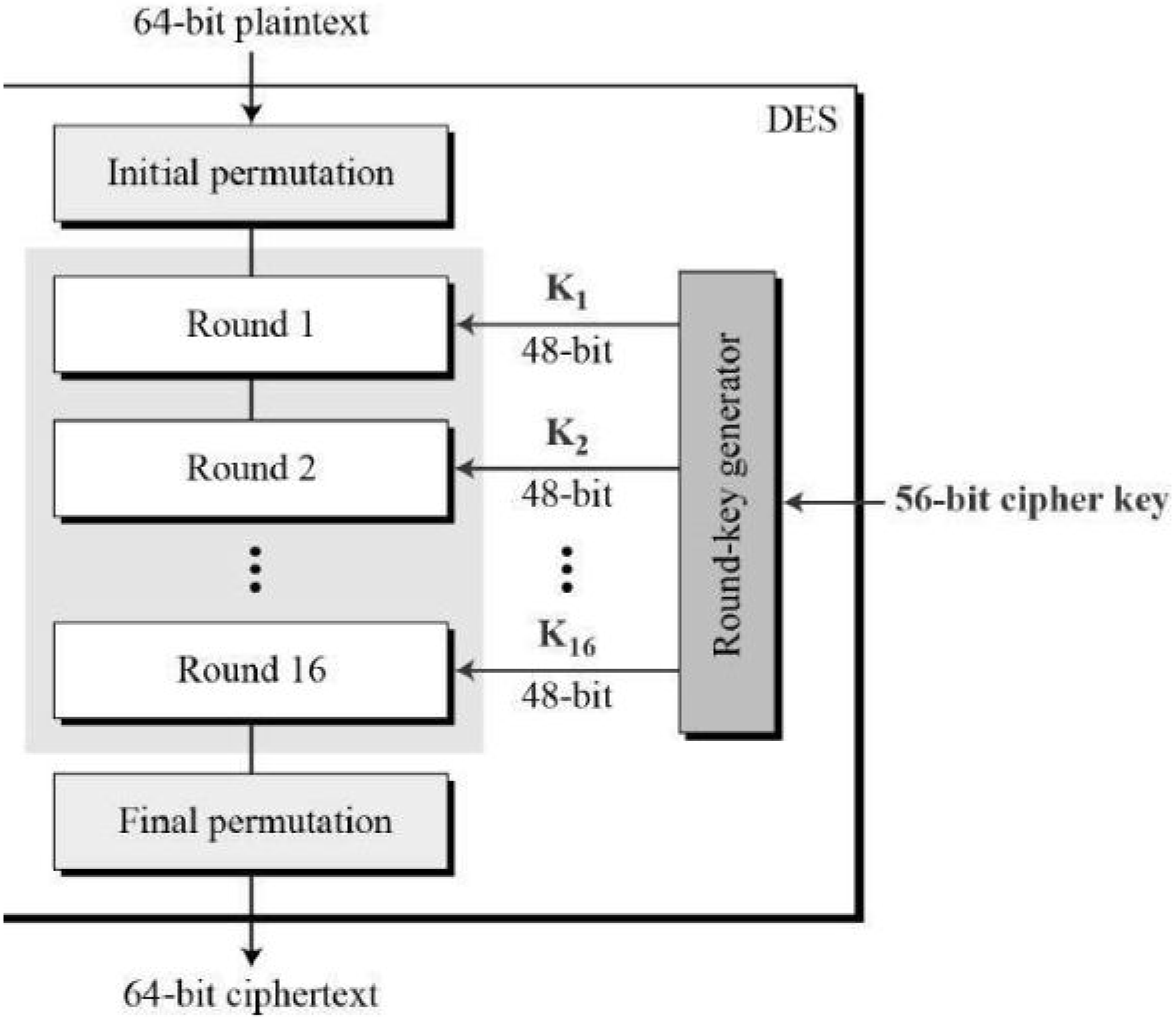
REPORT

About DES:

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

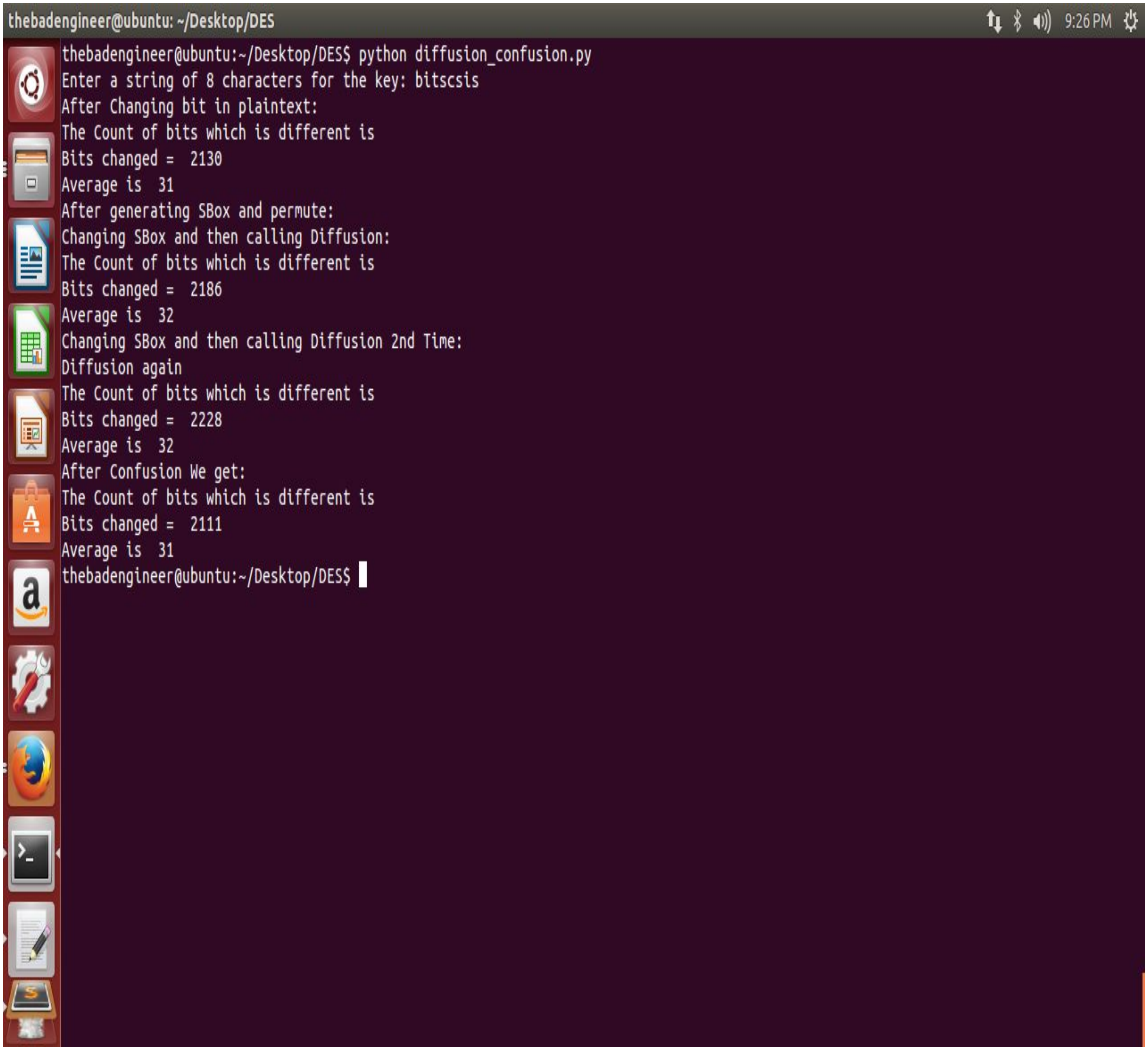
DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only)



Steps To Run:

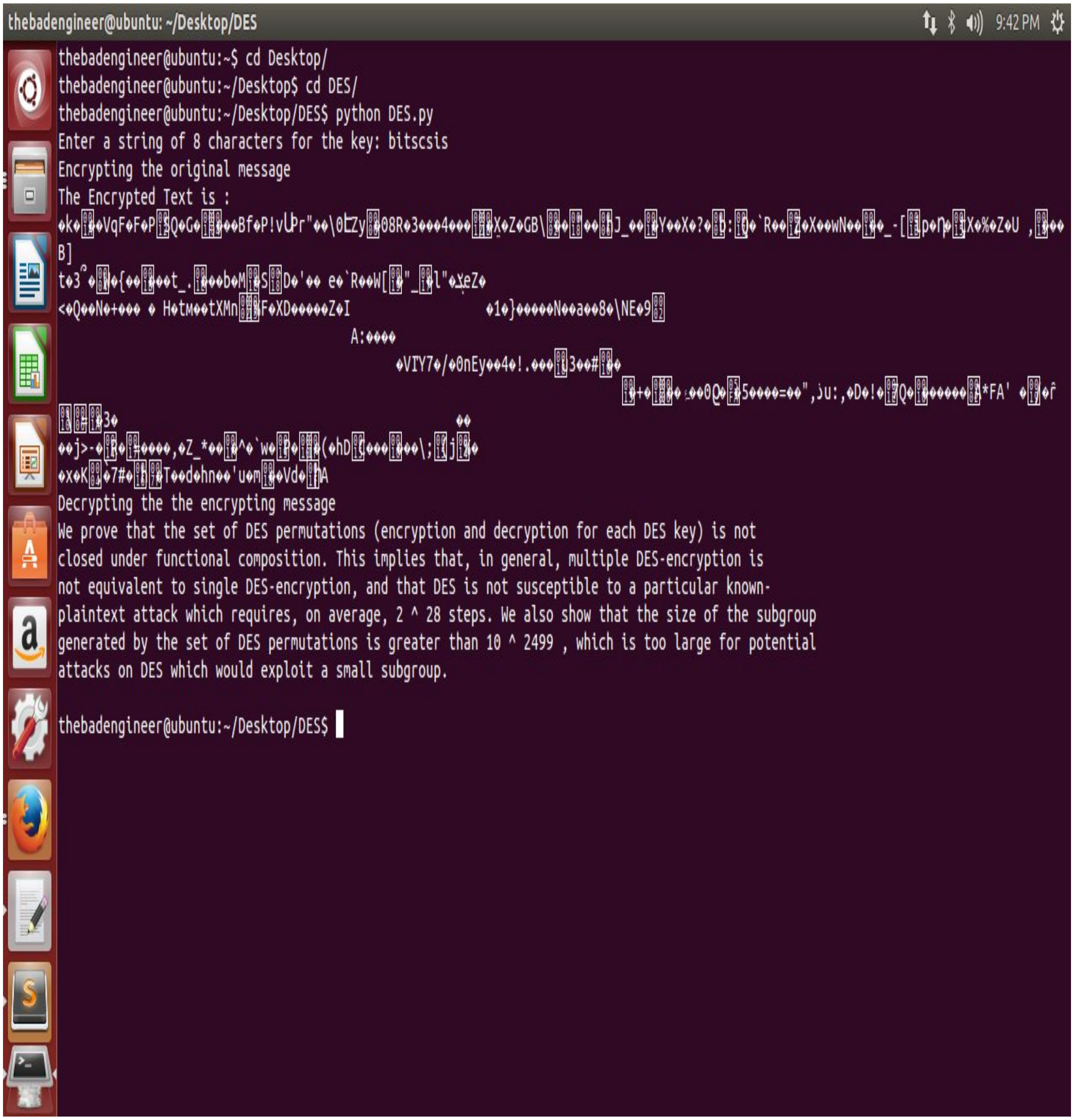
1.Run The DES.py and it will take Message.txt and generate two files Encrypted.txt and Decrypted.txt
2.After Running the Diffusion_Confusion.py It will Show the output the change bits after doing one bit change in the plaintext and change in one bit of key.

Average.py Screenshot(Output)



```
thebadengineer@ubuntu:~/Desktop/DES$ python diffusion_confusion.py
Enter a string of 8 characters for the key: bitscsis
After Changing bit in plaintext:
The Count of bits which is different is
Bits changed =  2130
Average is  31
After generating SBox and permute:
Changing SBox and then calling Diffusion:
The Count of bits which is different is
Bits changed =  2186
Average is  32
Changing SBox and then calling Diffusion 2nd Time:
Diffusion again
The Count of bits which is different is
Bits changed =  2228
Average is  32
After Confusion We get:
The Count of bits which is different is
Bits changed =  2111
Average is  31
thebadengineer@ubuntu:~/Desktop/DES$
```

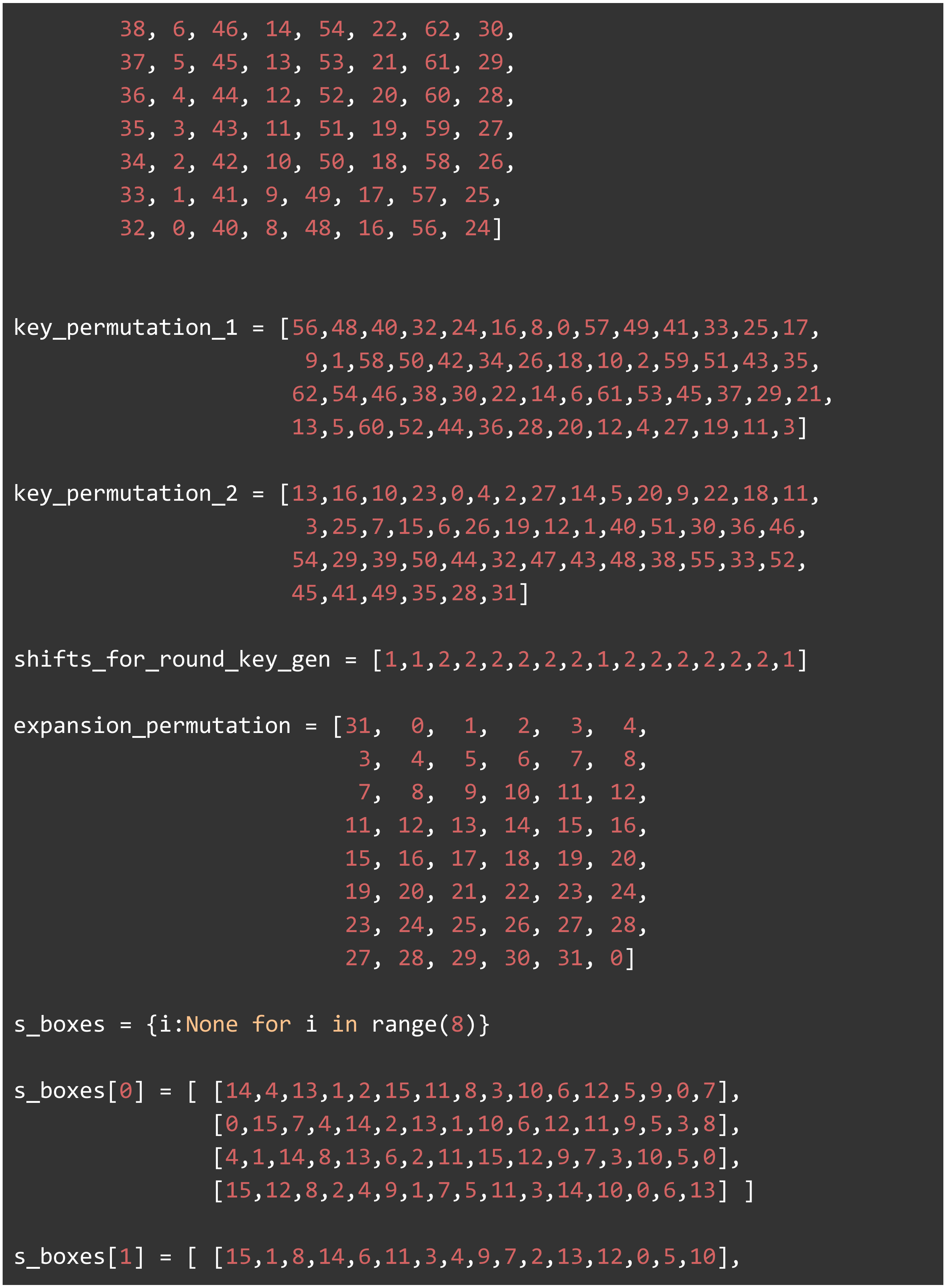
DES.py Screenshot



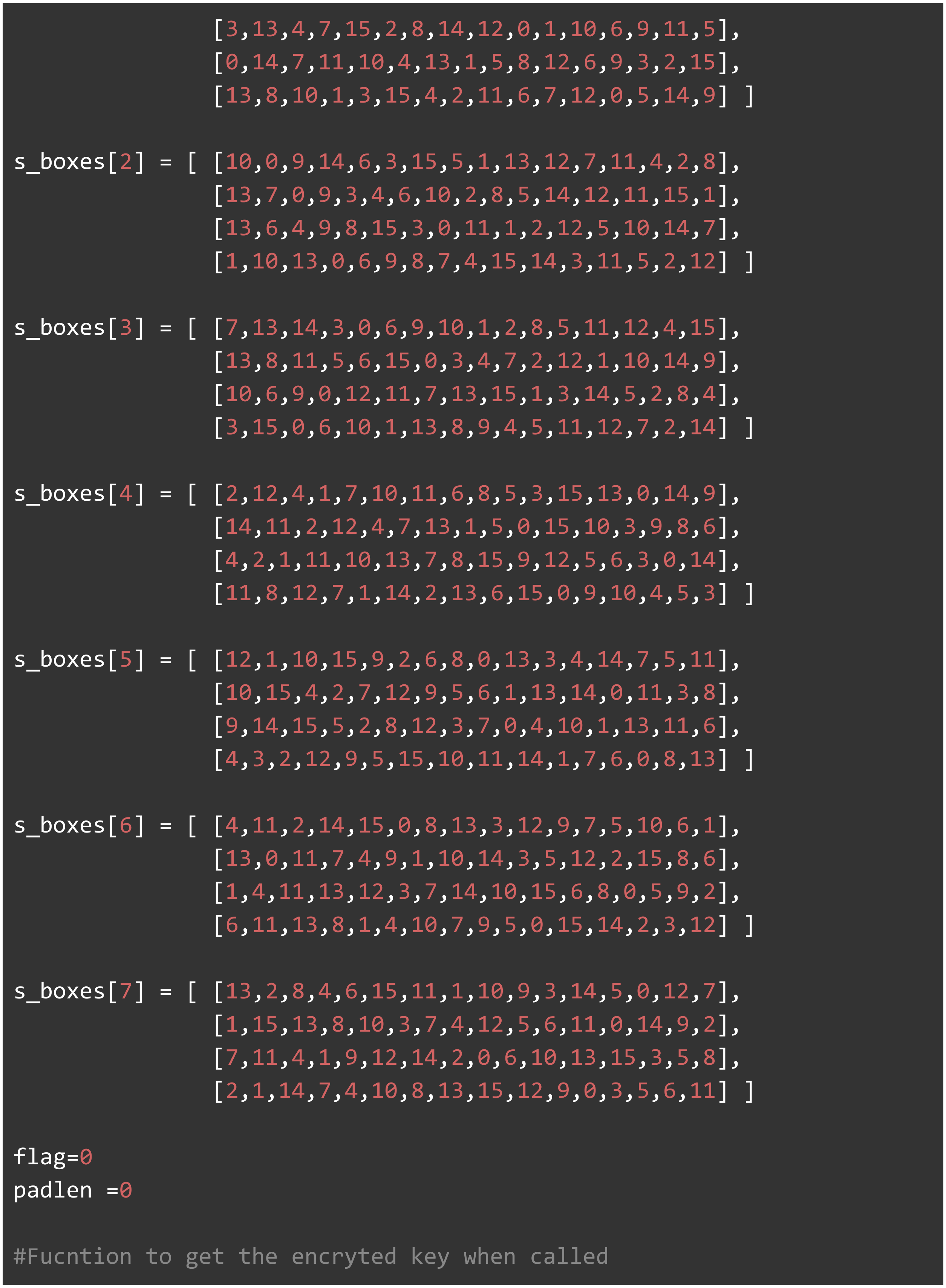
```
thebadengineer@ubuntu:~$ cd Desktop/
thebadengineer@ubuntu:~/Desktop$ cd DES/
thebadengineer@ubuntu:~/Desktop/DES$ python DES.py
Enter a string of 8 characters for the key: bitscsis
Encrypting the original message
The Encrypted Text is :
�k�▯�VqF�F�P▯Q�G�▯▯��Bf�P!vUPr"��\0ĿZy▯08R�3���4���▯▯X�Z�GB\▯�▯��▯J_��▯Y��X�?�▯:▯�`R��▯�X��wN��▯�_-[▯p�ɲ�▯X�%�Z�U ,▯��
B]
t�3̂�▯�{��▯��t_.▯��b�M▯S▯D�'�� e�`R��W[▯"_▯l"�ɤeZ�
<�Q��N�+��� � H�tM��tXMn▯▯%F�XD�����Z�I               �1�}�����N��a��8�\NE�9▯
                                A:����
                                     �VIY7�/�0nEy��4�!.���▯3��#▯�
                                                            ▯+�▯▯�꞉��0Q�▯5����=��",ȷu:,�D�!�▯Q�▯�����▯*FA' �▯�f̂
▯▯▯3�                                     ��
��j>-�▯�▯����,�Z_*��▯^�`w�▯�▯▯(�hD▯���▯��\;▯j▯�
�x�K▯�7#�▯▯T��d�hn��'u�m▯�Vd�▯A
Decrypting the the encrypting message
We prove that the set of DES permutations (encryption and decryption for each DES key) is not
closed under functional composition. This implies that, in general, multiple DES-encryption is
not equivalent to single DES-encryption, and that DES is not susceptible to a particular known-
plaintext attack which requires, on average, 2 ^ 28 steps. We also show that the size of the subgroup
generated by the set of DES permutations is greater than 10 ^ 2499 , which is too large for potential
attacks on DES which would exploit a small subgroup.

thebadengineer@ubuntu:~/Desktop/DES$ █
```

Code for DES.py:

```
#  -*- coding: utf-8 -*-

import sys
import BitVector

import sys
import os
import BitVector
import codecs

expansion_permutation = [31, 0, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8, 7, 8,
9, 10, 11, 12, 11, 12, 13, 14, 15, 16, 15, 16, 17, 18, 19, 20, 19,
20, 21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 0]

#padlen = 0
#Initial permut matrix for the datas
PI = [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,
      56, 48, 40, 32, 24, 16, 8, 0,
      58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6]
#Permut made after each SBox substitution for each round

P = [15, 6, 19, 20, 28, 12, 27, 16,
     0, 14, 22, 25, 4, 17, 30, 9,
     1, 7, 23, 13, 31, 26, 2, 8,
     18, 12, 29, 5, 21, 10, 3, 24]

#Final permut for datas after the 16 rounds
PI_1 = [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,
        32, 0, 40, 8, 48, 16, 56, 24]


key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31,  0]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [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] ]

s_boxes[1] = [ [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
```



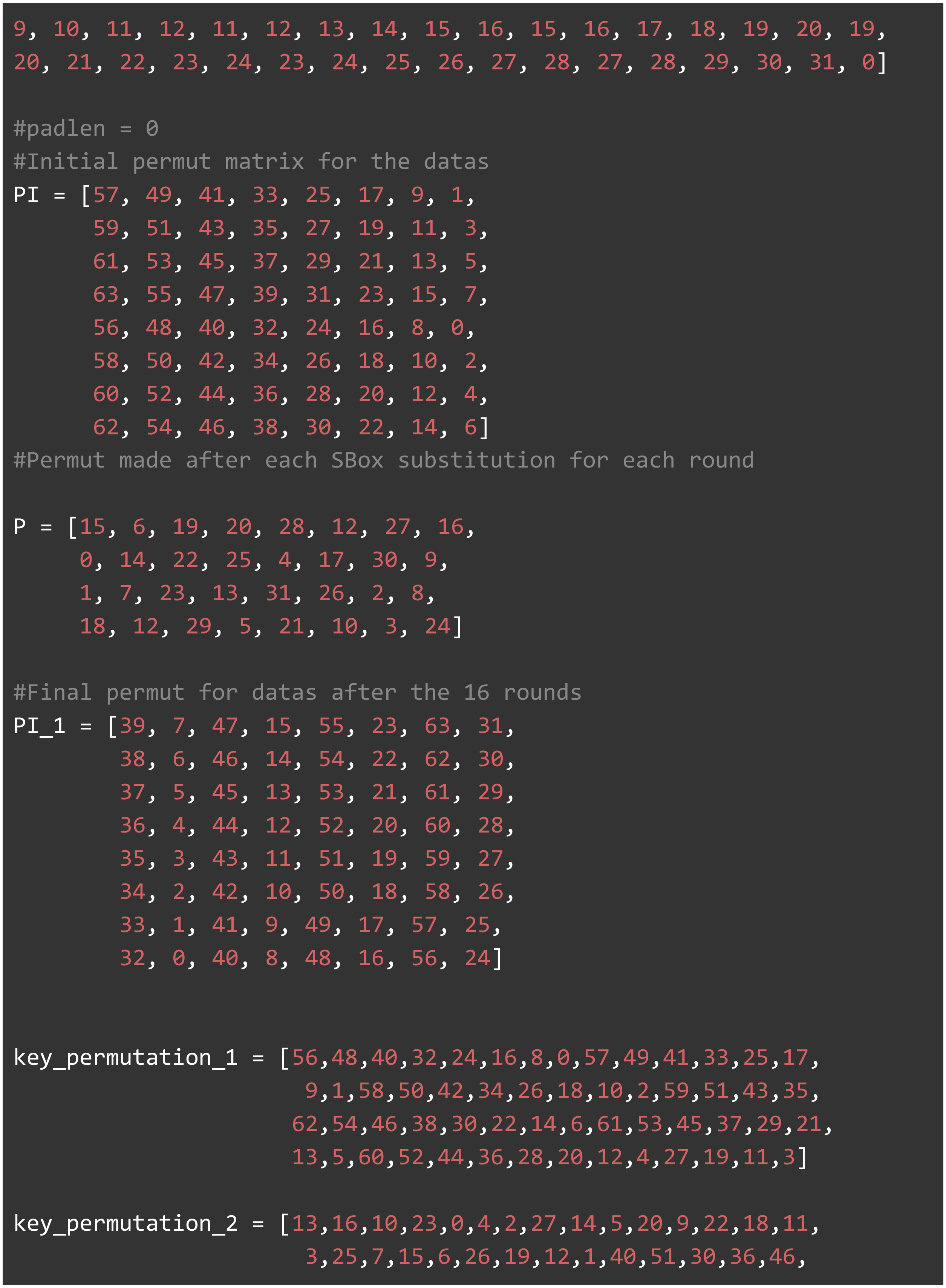
```
               [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
               [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
               [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9] ]

s_boxes[2] = [ [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] ]

s_boxes[3] = [ [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] ]

s_boxes[4] = [ [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] ]

s_boxes[5] = [ [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] ]

s_boxes[6] = [ [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] ]

s_boxes[7] = [ [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] ]

flag=0
padlen =0

#Fucntion to get the encryted key when called
```

```
def get_encryption_key():
    key = "bitscsis"
    while True:
        if sys.version_info[0] == 3:
            key = input("Enter a string of 8 characters for the key: ")
        else:
            key = raw_input("Enter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector.BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key


#Function to get round keys
def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector.BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector.BitVector(intVal =
s_boxes[sindex][row][column], size = 4)
    return output

 #Generating the key
key = get_encryption_key()


#Function to Decrypt the Encrypted file and get the original text
def decrypt():
    file = open("encrypted.txt", "r")        #Reading the encrypted
text
    s =file.read()
    file.close()
    round_key = generate_round_keys(key)         #Generating the
round keys fr decrypting
    bv = BitVector.BitVector(filename='encrypted.txt')
    string2=""
    while (bv.more_to_read):                     #While there is
something to read
        bitvec = bv.read_bits_from_file(64)      #Reading 8Bytes at
atime
        if len(bitvec) > 0:
            bitvec = bitvec.unpermute(PI_1)
            [LE, RE] = bitvec.divide_into_two()
            for count1 in range(15, -1,-1):             #Running the
loop backwards
                newRE = RE.permute(expansion_permutation)
                out_xor = (newRE ^ round_key[count1])
                s_box = substitute(out_xor)
                permute = s_box.permute(P)
                left = LE ^ permute                     #Swapping the
left part and right part
                LE = RE
                RE = left

            temp = LE                            #Last time Swapping
of LE and RE
            LE = RE
            RE = temp

            Sum = LE + RE
            Sum = BitVector.BitVector(bitstring=Sum)
            inv_Sum2 = Sum.unpermute(PI)
            string2 += inv_Sum2.get_bitvector_in_ascii()


    if flag ==1:                           #Checking if Remvinf
Padding is necesary
        string2= removepading(string2)
    file = open("decrypted.txt", "w")         #Writing to the file
named Decryted.txt
    file.write(string2)
    file.close()
    print(string2)


#Function to Encypt the The Message
def encrypt():
    round_key = generate_round_keys(key)     #Generating the Round
Keys
    bv = BitVector.BitVector(filename ='message.txt')    #Reading
from the message.txt
    length =0
    string1=""
    while (bv.more_to_read):                      #While there is
something to read
       bitvec = bv.read_bits_from_file( 64 )
       if len(bitvec) > 0:
            if len(bitvec)<64:                    #Checking if
Padding is needed or not
               bitvec=addpading(bitvec)
               bitvec=BitVector.BitVector(bitstring=bitvec)
            bitvec = bitvec.permute(PI)                #permute the
initial permutation
            [LE, RE] = bitvec.divide_into_two()
            for count1 in range(0,16):                   #Running the
cycle for 16 Rounds
                newRE = RE.permute(expansion_permutation)  #Permuting
the array
                out_xor = (newRE ^ round_key[count1])      #Xoring
for each round
                s_box = substitute(out_xor)                  #FBox
Function
                permute = s_box.permute(P)
                left = LE ^ permute
                LE=RE
                RE=left

            temp=LE                              #Inversing final
time
            LE=RE
            RE=temp

            Sum=LE+RE
            Sum=BitVector.BitVector(bitstring = Sum)
            inv_Sum = Sum.permute(PI_1)
            string1 += inv_Sum.get_bitvector_in_ascii()

    print("The Encrypted Text is :")
    print(string1)

    #Writing to the file named Encrypted.txt
    file = open("encrypted.txt", "w")
    file.write(string1)
    file.close()
```

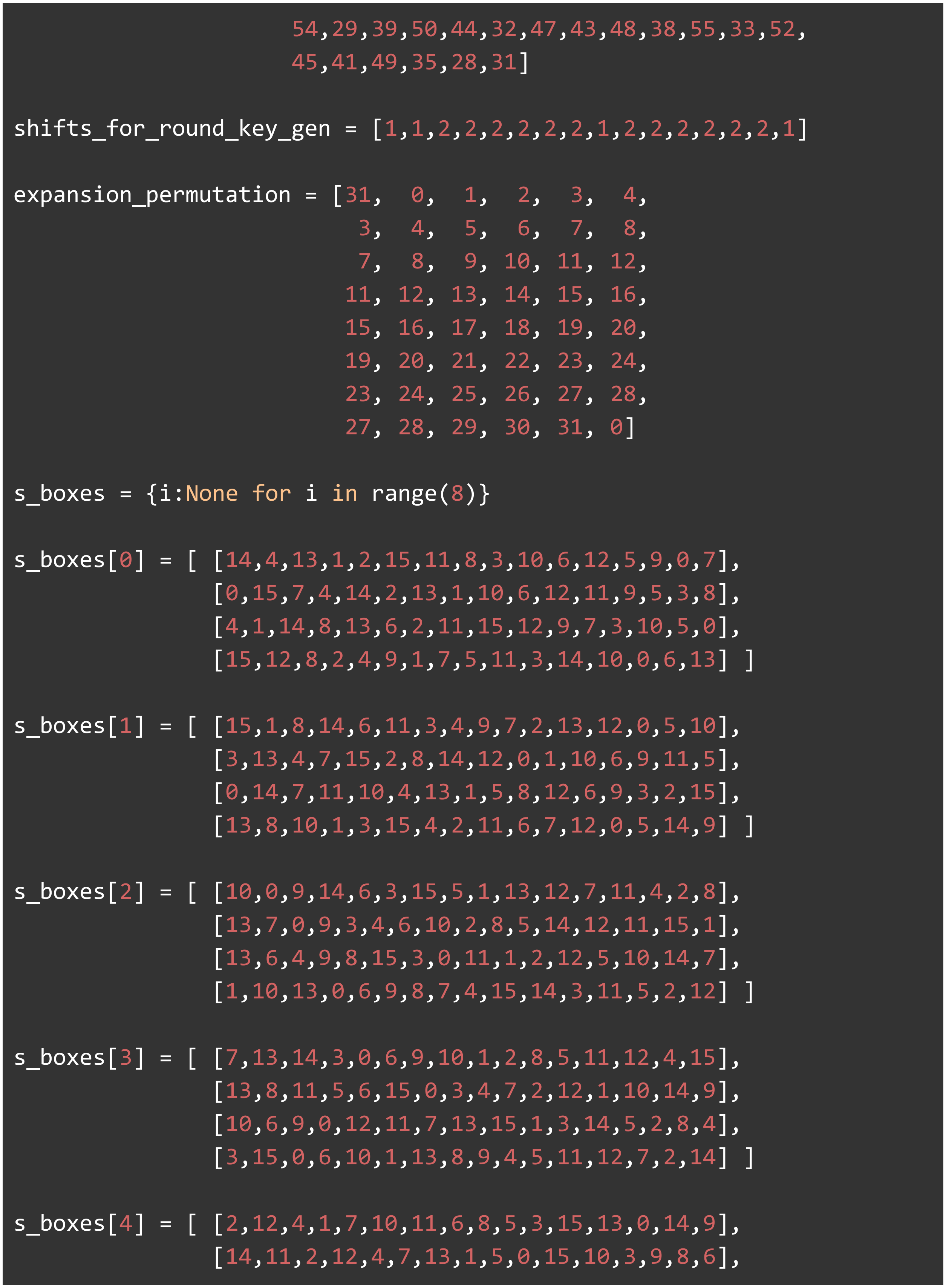
```
#Appending the bits to make it divisble by 8
def addpading(text):
    text = str(text)
    global padlen
    global flag
    flag=1
    padlen = 64-len(text)
    text+=padlen*"1"
    return text


#Removing the Padding Bits which was appended while ecrypting the 
message
def removepading(text):
    lengt = len(text) - (padlen/8)
    return text[0:int(lengt)]


print("Encrypting the original message")
encrypt()
print("Decrypting the the encrypting message")
decrypt()
```

Code For Average.py(Diffusion_Confusion):

```
#  -*- coding: utf-8 -*-

import sys
import BitVector

import sys
import os
import random
import BitVector
import codecs

expansion_permutation = [31, 0, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8, 7, 8,
```



```
9, 10, 11, 12, 11, 12, 13, 14, 15, 16, 15, 16, 17, 18, 19, 20, 19,
20, 21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 0]

#padlen = 0
#Initial permut matrix for the datas
PI = [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,
      56, 48, 40, 32, 24, 16, 8, 0,
      58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6]
#Permut made after each SBox substitution for each round

P = [15, 6, 19, 20, 28, 12, 27, 16,
     0, 14, 22, 25, 4, 17, 30, 9,
     1, 7, 23, 13, 31, 26, 2, 8,
     18, 12, 29, 5, 21, 10, 3, 24]

#Final permut for datas after the 16 rounds
PI_1 = [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,
        32, 0, 40, 8, 48, 16, 56, 24]


key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
```

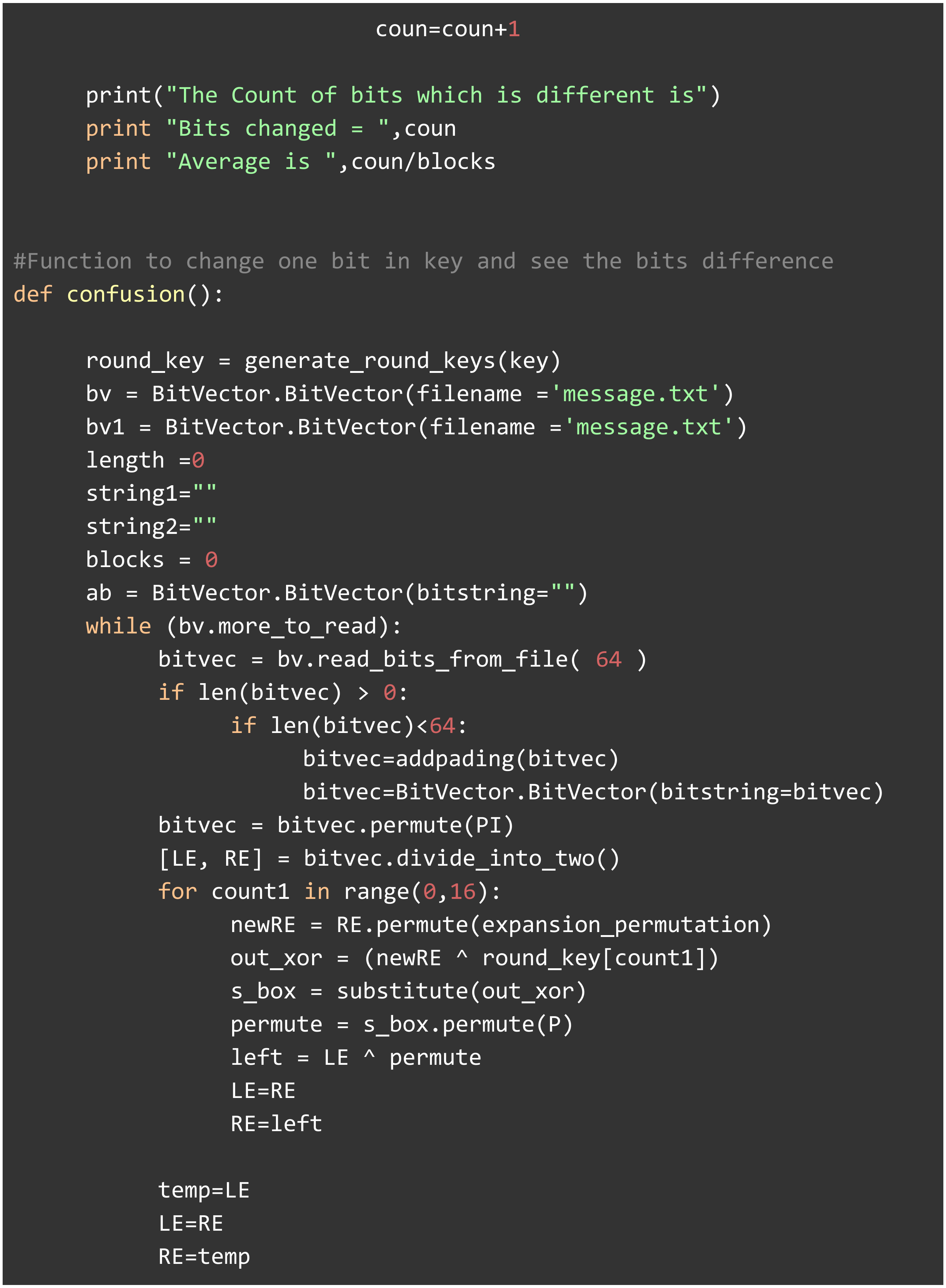


```
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [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] ]

s_boxes[1] = [ [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] ]

s_boxes[2] = [ [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] ]

s_boxes[3] = [ [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] ]

s_boxes[4] = [ [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] ]

s_boxes[5] = [ [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] ]

s_boxes[6] = [ [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] ]

s_boxes[7] = [ [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] ]

flag=0
padlen =0


def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes"
step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector.BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector.BitVector(intVal =
s_boxes[sindex][row][column], size = 4)
    return output
```

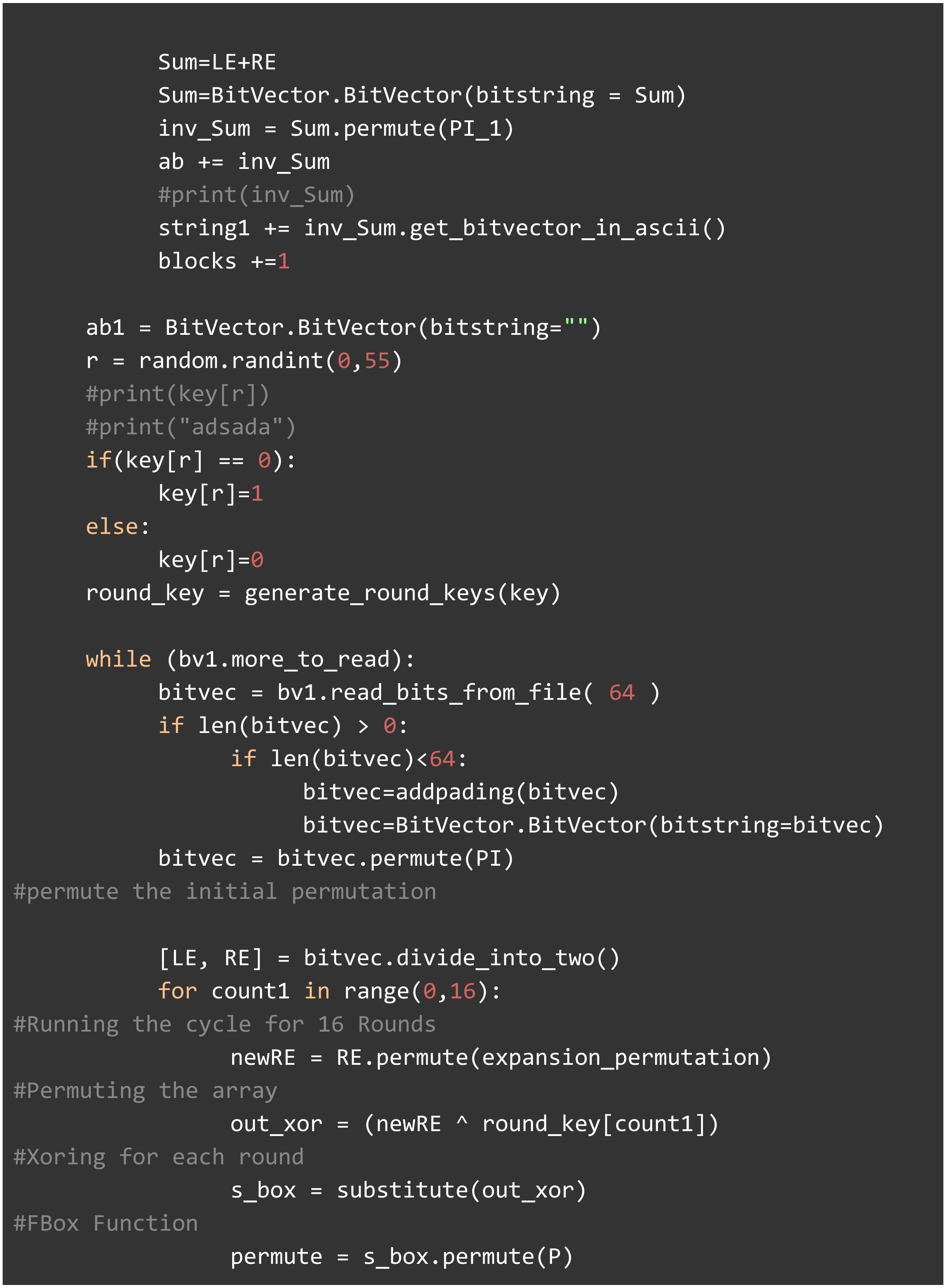
```
#Function to get encypted key
def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("Enter a string of 8 characters for the key:
")
        else:
            key = raw_input("Enter a string of 8 characters for the
key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try
again.\n")
            continue
        else:
            break
    key = BitVector.BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key


#function to get Round_keys
def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys

key=get_encryption_key()    #Getting the key
```

```
#Function to change one bit in the plaintext and seeing the cipher
text
def diffusion():

    round_key = generate_round_keys(key)
    bv = BitVector.BitVector(filename ='message.txt')
    bv1 = BitVector.BitVector(filename ='message.txt')

    length =0
    string1=""
    string2=""
    blocks = 0
    coun = 0


    while (bv.more_to_read):
        bitvec = bv.read_bits_from_file( 64 )        #Reading
8Bytes at a time
        if len(bitvec) > 0:
            if len(bitvec)<64:                             #Checking
if Padding is needed
                bitvec=addpading(bitvec)
                bitvec=BitVector.BitVector(bitstring=bitvec)
            bitvec = bitvec.permute(PI)
            [LE, RE] = bitvec.divide_into_two()
            for count1 in range(0,16):                    #Running
the loop for 16 Feistal Cycles
                newRE = RE.permute(expansion_permutation)
                out_xor = (newRE ^ round_key[count1])
                s_box = substitute(out_xor)
                permute = s_box.permute(P)
                left = LE ^ permute
                LE=RE
                RE=left

            temp=LE
            LE=RE
#Swapping for the last time after 16 rounds
```

```
                RE=temp

                Sum=LE+RE
                Sum=BitVector.BitVector(bitstring = Sum)
                inv_Sum = Sum.permute(PI_1)


                string1 += inv_Sum.get_bitvector_in_ascii()
                blocks +=1
                r = random.randint(0,63)
#Chaning the bits in the plaintext
                if(bitvec[r] == 0):
                    ka=1
                else:
                    ka=0
                bitvec[r] = ka
                [LE, RE] = bitvec.divide_into_two()
                for count1 in range(0,16):
                    newRE = RE.permute(expansion_permutation)
                    out_xor = (newRE ^ round_key[count1])
                    s_box = substitute(out_xor)
                    permute = s_box.permute(P)
                    left = LE ^ permute
                    LE=RE
                    RE=left

                temp=LE
                LE=RE
                RE=temp

                Sum=LE+RE
                Sum=BitVector.BitVector(bitstring = Sum)
                inv_Sum2 = Sum.permute(PI_1)

                #print(inv_Sum2)
                string2 += inv_Sum2.get_bitvector_in_ascii()
                for i in range(0,len(inv_Sum)):
                    if(inv_Sum[i]!=inv_Sum2[i]):
```



```
                    coun=coun+1

    print("The Count of bits which is different is")
    print "Bits changed = ",coun
    print "Average is ",coun/blocks


#Function to change one bit in key and see the bits difference
def confusion():

    round_key = generate_round_keys(key)
    bv = BitVector.BitVector(filename ='message.txt')
    bv1 = BitVector.BitVector(filename ='message.txt')
    length =0
    string1=""
    string2=""
    blocks = 0
    ab = BitVector.BitVector(bitstring="")
    while (bv.more_to_read):
        bitvec = bv.read_bits_from_file( 64 )
        if len(bitvec) > 0:
            if len(bitvec)<64:
                bitvec=addpading(bitvec)
                bitvec=BitVector.BitVector(bitstring=bitvec)
        bitvec = bitvec.permute(PI)
        [LE, RE] = bitvec.divide_into_two()
        for count1 in range(0,16):
            newRE = RE.permute(expansion_permutation)
            out_xor = (newRE ^ round_key[count1])
            s_box = substitute(out_xor)
            permute = s_box.permute(P)
            left = LE ^ permute
            LE=RE
            RE=left

        temp=LE
        LE=RE
        RE=temp
```



```
        Sum=LE+RE
        Sum=BitVector.BitVector(bitstring = Sum)
        inv_Sum = Sum.permute(PI_1)
        ab += inv_Sum
        #print(inv_Sum)
        string1 += inv_Sum.get_bitvector_in_ascii()
        blocks +=1

    ab1 = BitVector.BitVector(bitstring="")
    r = random.randint(0,55)
    #print(key[r])
    #print("adsada")
    if(key[r] == 0):
        key[r]=1
    else:
        key[r]=0
    round_key = generate_round_keys(key)

    while (bv1.more_to_read):
        bitvec = bv1.read_bits_from_file( 64 )
        if len(bitvec) > 0:
            if len(bitvec)<64:
                bitvec=addpading(bitvec)
                bitvec=BitVector.BitVector(bitstring=bitvec)
        bitvec = bitvec.permute(PI)
#permute the initial permutation

        [LE, RE] = bitvec.divide_into_two()
        for count1 in range(0,16):
#Running the cycle for 16 Rounds
            newRE = RE.permute(expansion_permutation)
#Permuting the array
            out_xor = (newRE ^ round_key[count1])
#Xoring for each round
            s_box = substitute(out_xor)
#FBox Function
            permute = s_box.permute(P)
```

```
                 left = LE ^ permute
                 LE=RE
                 RE=left

           temp=LE
           LE=RE                               #Inversing final time
           RE=temp

           Sum=LE+RE
           Sum=BitVector.BitVector(bitstring = Sum)
           inv_Sum2 = Sum.permute(PI_1)
           ab1 += inv_Sum2
           string2 += inv_Sum.get_bitvector_in_ascii()
#
      coun=0
      for i in range(0,len(ab)):              #Loop to count how many
bits are changed in ciphertext
           if(ab[i]!=ab1[i]):
                      coun=coun+1
      print("The Count of bits which is different is")
      print "Bits changed = ",coun
      print "Average is ",coun/blocks

#Function To generate S_Box
def generateBox():
  global s_boxes
  s_boxes = {i:None for i in range(8)}
  for index in range(8):
    templ = []
    for jindex in range(4):
      var = list(range(0,16))
      var = random.sample(var,16)
      templ.append(var)

    s_boxes[index] = templ

#Appending the bits to make it divisble by 8
def addpading(text):
    text = str(text)
    global padlen
    global flag
    flag=1
    padlen = 64-len(text)
    text+=padlen*"1"
    return text

#Removing the Padding Bits which was appended while ecrypting the
message
def removepading(text):
    lengt = len(text) - (padlen/8)
    return text[0:int(lengt)]

print("After Changing bit in plaintext:")
diffusion()
print("After generating SBox and permute:")
generateBox()
print("Changing SBox and then calling Diffusion:")
diffusion()
print("Changing SBox and then calling Diffusion 2nd Time:")
generateBox()
print("Diffusion again")
diffusion()
print("After Confusion We get:")
confusion()
```