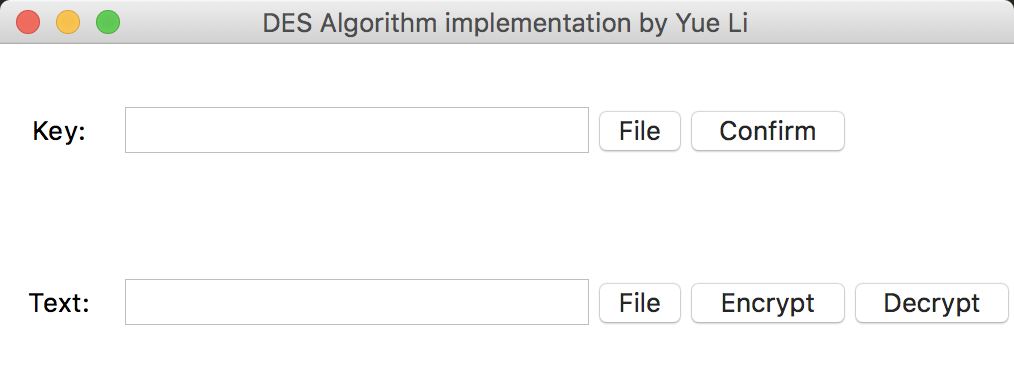
**Project 1 Lab Report**

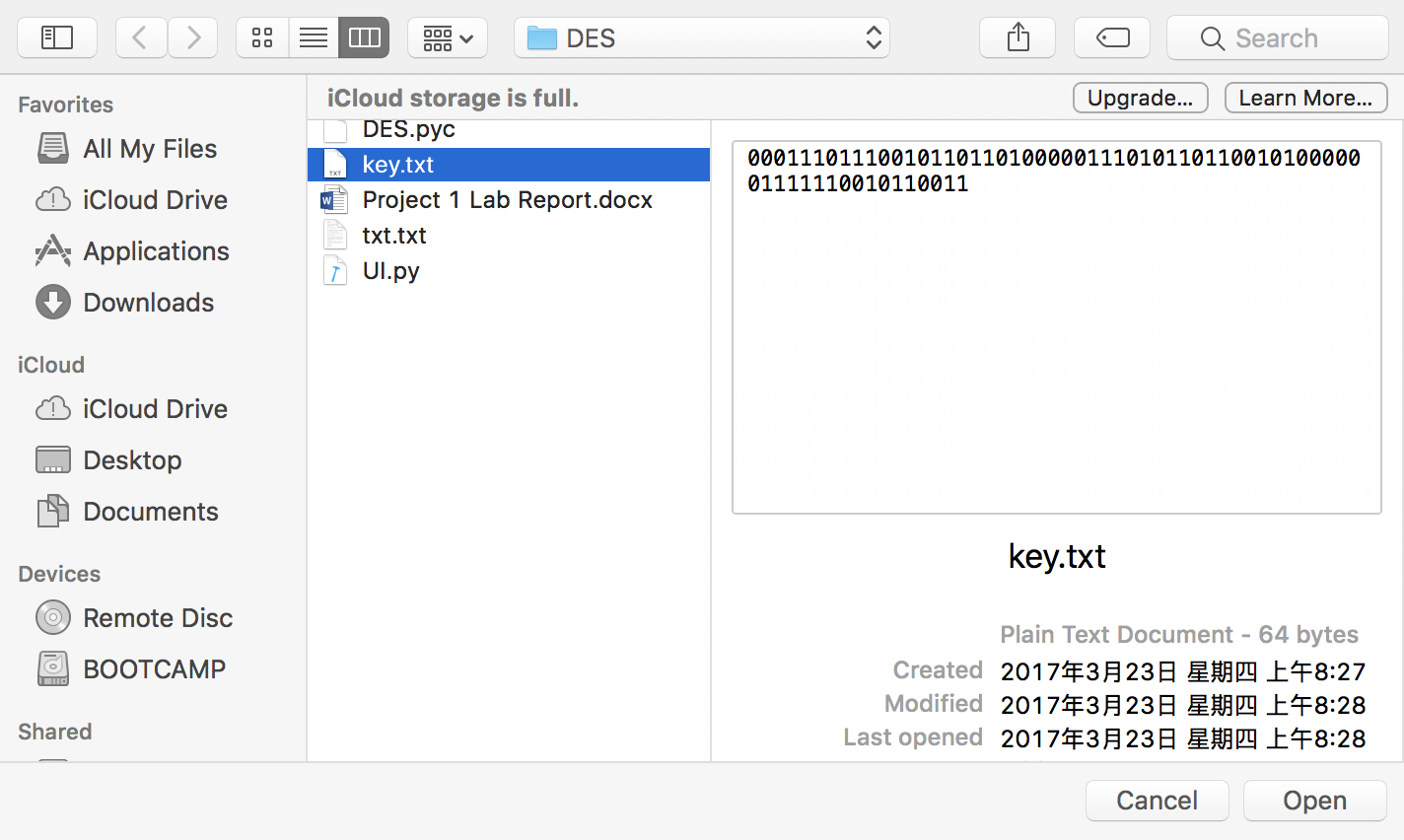
李悦 14080214

1. **Demonstration**

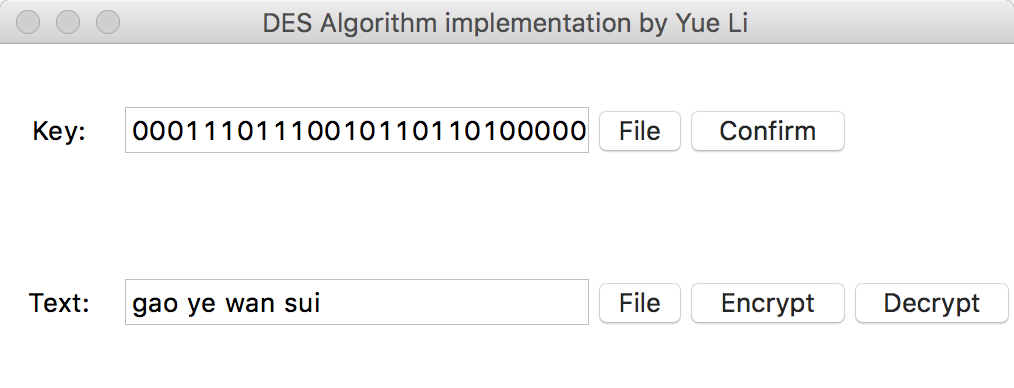
**Start state**

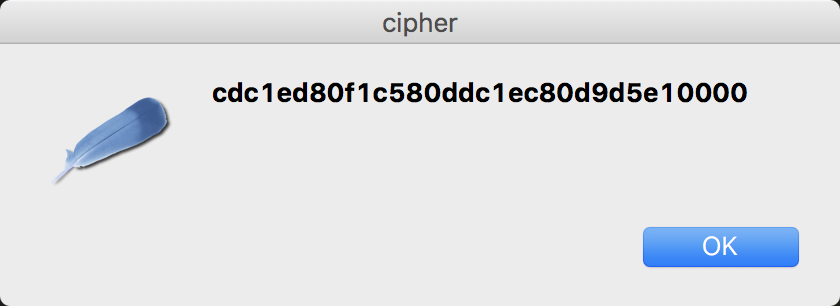
****

**Key inputting**

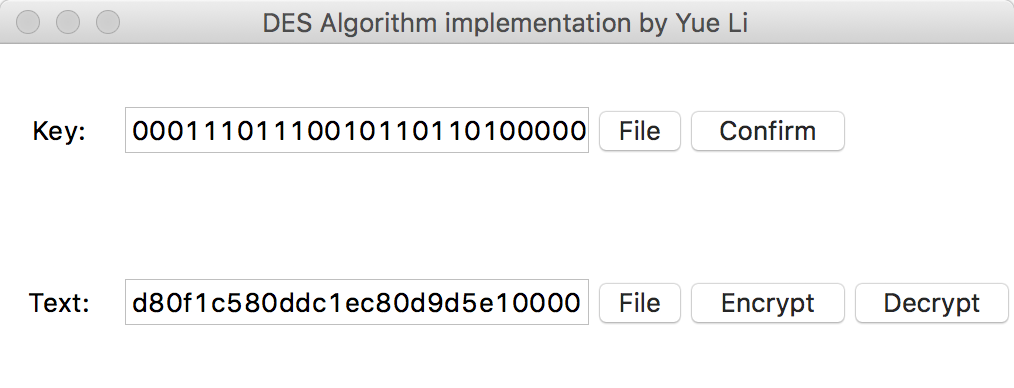
****

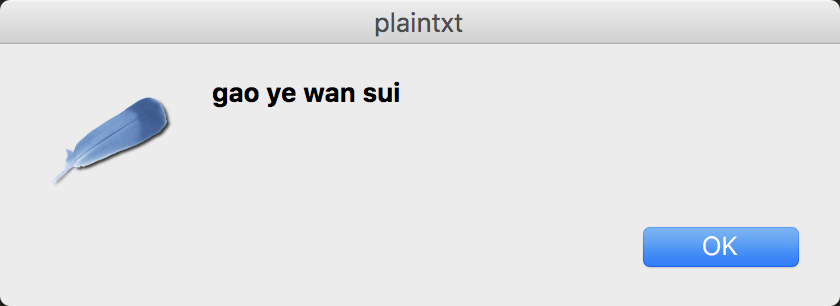
**Encryption**

****

****

**Decryption**

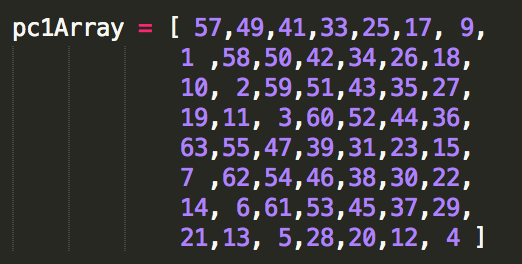
****

****

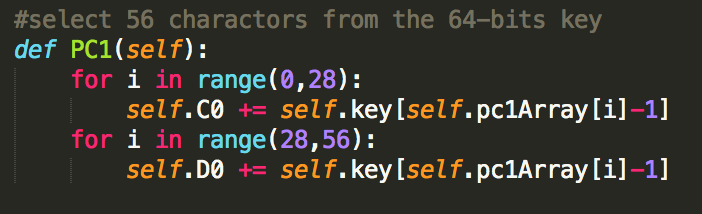
1. **DES Encryption algorithm and the implementation in Python**

**Step 1: Create 16 subkeys, each of which is 48-bits long.**

The 64-bit key is permuted according to the following table, PC-1.



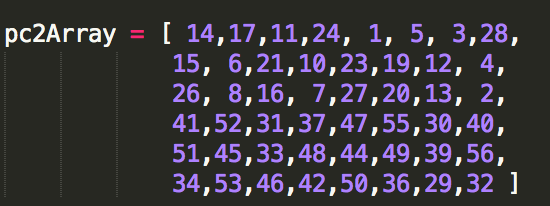
From the original 64-bit key, we get the 56-bit permutation. Next, split this key into left and right halves, C0 and D0, where each half has 28 bits.

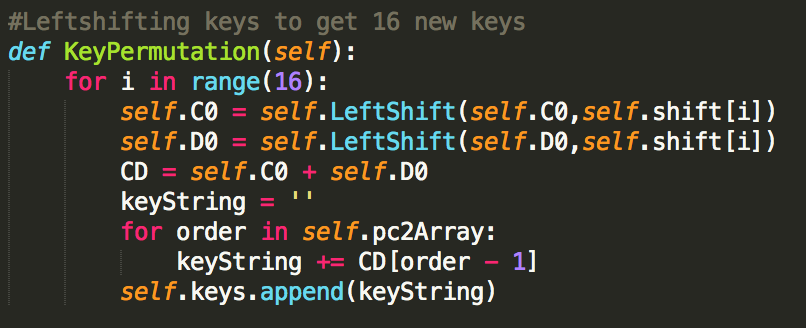


With C0 and D0 defined, we now create sixteen blocks Cn and Dn, 1<=n<=16. Each pair of blocks Cn and Dn is formed from the previous pair Cn-1 and Dn-1, respectively, for n = 1, 2, ..., 16, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.



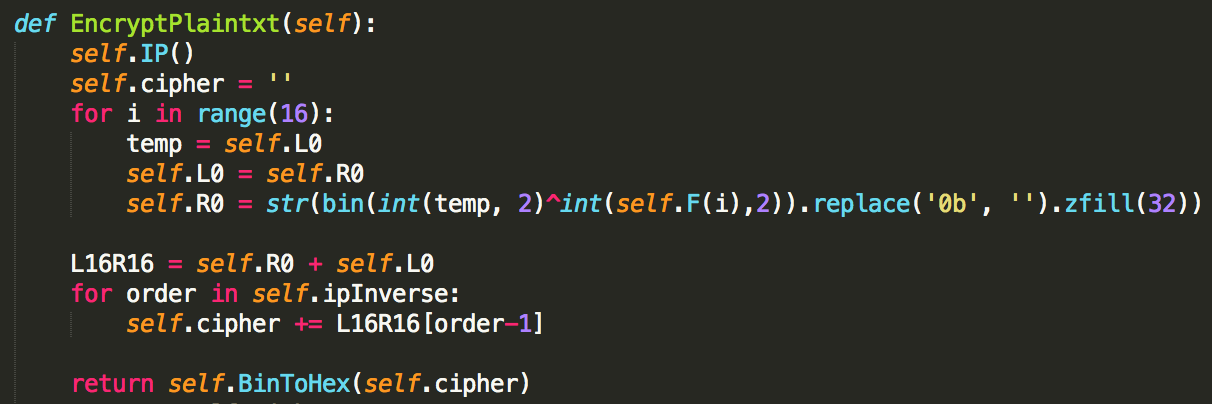
We now form the keys Kn, for 1<=n<=16, by applying the following permutation table to each of the concatenated pairs CnDn. Each pair has 56 bits, but PC-2 only uses 48 of these.



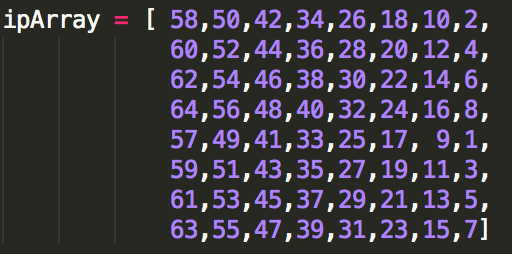


**Step 2: Encode each 64-bit block of data.**

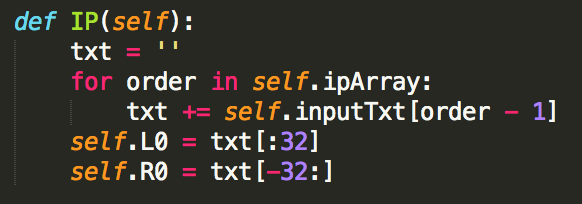
The whole implementation of encryption function is depicted as following:



There is an initial permutation IP of the 64 bits of the message data M. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order.



Next divide the permuted block IP into a left half L0 of 32 bits, and a right half R0 of 32 bits.



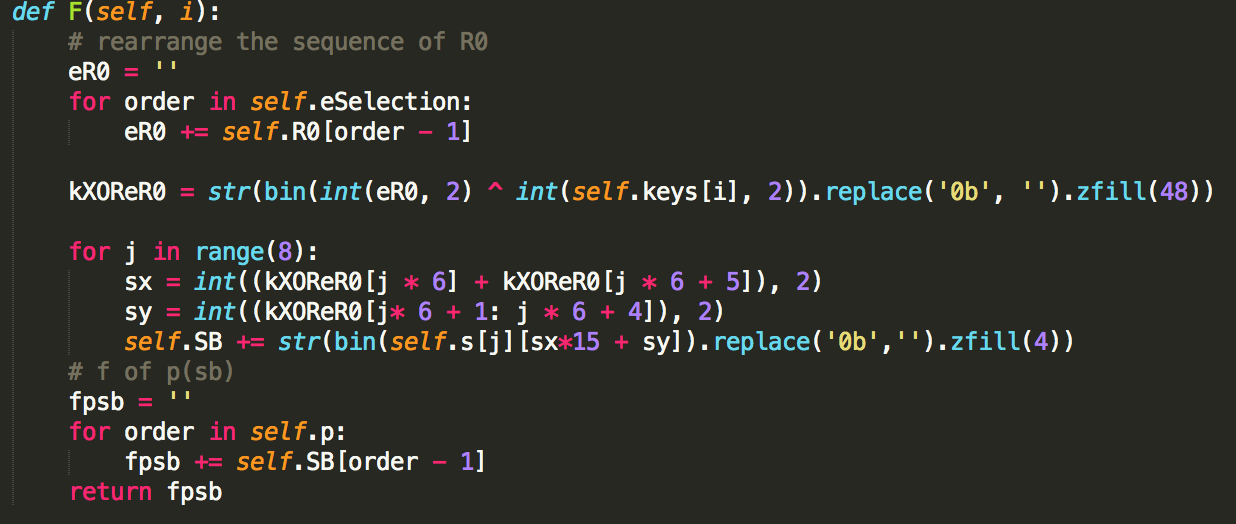
We now proceed through 16 iterations, for 1<=n<=16, using a function f which operates on two blocks--a data block of 32 bits and a key Kn of 48 bits--to produce a block of 32 bits. Let + denote XOR addition, (bit-by-bit addition modulo 2). Then for n going from 1 to 16 we calculate

Ln = Rn-1

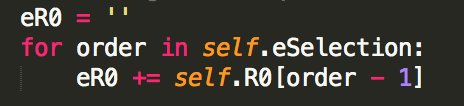
Rn = Ln-1 + f(Rn-1,Kn)

The whole implementation of function f is depicted in the following picture:

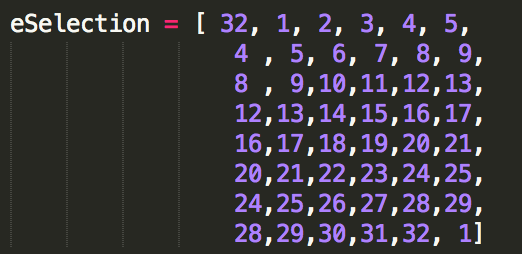
i is the iteration number.



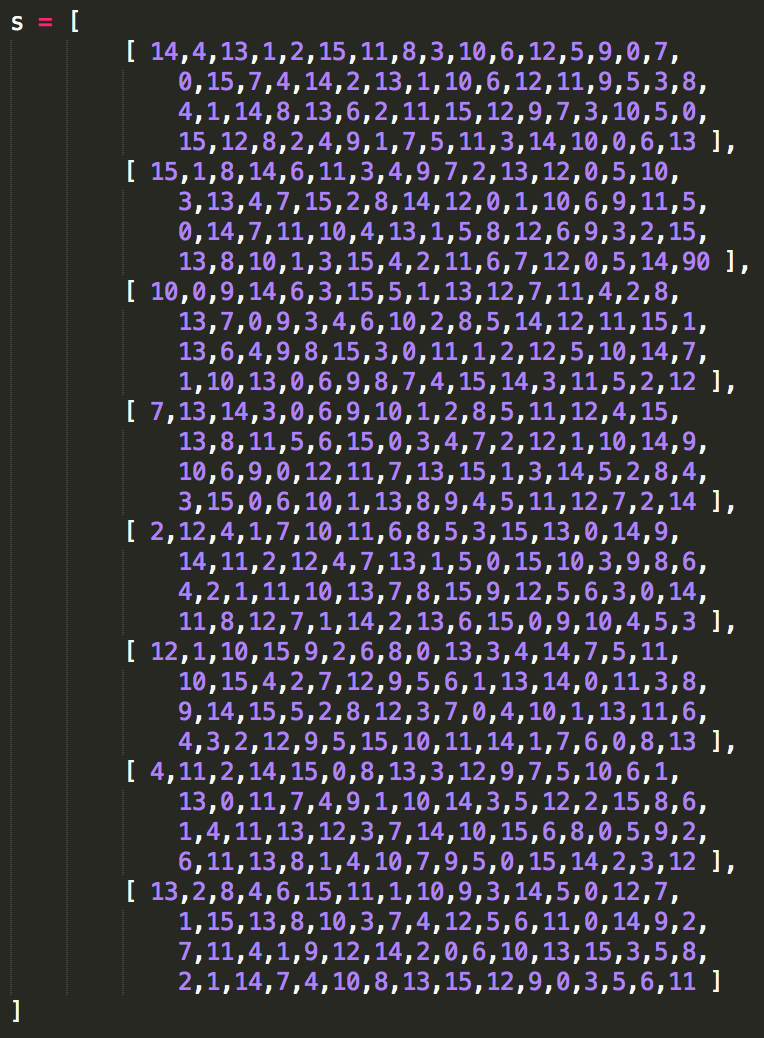
To calculate f, we first expand each block Rn-1 from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in Rn-1 . We'll call the use of this selection table the function E. Thus E(Rn-1) has a 32 bit input block, and a 48 bit output block.



Let E be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:



To this point we have expanded Rn-1 from 32 bits to 48 bits, using the selection table, and XORed the result with the key Kn . We now have 48 bits, or eight groups of six bits. Each group of six bits will give us an address in a different S box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the S boxes) for 32 bits total.



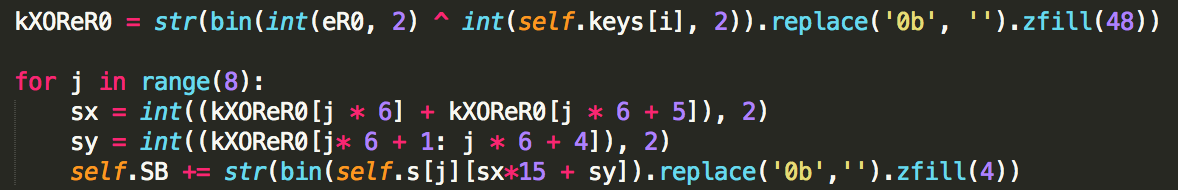
Write the previous result, which is 48 bits, in the form:

Kn + E(Rn-1) =B1B2B3B4B5B6B7B8,

where each Bi is a group of six bits. We now calculate

S1(B1)S2(B2)S3(B3)S4(B4)S5(B5)S6(B6)S7(B7)S8(B8)

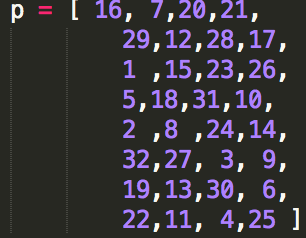
where Si(Bi) referres to the output of the i-th S box.



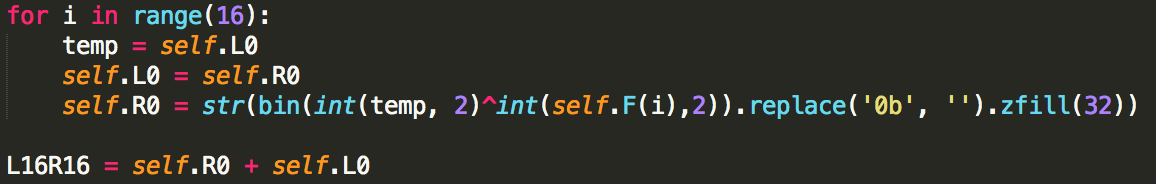
The final stage in the calculation of f is to do a permutation P of the S-box output to obtain the final value of f:

f = P(S1(B1)S2(B2)...S8(B8))

The permutation P is defined in the following table. P yields a 32-bit output from a 32-bit input by permuting the bits of the input block.



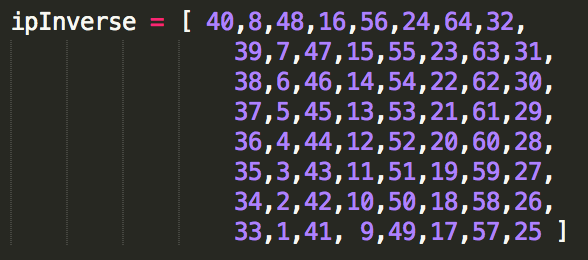
This results in a final block, for n = 16, of L16R16. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation f.



At the end of the sixteenth round we have the blocks L16 and R16. We then reverse the order of the two blocks into the 64-bit block

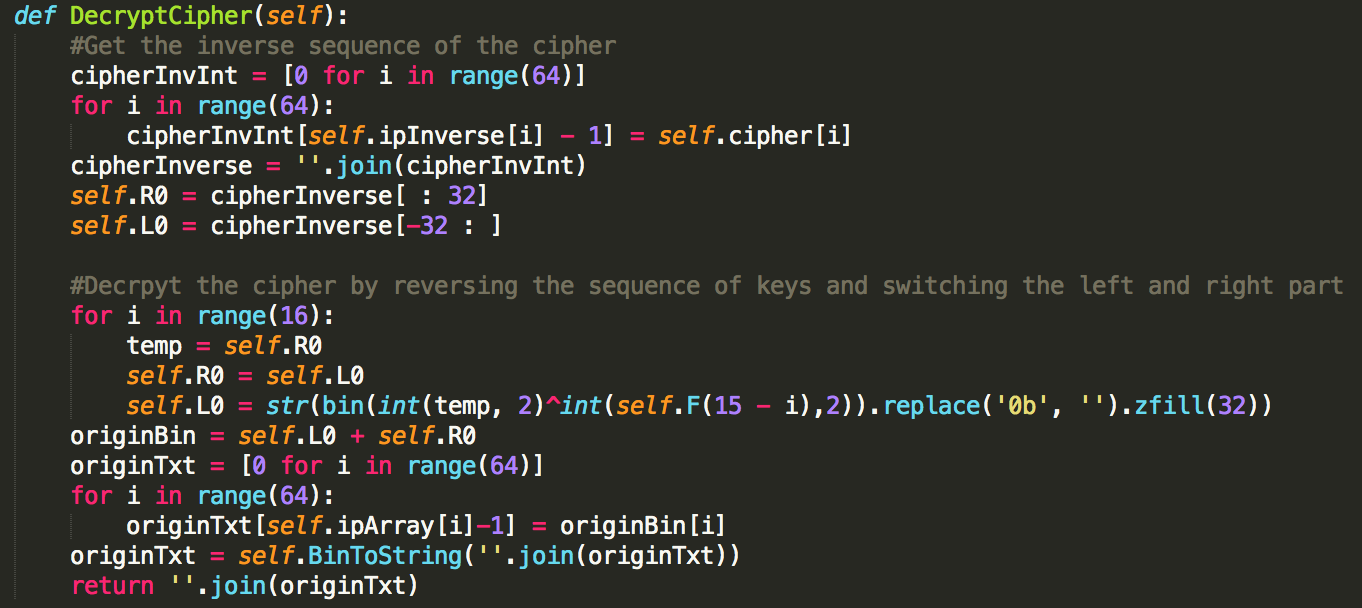
R16L16

and apply a final permutation IP-1 as defined by the following table:

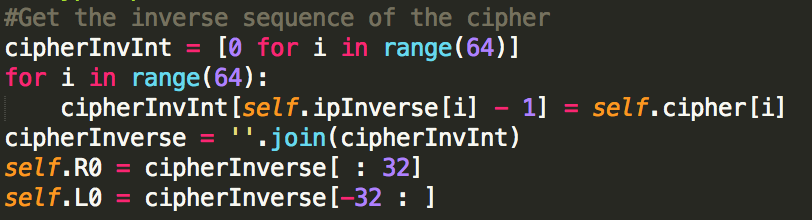


1. **DES Decryption algorithm and the implementation in Python**

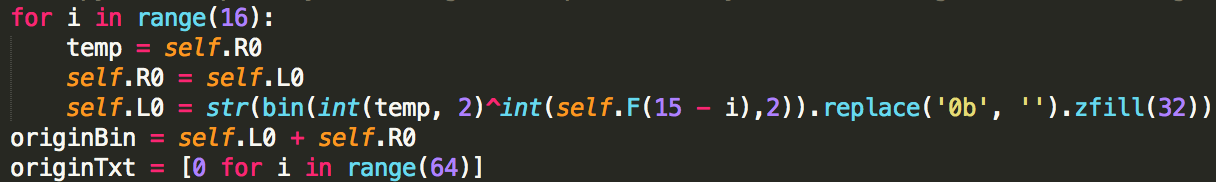
The whole implementation of encryption function is depicted as following:



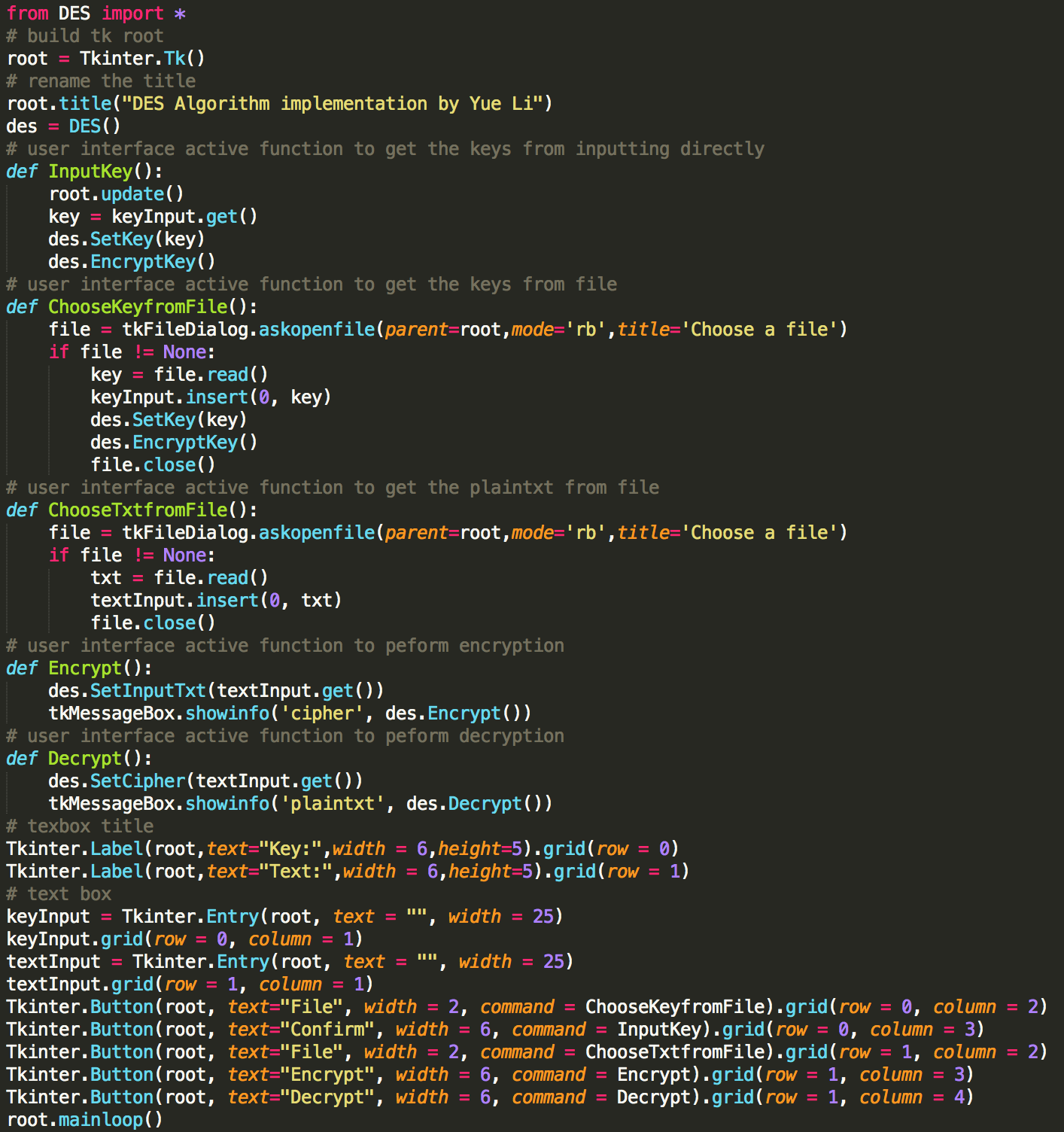
First we reverse the cipher using the IP inverse array and get R0 as the left part of the cipher and L0 as the right part of the cipher.



Then we get the initial R0 and L0 by using the same F function but employing the opposite sequence of the 16 subkeys.

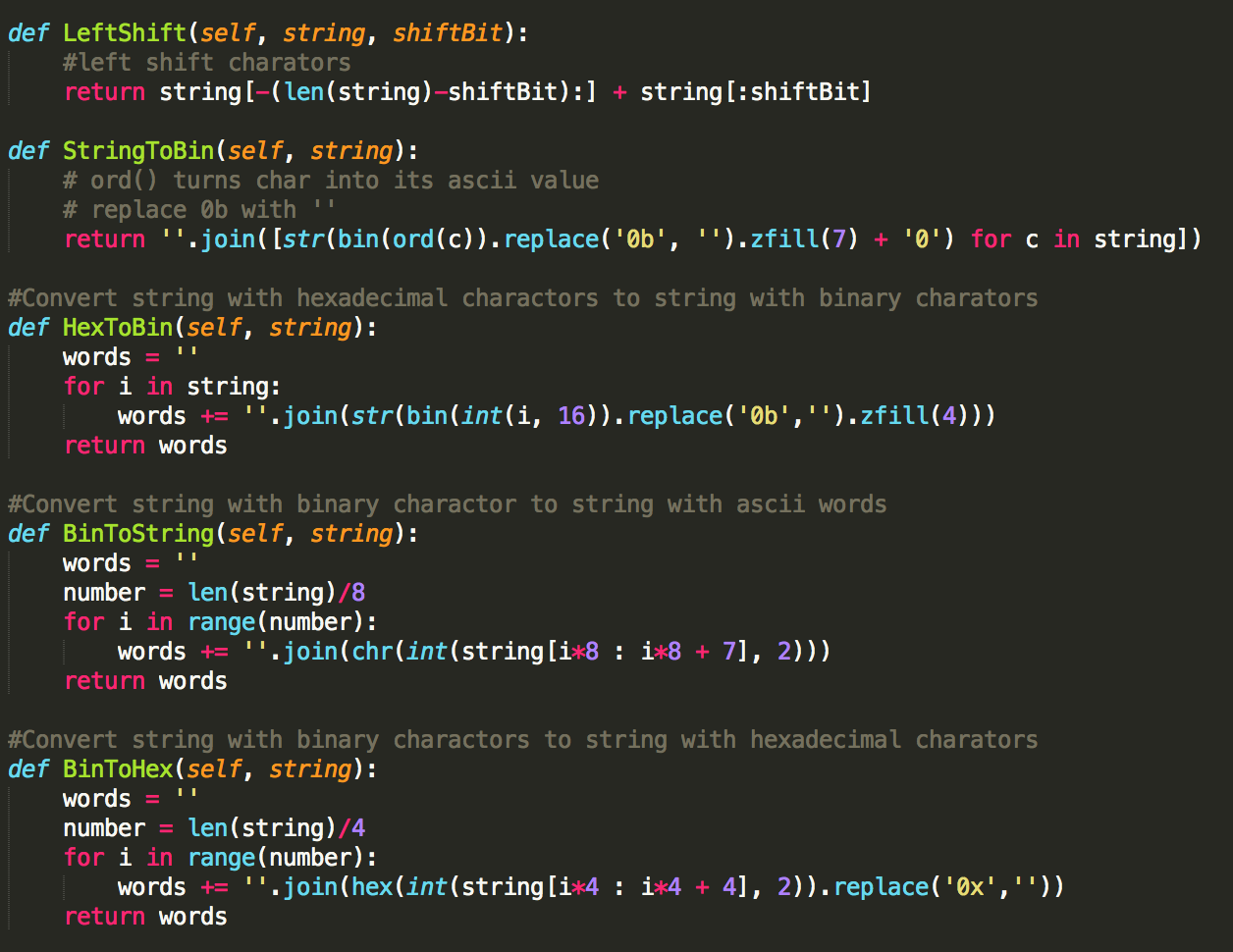


1. **User Interface and the implementation in Python**

****

1. **Other function and the implementation in Python**

This include left shifting strings and converting between string in ASCII characters to string in binary characters and between string in binary characters to hexadecimal characters.

****

1. **Other function and the implementation in Python**

This involve the handling for plaintexts and cipher that are not exact 64-bits long



1. **Complete source codes in Python**

Des.py

class DES(object):

"""docstring for DES"""

C0=''

D0=''

L0=''

R0=''

key = []

keys = []

inputTxt = ''

cipher = ''

SB = ''

remainder = 0

blockNum = 0

pc1Array = [ 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 ]

pc2Array = [ 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 ]

shift = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

ipArray = [ 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]

eSelection = [ 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]

s = [

[ 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,

13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,90 ],

[ 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 ]

]

p = [ 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 ]

ipInverse = [ 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 \_\_init\_\_(self):

pass

def EncryptKey(self):

self.PC1()

self.KeyPermutation()

def EncryptPlaintxt(self):

self.IP()

self.cipher = ''

for i in range(16):

temp = self.L0

self.L0 = self.R0

self.R0 = str(bin(int(temp, 2)^int(self.F(i),2)).replace('0b', '').zfill(32))

L16R16 = self.R0 + self.L0

for order in self.ipInverse:

self.cipher += L16R16[order-1]

return self.BinToHex(self.cipher)

#return self.cipher

def DecryptCipher(self):

#Get the inverse sequence of the cipher

cipherInvInt = [0 for i in range(64)]

for i in range(64):

cipherInvInt[self.ipInverse[i] - 1] = self.cipher[i]

cipherInverse = ''.join(cipherInvInt)

self.R0 = cipherInverse[ : 32]

self.L0 = cipherInverse[-32 : ]

#Decrpyt the cipher by reversing the sequence of keys and switching the left and right part

for i in range(16):

temp = self.R0

self.R0 = self.L0

self.L0 = str(bin(int(temp, 2)^int(self.F(15 - i),2)).replace('0b', '').zfill(32))

originBin = self.L0 + self.R0

originTxt = [0 for i in range(64)]

for i in range(64):

originTxt[self.ipArray[i]-1] = originBin[i]

originTxt = self.BinToString(''.join(originTxt))

return ''.join(originTxt)

def F(self, i):

# rearrange the sequence of R0

eR0 = ''

for order in self.eSelection:

eR0 += self.R0[order - 1]

kXOReR0 = str(bin(int(eR0, 2) ^ int(self.keys[i], 2)).replace('0b', '').zfill(48))

for j in range(8):

sx = int((kXOReR0[j \* 6] + kXOReR0[j \* 6 + 5]), 2)

sy = int((kXOReR0[j\* 6 + 1: j \* 6 + 4]), 2)

self.SB += str(bin(self.s[j][sx\*15 + sy]).replace('0b','').zfill(4))

# f of p(sb)

fpsb = ''

for order in self.p:

fpsb += self.SB[order - 1]

return fpsb

def IP(self):

txt = ''

for order in self.ipArray:

txt += self.inputTxt[order - 1]

self.L0 = txt[:32]

self.R0 = txt[-32:]

#select 56 charactors from the 64-bits key

def PC1(self):

for i in range(0,28):

self.C0 += self.key[self.pc1Array[i]-1]

for i in range(28,56):

self.D0 += self.key[self.pc1Array[i]-1]

#Leftshifting keys to get 16 new keys

def KeyPermutation(self):

for i in range(16):

self.C0 = self.LeftShift(self.C0,self.shift[i])

self.D0 = self.LeftShift(self.D0,self.shift[i])

CD = self.C0 + self.D0

keyString = ''

for order in self.pc2Array:

keyString += CD[order - 1]

self.keys.append(keyString)

def LeftShift(self, string, shiftBit):

#left shift charators

return string[-(len(string)-shiftBit):] + string[:shiftBit]

def StringToBin(self, string):

# ord() turns char into its ascii value

# replace 0b with ''

return ''.join([str(bin(ord(c)).replace('0b', '').zfill(7) + '0') for c in string])

#Convert string with hexadecimal charactors to string with binary charators

def HexToBin(self, string):

words = ''

for i in string:

words += ''.join(str(bin(int(i, 16)).replace('0b','').zfill(4)))

return words

#Convert string with binary charactor to string with ascii words

def BinToString(self, string):

words = ''

number = len(string)/8

for i in range(number):

words += ''.join(chr(int(string[i\*8 : i\*8 + 7], 2)))

return words

#Convert string with binary charactors to string with hexadecimal charators

def BinToHex(self, string):

words = ''

number = len(string)/4

for i in range(number):

words += ''.join(hex(int(string[i\*4 : i\*4 + 4], 2)).replace('0x',''))

return words

def Encrypt(self):

fullCipher = ''

self.inputTxt = self.StringToBin(self.inputTxt)

#Ensuring input text could be divide into int number of blocks

self.remainder = len(self.inputTxt)%64

if(self.remainder != 0):

for i in range(64 - self.remainder):

self.inputTxt += '0'

self.blockNum = len(self.inputTxt)/64

inputTxtBackup = self.inputTxt

#Encrypting each blocks

for i in range(self.blockNum):

self.inputTxt = inputTxtBackup[i\*64 : i\*64 + 64]

fullCipher += self.EncryptPlaintxt()

return fullCipher

def Decrypt(self):

fullPlaintxt = ''

self.cipher = self.HexToBin(self.cipher)

self.remainder = len(self.cipher)%64

if(self.remainder != 0):

for i in range(64 - self.remainder):

self.cipher += '0'

cipherBackup = self.cipher

self.blockNum = len(self.cipher)/64

#Decrypting each blocks

for i in range(self.blockNum):

self.cipher = cipherBackup[i\*64 : i\*64 + 64]

fullPlaintxt += self.DecryptCipher()

#Ignoring the appended bits

return fullPlaintxt

def SetKey(self, key):

self.key = key

def SetInputTxt(self, inputTxt):

self.inputTxt = inputTxt

def SetCipher(self, cipher):

self.cipher = cipher

UI.py

import Tkinter

import tkMessageBox

import tkFileDialog

from DES import \*

# build tk root

root = Tkinter.Tk()

# rename the title

root.title("DES Algorithm implementation by Yue Li")

des = DES()

# user interface active function to get the keys from inputting directly

def InputKey():

root.update()

key = keyInput.get()

des.SetKey(key)

des.EncryptKey()

# user interface active function to get the keys from file

def ChooseKeyfromFile():

file = tkFileDialog.askopenfile(parent=root,mode='rb',title='Choose a file')

if file != None:

key = file.read()

keyInput.insert(0, key)

des.SetKey(key)

des.EncryptKey()

file.close()

# user interface active function to get the plaintxt from file

def ChooseTxtfromFile():

file = tkFileDialog.askopenfile(parent=root,mode='rb',title='Choose a file')

if file != None:

txt = file.read()

textInput.insert(0, txt)

file.close()

# user interface active function to peform encryption

def Encrypt():

des.SetInputTxt(textInput.get())

tkMessageBox.showinfo('cipher', des.Encrypt())

# user interface active function to peform decryption

def Decrypt():

des.SetCipher(textInput.get())

tkMessageBox.showinfo('plaintxt', des.Decrypt())

# texbox title

Tkinter.Label(root,text="Key:",width = 6,height=5).grid(row = 0)

Tkinter.Label(root,text="Text:",width = 6,height=5).grid(row = 1)

# text box

keyInput = Tkinter.Entry(root, text = "", width = 25)

keyInput.grid(row = 0, column = 1)

textInput = Tkinter.Entry(root, text = "", width = 25)

textInput.grid(row = 1, column = 1)

Tkinter.Button(root, text="File", width = 2, command = ChooseKeyfromFile).grid(row = 0, column = 2)

Tkinter.Button(root, text="Confirm", width = 6, command = InputKey).grid(row = 0, column = 3)

Tkinter.Button(root, text="File", width = 2, command = ChooseTxtfromFile).grid(row = 1, column = 2)

Tkinter.Button(root, text="Encrypt", width = 6, command = Encrypt).grid(row = 1, column = 3)

Tkinter.Button(root, text="Decrypt", width = 6, command = Decrypt).grid(row = 1, column = 4)

root.mainloop()