(left\_shifted, constant)  
  
 # Left shift the first subkey by one bit  
 subkey2 = left\_shift(subkey1, block\_size)  
  
 return subkey1, subkey2  
  
  
def block\_cipher(block):  
 # Block cipher implementation  
 # Replace this with your actual block cipher function  
 # For demonstration purposes, we'll just return the block as is  
 return block  
  
  
def left\_shift(data, block\_size):  
 # Left shift implementation  
 # Shifts the bits of the data by one position to the left  
 # For demonstration purposes, we'll just shift the bytes as is  
 shifted = bytearray(data)  
 shift\_amount = 1  
 while shift\_amount < block\_size:  
 carry = 0  
 for i in range(len(shifted) - 1, -1, -1):  
 temp = shifted[i] & 0x80  
 shifted[i] = ((shifted[i] << 1) & 0xFF) | carry  
 carry = temp >> 7  
 shift\_amount += 1  
 return bytes(shifted)  
  
  
def xor(data, constant):  
 # XOR implementation  
 # Performs bitwise XOR operation between the data and constant  
 # For demonstration purposes, we'll just XOR the bytes as is  
 xored = bytearray(data)  
 for i in range(len(xored)):  
 xored[i] ^= constant  
 return bytes(xored)  
  
  
# Test the subkey generation for block size 64 bits  
subkey1, subkey2 = generate\_subkeys(64)  
print("Subkey 1 (64-bit):", subkey1.hex())  
print("Subkey 2 (64-bit):", subkey2.hex())  
  
# Test the subkey generation for block size 128 bits  
subkey1, subkey2 = generate\_subkeys(128)  
print("Subkey 1 (128-bit):", subkey1.hex())  
print("Subkey 2 (128-bit):", subkey2.hex())



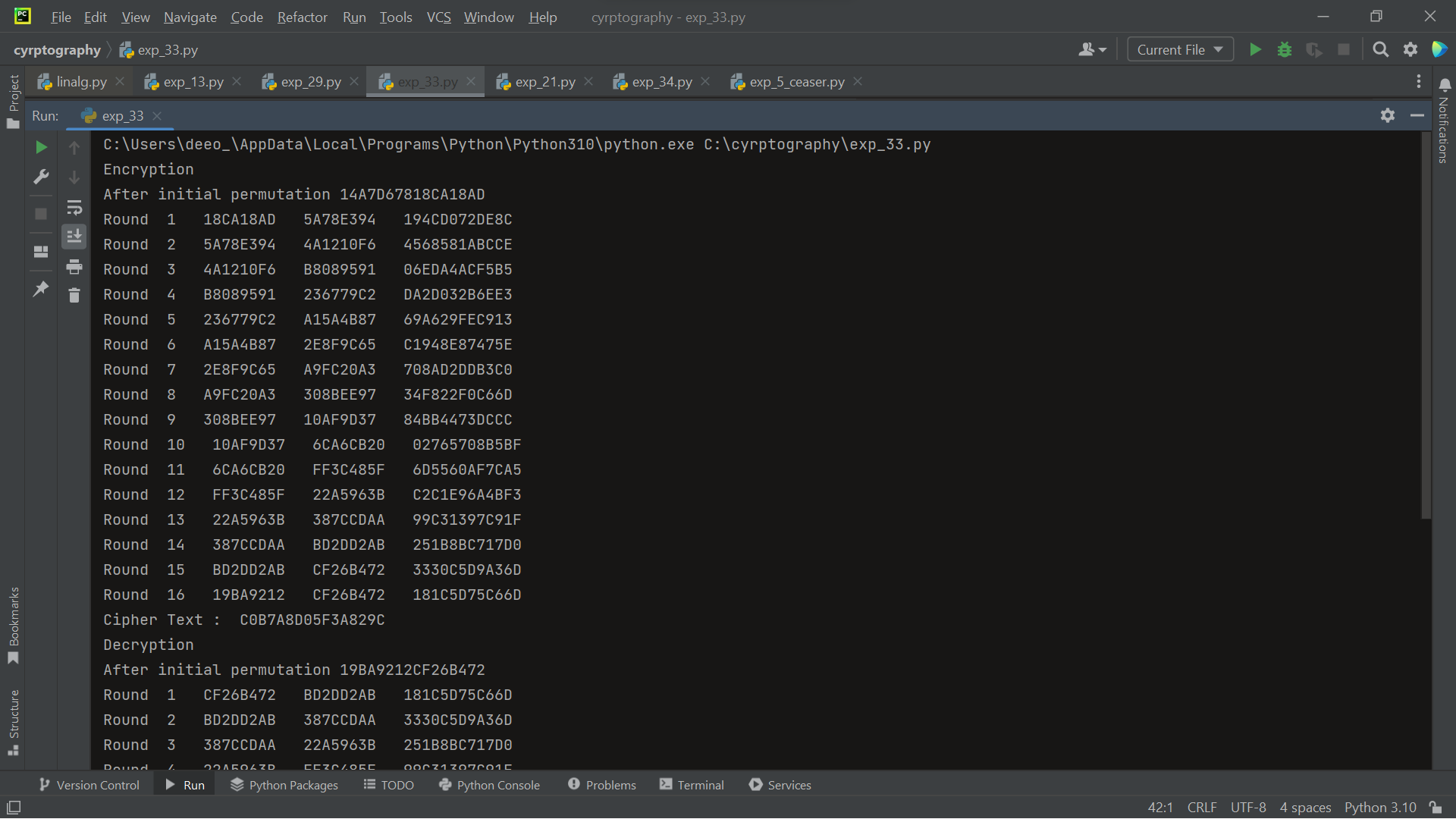
32. Write a Python program for DSA, because the value of k is generated for each signature, even if the same message is signed twice on different occasions, the signatures will differ. This is not true of RSA signatures. Write a Python program for implication of this difference?

from cryptography.exceptions import InvalidSignature  
from cryptography.hazmat.primitives import hashes  
from cryptography.hazmat.primitives.asymmetric import dsa  
from cryptography.hazmat.primitives import serialization  
  
# Generate a new DSA key pair  
private\_key = dsa.generate\_private\_key(key\_size=2048)  
public\_key = private\_key.public\_key()  
  
# Serialize the private and public keys  
private\_pem = private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.PKCS8,  
 encryption\_algorithm=serialization.NoEncryption()  
)  
public\_pem = public\_key.public\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PublicFormat.SubjectPublicKeyInfo  
)  
  
# Sign a message using SHA-256  
message = b"Message to be signed"  
signature = private\_key.sign(message, hashes.SHA256())  
  
# Verify the signature  
try:  
 public\_key.verify(signature, message, hashes.SHA256())  
 print("Signature is valid.")  
except InvalidSignature:  
 print("Signature is invalid.")



33. Write a Python program for Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. Implement in Python programming

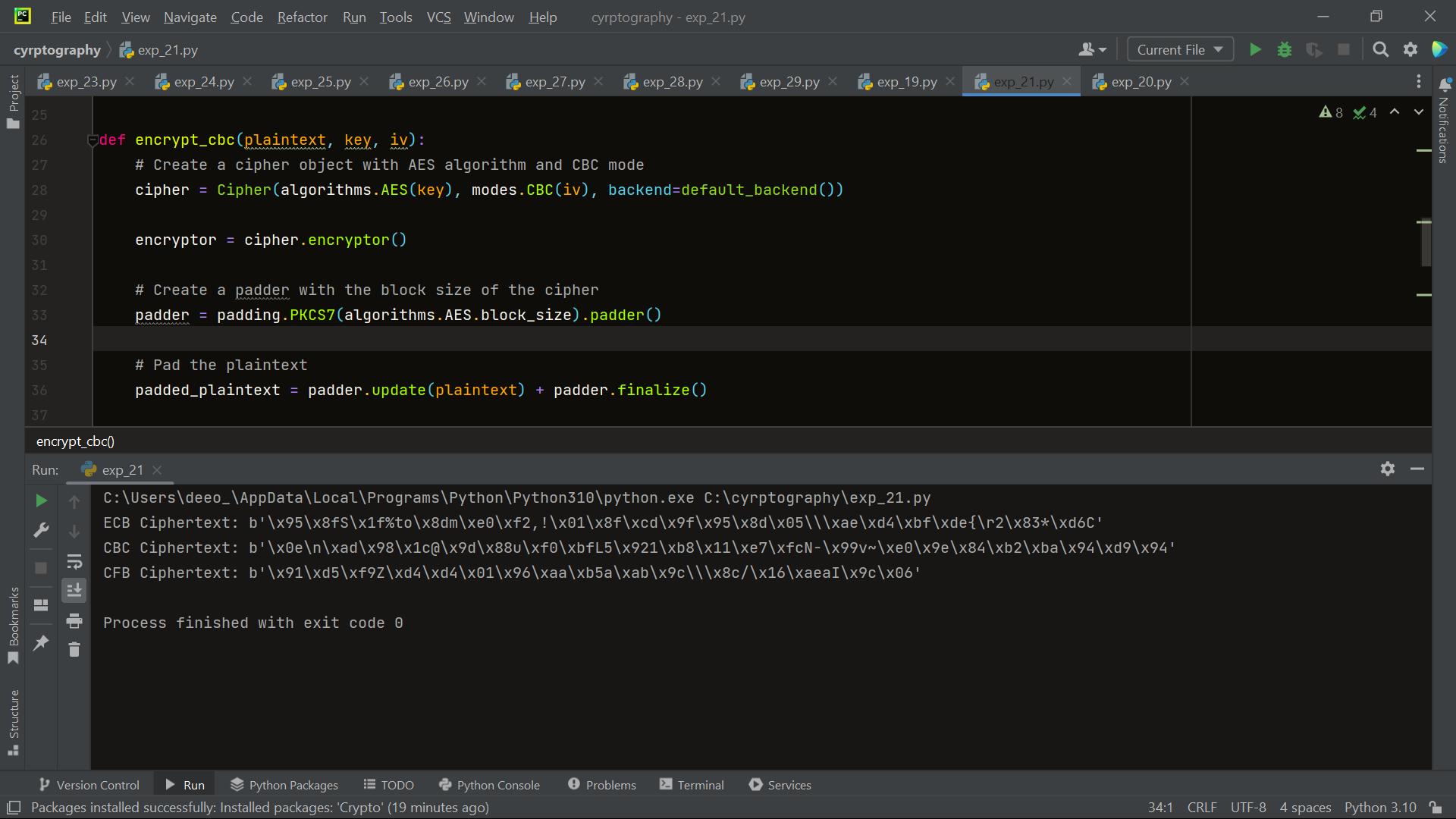
def hex2bin(s):  
 mp = {'0': "0000",  
 '1': "0001",  
 '2': "0010",  
 '3': "0011",  
 '4': "0100",  
 '5': "0101",  
 '6': "0110",  
 '7': "0111",  
 '8': "1000",  
 '9': "1001",  
 'A': "1010",  
 'B': "1011",  
 'C': "1100",  
 'D': "1101",  
 'E': "1110",  
 'F': "1111"}  
 bin = ""  
 for i in range(len(s)):  
 bin = bin + mp[s[i]]  
 return bin  
  
  
# Binary to hexadecimal conversion  
  
  
def bin2hex(s):  
 mp = {"0000": '0',  
 "0001": '1',  
 "0010": '2',  
 "0011": '3',  
 "0100": '4',  
 "0101": '5',  
 "0110": '6',  
 "0111": '7',  
 "1000": '8',  
 "1001": '9',  
 "1010": 'A',  
 "1011": 'B',  
 "1100": 'C',  
 "1101": 'D',  
 "1110": 'E',  
 "1111": 'F'}  
 hex = ""  
 for i in range(0, len(s), 4):  
 ch = ""  
 ch = ch + s[i]  
 ch = ch + s[i + 1]  
 ch = ch + s[i + 2]  
 ch = ch + s[i + 3]  
 hex = hex + mp[ch]  
  
 return hex  
  
  
# Binary to decimal conversion  
  
  
def bin2dec(binary):  
 binary1 = binary  
 decimal, i, n = 0, 0, 0  
 while (binary != 0):  
 dec = binary % 10  
 decimal = decimal + dec \* pow(2, i)  
 binary = binary // 10  
 i += 1  
 return decimal  
  
  
# Decimal to binary conversion  
  
  
def dec2bin(num):  
 res = bin(num).replace("0b", "")  
 if len(res) % 4 != 0:  
 div = len(res) / 4  
 div = int(div)  
 counter = (4 \* (div + 1)) - len(res)  
 for i in range(0, counter):  
 res = '0' + res  
 return res  
  
  
# Permute function to rearrange the bits  
  
  
def permute(k, arr, n):  
 permutation = ""  
 for i in range(0, n):  
 permutation = permutation + k[arr[i] - 1]  
 return permutation  
  
  
# shifting the bits towards left by nth shifts  
  
  
def shift\_left(k, nth\_shifts):  
 s = ""  
 for i in range(nth\_shifts):  
 for j in range(1, len(k)):  
 s = s + k[j]  
 s = s + k[0]  
 k = s  
 s = ""  
 return k  
  
  
# calculating xow of two strings of binary number a and b  
  
  
def xor(a, b):  
 ans = ""  
 for i in range(len(a)):  
 if a[i] == b[i]:  
 ans = ans + "0"  
 else:  
 ans = ans + "1"  
 return ans  
  
  
# Table of Position of 64 bits at initial level: Initial Permutation Table  
initial\_perm = [58, 50, 42, 34, 26, 18, 10, 2,  
 60, 52, 44, 36, 28, 20, 12, 4,  
 62, 54, 46, 38, 30, 22, 14, 6,  
 64, 56, 48, 40, 32, 24, 16, 8,  
 57, 49, 41, 33, 25, 17, 9, 1,  
 59, 51, 43, 35, 27, 19, 11, 3,  
 61, 53, 45, 37, 29, 21, 13, 5,  
 63, 55, 47, 39, 31, 23, 15, 7]  
  
# Expansion D-box Table  
exp\_d = [32, 1, 2, 3, 4, 5, 4, 5,  
 6, 7, 8, 9, 8, 9, 10, 11,  
 12, 13, 12, 13, 14, 15, 16, 17,  
 16, 17, 18, 19, 20, 21, 20, 21,  
 22, 23, 24, 25, 24, 25, 26, 27,  
 28, 29, 28, 29, 30, 31, 32, 1]  
  
# Straight Permutation Table  
per = [16, 7, 20, 21,  
 29, 12, 28, 17,  
 1, 15, 23, 26,  
 5, 18, 31, 10,  
 2, 8, 24, 14,  
 32, 27, 3, 9,  
 19, 13, 30, 6,  
 22, 11, 4, 25]  
  
# S-box Table  
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],  
 [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],  
 [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],  
 [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],  
  
 [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],  
 [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],  
 [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],  
 [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],  
  
 [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],  
 [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],  
 [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],  
 [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],  
  
 [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],  
 [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],  
 [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],  
 [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],  
  
 [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],  
 [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],  
 [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],  
 [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],  
  
 [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],  
 [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],  
 [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],  
 [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],  
  
 [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],  
 [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],  
 [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],  
 [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],  
  
 [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],  
 [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],  
 [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],  
 [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]  
  
# Final Permutation Table  
final\_perm = [40, 8, 48, 16, 56, 24, 64, 32,  
 39, 7, 47, 15, 55, 23, 63, 31,  
 38, 6, 46, 14, 54, 22, 62, 30,  
 37, 5, 45, 13, 53, 21, 61, 29,  
 36, 4, 44, 12, 52, 20, 60, 28,  
 35, 3, 43, 11, 51, 19, 59, 27,  
 34, 2, 42, 10, 50, 18, 58, 26,  
 33, 1, 41, 9, 49, 17, 57, 25]  
  
  
def encrypt(pt, rkb, rk):  
 pt = hex2bin(pt)  
  
 # Initial Permutation  
 pt = permute(pt, initial\_perm, 64)  
 print("After initial permutation", bin2hex(pt))  
  
 # Splitting  
 left = pt[0:32]  
 right = pt[32:64]  
 for i in range(0, 16):  
 # Expansion D-box: Expanding the 32 bits data into 48 bits  
 right\_expanded = permute(right, exp\_d, 48)  
  
 # XOR RoundKey[i] and right\_expanded  
 xor\_x = xor(right\_expanded, rkb[i])  
  
 # S-boxex: substituting the value from s-box table by calculating row and column  
 sbox\_str = ""  
 for j in range(0, 8):  
 row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))  
 col = bin2dec(  
 int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))  
 val = sbox[j][row][col]  
 sbox\_str = sbox\_str + dec2bin(val)  
  
 # Straight D-box: After substituting rearranging the bits  
 sbox\_str = permute(sbox\_str, per, 32)  
  
 # XOR left and sbox\_str  
 result = xor(left, sbox\_str)  
 left = result  
  
 # Swapper  
 if (i != 15):  
 left, right = right, left  
 print("Round ", i + 1, " ", bin2hex(left),  
 " ", bin2hex(right), " ", rk[i])  
  
 # Combination  
 combine = left + right  
  
 # Final permutation: final rearranging of bits to get cipher text  
 cipher\_text = permute(combine, final\_perm, 64)  
 return cipher\_text  
  
  
pt = "123456ABCD132536"  
key = "AABB09182736CCDD"  
  
# Key generation  
# --hex to binary  
key = hex2bin(key)  
  
# --parity bit drop table  
keyp = [57, 49, 41, 33, 25, 17, 9,  
 1, 58, 50, 42, 34, 26, 18,  
 10, 2, 59, 51, 43, 35, 27,  
 19, 11, 3, 60, 52, 44, 36,  
 63, 55, 47, 39, 31, 23, 15,  
 7, 62, 54, 46, 38, 30, 22,  
 14, 6, 61, 53, 45, 37, 29,  
 21, 13, 5, 28, 20, 12, 4]  
  
# getting 56 bit key from 64 bit using the parity bits  
key = permute(key, keyp, 56)  
  
# Number of bit shifts  
shift\_table = [1, 1, 2, 2,  
 2, 2, 2, 2,  
 1, 2, 2, 2,  
 2, 2, 2, 1]  
  
# Key- Compression Table : Compression of key from 56 bits to 48 bits  
key\_comp = [14, 17, 11, 24, 1, 5,  
 3, 28, 15, 6, 21, 10,  
 23, 19, 12, 4, 26, 8,  
 16, 7, 27, 20, 13, 2,  
 41, 52, 31, 37, 47, 55,  
 30, 40, 51, 45, 33, 48,  
 44, 49, 39, 56, 34, 53,  
 46, 42, 50, 36, 29, 32]  
  
# Splitting  
left = key[0:28] # rkb for RoundKeys in binary  
right = key[28:56] # rk for RoundKeys in hexadecimal  
  
rkb = []  
rk = []  
for i in range(0, 16):  
 # Shifting the bits by nth shifts by checking from shift table  
 left = shift\_left(left, shift\_table[i])  
 right = shift\_left(right, shift\_table[i])  
  
 # Combination of left and right string  
 combine\_str = left + right  
  
 # Compression of key from 56 to 48 bits  
 round\_key = permute(combine\_str, key\_comp, 48)  
  
 rkb.append(round\_key)  
 rk.append(bin2hex(round\_key))  
  
print("Encryption")  
cipher\_text = bin2hex(encrypt(pt, rkb, rk))  
print("Cipher Text : ", cipher\_text)  
  
print("Decryption")  
rkb\_rev = rkb[::-1]  
rk\_rev = rk[::-1]  
text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))  
print("Plain Text : ", text)





34. Write a Python program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.primitives import padding  
from cryptography.hazmat.backends import default\_backend  
import os  
  
  
def encrypt\_ecb(plaintext, key):  
 # Create a cipher object with AES algorithm and ECB mode  
 cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.AES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def encrypt\_cbc(plaintext, key, iv):  
 # Create a cipher object with AES algorithm and CBC mode  
 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Create a padder with the block size of the cipher  
 padder = padding.PKCS7(algorithms.AES.block\_size).padder()  
  
 # Pad the plaintext  
 padded\_plaintext = padder.update(plaintext) + padder.finalize()  
  
 # Encrypt the padded plaintext  
 ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
def encrypt\_cfb(plaintext, key, iv):  
 # Create a cipher object with AES algorithm and CFB mode  
 cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())  
  
 encryptor = cipher.encryptor()  
  
 # Encrypt the plaintext  
 ciphertext = encryptor.update(plaintext) + encryptor.finalize()  
  
 # Return the ciphertext  
 return ciphertext  
  
  
# Example usage  
key = os.urandom(16) # Generate a random 16-byte AES key  
iv = os.urandom(algorithms.AES.block\_size // 8) # Generate a random IV with the block size of AES  
plaintext = b'This is a test message'  
  
# Encryption in ECB mode  
ciphertext\_ecb = encrypt\_ecb(plaintext, key)  
print("ECB Ciphertext:", ciphertext\_ecb)  
  
# Encryption in CBC mode  
ciphertext\_cbc = encrypt\_cbc(plaintext, key, iv)  
print("CBC Ciphertext:", ciphertext\_cbc)  
  
# Encryption in CFB mode  
ciphertext\_cfb = encrypt\_cfb(plaintext, key, iv)  
print("CFB Ciphertext:", ciphertext\_cfb)



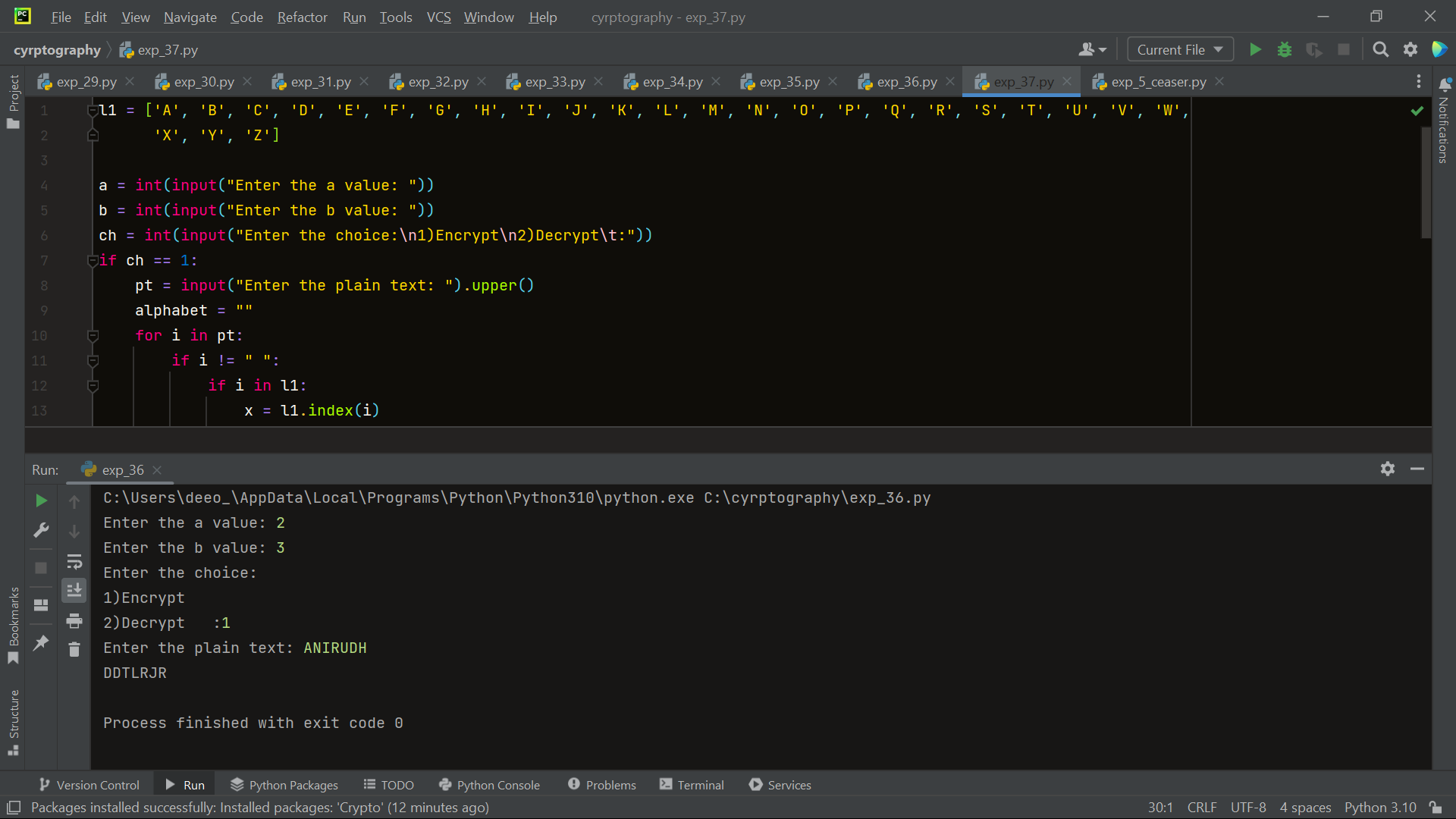
35. Write a Python program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.

import random  
  
  
def encrypt(plaintext, key):  
 ciphertext = ""  
 key\_length = len(key)  
  
 for i, char in enumerate(plaintext):  
 if char.isalpha():  
 char\_value = ord(char.upper()) - ord('A')  
 key\_value = key[i % key\_length]  
 encrypted\_value = (char\_value + key\_value) % 26  
 encrypted\_char = chr(encrypted\_value + ord('A'))  
 ciphertext += encrypted\_char  
 else:  
 ciphertext += char  
  
 return ciphertext  
  
  
def decrypt(ciphertext, key):  
 plaintext = ""  
 key\_length = len(key)  
  
 for i, char in enumerate(ciphertext):  
 if char.isalpha():  
 char\_value = ord(char.upper()) - ord('A')  
 key\_value = key[i % key\_length]  
 decrypted\_value = (char\_value - key\_value) % 26  
 decrypted\_char = chr(decrypted\_value + ord('A'))  
 plaintext += decrypted\_char  
 else:  
 plaintext += char  
  
 return plaintext  
  
  
# Example usage  
plaintext = "HELLO WORLD"  
key = [3, 19, 5] # Example key: random numbers between 0 and 26  
  
ciphertext = encrypt(plaintext, key)  
print("Ciphertext:", ciphertext)  
  
decrypted\_text = decrypt(ciphertext, key)  
print("Decrypted text:", decrypted\_text)



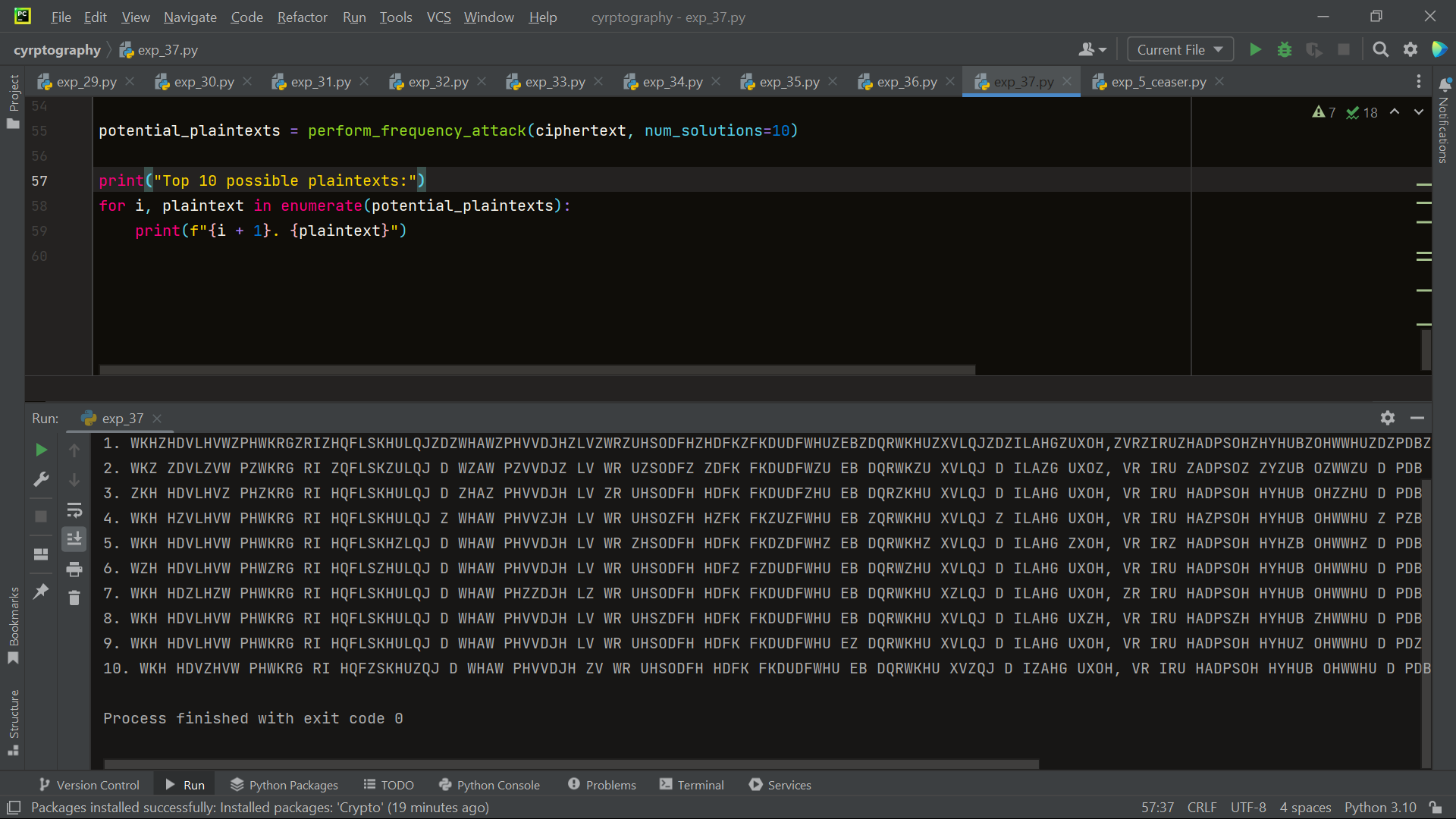
36. Write a Python program for Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.

l1 = ['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']  
  
a = int(input("Enter the a value: "))  
b = int(input("Enter the b value: "))  
ch = int(input("Enter the choice:\n1)Encrypt\n2)Decrypt\t:"))  
if ch == 1:  
 pt = input("Enter the plain text: ").upper()  
 alphabet = ""  
 for i in pt:  
 if i != " ":  
 if i in l1:  
 x = l1.index(i)  
 new\_index = int(((a \* x) + b) % 26)  
 alphabet = alphabet + l1[new\_index]  
  
 print(alphabet)  
  
if ch == 2:  
 ct = input("Enter the cipher text: ").upper()  
 alphabet = ""  
 for i in ct:  
 if i != " ":  
 if i in l1:  
 x = l1.index(i)  
 new\_index = int(((x - b) / a) % 26)  
 alphabet = alphabet + l1[new\_index]  
  
 print(alphabet)



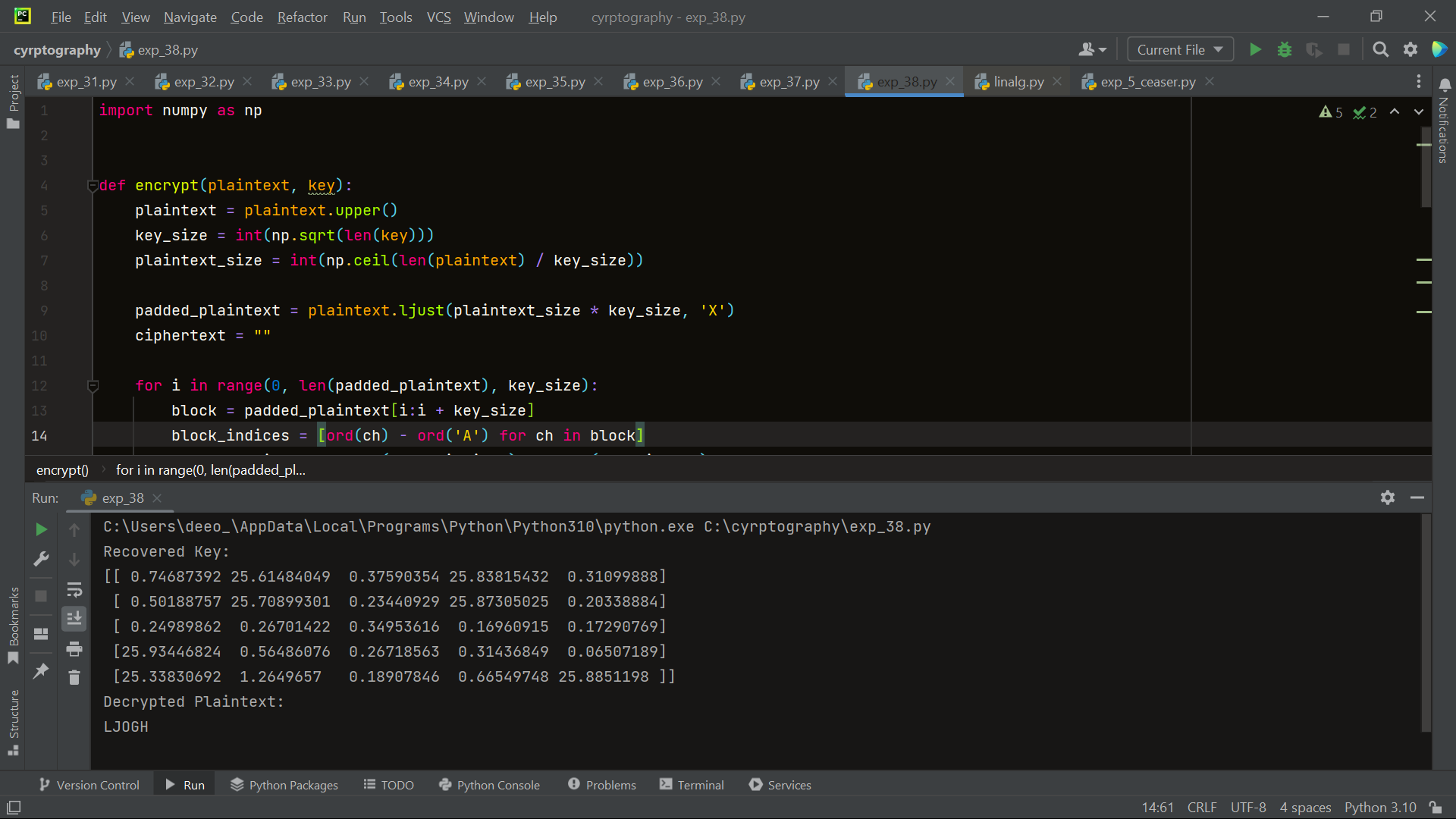
37. Write a Python program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

import string  
import collections  
  
  
def calculate\_frequency(text):  
 # Count the frequency of each letter in the text  
 frequency = collections.Counter(text)  
  
 # Calculate the relative frequency of each letter  
 total = sum(frequency.values())  
 relative\_frequency = {letter: count / total for letter, count in frequency.items()}  
  
 return relative\_frequency  
  
  
def decrypt(ciphertext, key):  
 # Create a decryption key using the given substitution key  
 decryption\_key = str.maketrans(key, string.ascii\_uppercase)  
  
 # Decrypt the ciphertext using the decryption key  
 plaintext = ciphertext.translate(decryption\_key)  
  
 return plaintext  
  
  
def perform\_frequency\_attack(ciphertext, num\_solutions=10):  
 # Calculate the letter frequency of the ciphertext  
 ciphertext\_frequency = calculate\_frequency(ciphertext)  
  
 # Sort the letter frequencies in descending order  
 sorted\_frequency = sorted(ciphertext\_frequency.items(), key=lambda x: x[1], reverse=True)  
  
 # Generate a list of potential plaintexts with different substitution keys  
 potential\_plaintexts = []  
  
 for i in range(num\_solutions):  
 # Extract the most likely ciphertext letter  
 ciphertext\_letter = sorted\_frequency[i][0]  
  
 # Create a substitution key based on the ciphertext letter  
 key = ciphertext\_letter \* len(string.ascii\_uppercase)  
  
 # Decrypt the ciphertext using the substitution key  
 plaintext = decrypt(ciphertext, key)  
  
 # Add the potential plaintext to the list  
 potential\_plaintexts.append(plaintext)  
  
 return potential\_plaintexts  
  
  
# Example usage  
ciphertext = "WKH HDVLHVW PHWKRG RI HQFLSKHULQJ D WHAW PHVVDJH LV WR UHSODFH HDFK FKDUDFWHU EB DQRWKHU XVLQJ D ILAHG UXOH, VR IRU HADPSOH HYHUB OHWWHU D PDB EH UHSODFHG EB WKH OHWWHU DQG HYHUB OHWWHU E EB WKH PDB."  
  
potential\_plaintexts = perform\_frequency\_attack(ciphertext, num\_solutions=10)  
  
print("Top 10 possible plaintexts:")  
for i, plaintext in enumerate(potential\_plaintexts):  
 print(f"{i + 1}. {plaintext}")



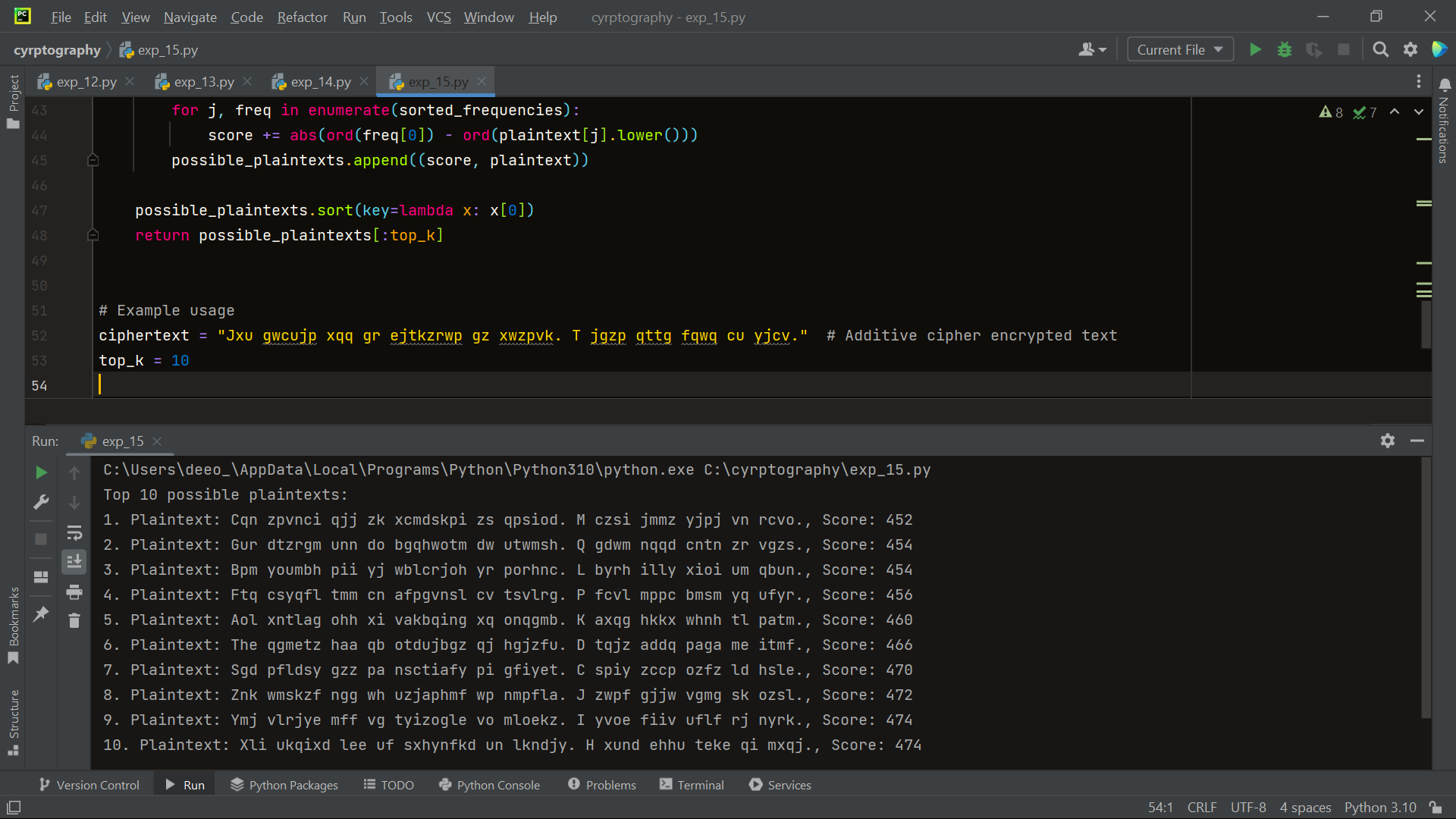
38. Write a Python program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext– ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted. Implement in Python programming.

import numpy as np  
  
  
def encrypt(plaintext, key):  
 plaintext = plaintext.upper()  
 key\_size = int(np.sqrt(len(key)))  
 plaintext\_size = int(np.ceil(len(plaintext) / key\_size))  
  
 padded\_plaintext = plaintext.ljust(plaintext\_size \* key\_size, 'X')  
 ciphertext = ""  
  
 for i in range(0, len(padded\_plaintext), key\_size):  
 block = padded\_plaintext[i:i + key\_size]  
 block\_indices = [ord(ch) - ord('A') for ch in block]  
 block\_matrix = np.array(block\_indices).reshape(key\_size, 1)  
  
 encrypted\_block\_matrix = np.dot(key, block\_matrix) % 26  
 encrypted\_block = "".join(chr(index + ord('A')) for index in encrypted\_block\_matrix.flatten())  
 ciphertext += encrypted\_block  
  
 return ciphertext  
  
  
def known\_plaintext\_attack(plaintexts, ciphertexts):  
 plaintext\_matrix = np.array([list(map(lambda x: ord(x) - ord('A'), plaintext)) for plaintext in plaintexts])  
 ciphertext\_matrix = np.array([list(map(lambda x: ord(x) - ord('A'), ciphertext)) for ciphertext in ciphertexts])  
  
 key = np.dot(np.linalg.pinv(plaintext\_matrix), ciphertext\_matrix) % 26  
  
 return key  
  
  
def decrypt(ciphertext, key):  
 ciphertext\_matrix = np.array(list(map(lambda x: ord(x) - ord('A'), ciphertext)))  
 decrypted\_matrix = np.dot(ciphertext\_matrix, key) % 26  
 decrypted\_text = ''.join(chr(int(val) + ord('A')) for val in decrypted\_matrix)  
 return decrypted\_text  
  
  
# Example usage  
plaintexts = ["HELLO", "WORLD"]  
ciphertexts = ["AXNNE", "ZCVDN"]  
key = known\_plaintext\_attack(plaintexts, ciphertexts)  
  
print("Recovered Key:")  
print(key)  
  
# Test the decryption with the recovered key  
recovered\_plaintext = decrypt(ciphertexts[0], key)  
print("Decrypted Plaintext:")  
print(recovered\_plaintext)



1. Write a Python program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

import string  
  
  
def get\_letter\_frequencies(ciphertext):  
 frequencies = {letter: 0 for letter in string.ascii\_lowercase}  
 total\_letters = 0  
  
 for letter in ciphertext:  
 if letter.isalpha():  
 frequencies[letter.lower()] += 1  
 total\_letters += 1  
  
 for letter in frequencies:  
 frequencies[letter] /= total\_letters  
  
 return frequencies  
  
  
def decrypt(ciphertext, key):  
 plaintext = ""  
 for letter in ciphertext:  
 if letter.isalpha():  
 is\_upper = letter.isupper()  
 letter = letter.lower()  
 decrypted\_letter = chr((ord(letter) - ord('a') - key) % 26 + ord('a'))  
 if is\_upper:  
 decrypted\_letter = decrypted\_letter.upper()  
 plaintext += decrypted\_letter  
 else:  
 plaintext += letter  
 return plaintext  
  
  
def letter\_frequency\_attack(ciphertext, top\_k=10):  
 frequencies = get\_letter\_frequencies(ciphertext)  
 sorted\_frequencies = sorted(frequencies.items(), key=lambda x: x[1], reverse=True)  
 possible\_plaintexts = []  
  
 for i in range(26):  
 key = i  
 plaintext = decrypt(ciphertext, key)  
 score = 0  
 for j, freq in enumerate(sorted\_frequencies):  
 score += abs(ord(freq[0]) - ord(plaintext[j].lower()))  
 possible\_plaintexts.append((score, plaintext))  
  
 possible\_plaintexts.sort(key=lambda x: x[0])  
 return possible\_plaintexts[:top\_k]  
  
  
# Example usage  
ciphertext = "Jxu gwcujp xqq gr ejtkzrwp gz xwzpvk. T jgzp qttg fqwq cu yjcv." # Additive cipher encrypted text  
top\_k = 10  
  
results = letter\_frequency\_attack(ciphertext, top\_k)  
  
print(f"Top {top\_k} possible plaintexts:")  
for i, result in enumerate(results, 1):  
 score, plaintext = result  
 print(f"{i}. Plaintext: {plaintext}, Score: {score}")



40 . Write a Python program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

import string  
import collections  
  
  
def calculate\_frequency(text):  
 # Count the frequency of each letter in the text  
 frequency = collections.Counter(text)  
  
 # Calculate the relative frequency of each letter  
 total = sum(frequency.values())  
 relative\_frequency = {letter: count / total for letter, count in frequency.items()}  
  
 return relative\_frequency  
  
  
def decrypt(ciphertext, key):  
 # Create a decryption key using the given substitution key  
 decryption\_key = str.maketrans(key, string.ascii\_uppercase)  
  
 # Decrypt the ciphertext using the decryption key  
 plaintext = ciphertext.translate(decryption\_key)  
  
 return plaintext  
  
  
def perform\_frequency\_attack(ciphertext, num\_solutions=10):  
 # Calculate the letter frequency of the ciphertext  
 ciphertext\_frequency = calculate\_frequency(ciphertext)  
  
 # Sort the letter frequencies in descending order  
 sorted\_frequency = sorted(ciphertext\_frequency.items(), key=lambda x: x[1], reverse=True)  
  
 # Generate a list of potential plaintexts with different substitution keys  
 potential\_plaintexts = []  
  
 for i in range(num\_solutions):  
 # Extract the most likely ciphertext letter  
 ciphertext\_letter = sorted\_frequency[i][0]  
  
 # Create a substitution key based on the ciphertext letter  
 key = ciphertext\_letter \* len(string.ascii\_uppercase)  
  
 # Decrypt the ciphertext using the substitution key  
 plaintext = decrypt(ciphertext, key)  
  
 # Add the potential plaintext to the list  
 potential\_plaintexts.append(plaintext)  
  
 return potential\_plaintexts  
  
  
# Example usage  
ciphertext = "WKH HDVLHVW PHWKRG RI HQFLSKHULQJ D WHAW PHVVDJH LV WR UHSODFH HDFK FKDUDFWHU EB DQRWKHU XVLQJ D ILAHG UXOH, VR IRU HADPSOH HYHUB OHWWHU D PDB EH UHSODFHG EB WKH OHWWHU DQG HYHUB OHWWHU E EB WKH PDB."  
  
potential\_plaintexts = perform\_frequency\_attack(ciphertext, num\_solutions=10)  
  
print("Top 10 possible plaintexts:")  
for i, plaintext in enumerate(potential\_plaintexts):  
 print(f"{i + 1}. {plaintext}")

