Contents

Synopsis:

This is the report for ISEC2000 Fundamental Concepts of Cryptography Assignment 1. It will go through all aspects mentioned and required in the assignment brief, which are:

- Going through code for the letter frequency analysis and brute forcing affine decryption step-by-step.
- The resulting substitution table got from the attacks.
- The key found by the brute forcing using the affine attack.

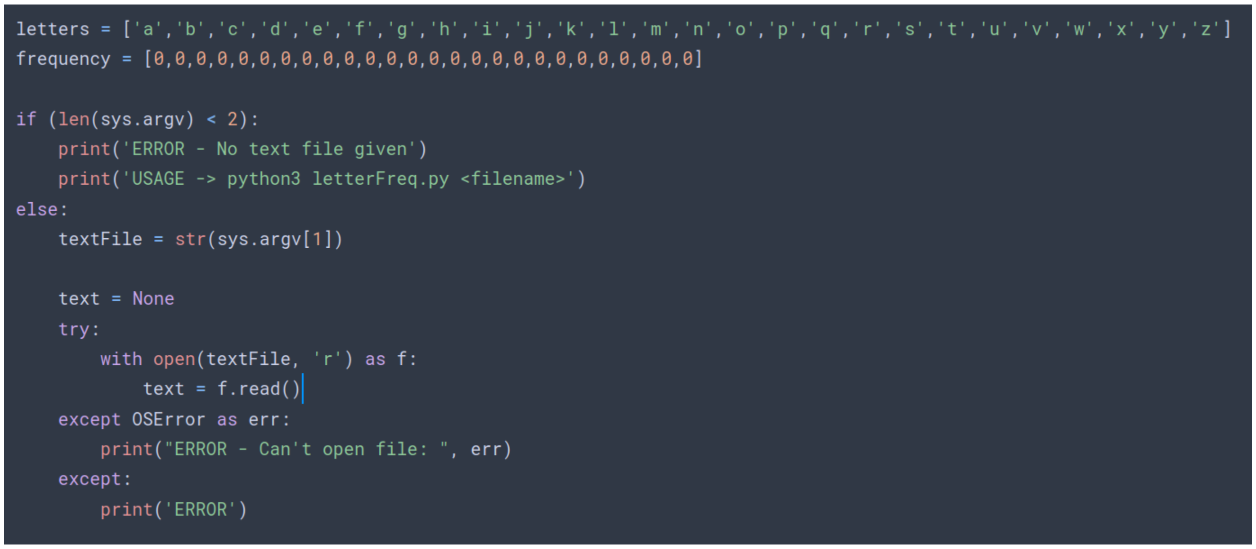
Question 1:

(a) Code Walkthrough:

letterFreq.py:

This is the code for the letter frequency analysis.

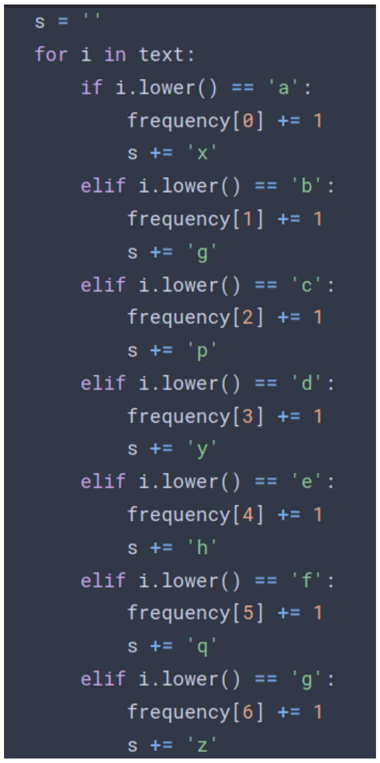
1



```python
letters = ['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']
frequency = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

if (len(sys.argv) < 2):
    print('ERROR - No text file given')
    print('USAGE -> python3 letterFreq.py <filename>')
else:
    textFile = str(sys.argv[1])

    text = None
    try:
        with open(textFile, 'r') as f:
            text = f.read()
    except OSError as err:
        print("ERROR - Can't open file: ", err)
    except:
        print('ERROR')
```

- Figure 1: "letters" list created that stores all 26 English alphabets, and corresponding "frequency" list created, storing 26 integers, initialized as 0.
- Figure 1: First if function is just to check whether a file is given as a command line argument to the program or not. Usage is shown if incorrectly executed.
- Figure 1: Once executed correctly, "textFile" variable stores filename, and open() function uses this to open and read the contents of the file into the "text" variable. Exceptions set in case of errors.
- Figure 2: For loop, looping through every character in the "text" variable, with a long string of if-else, statements that compare each character in "text" to an alphabet and increments their corresponding position in the "frequency" list. For example, if character is "a" increments integer at index 0 of "frequency" list.

2

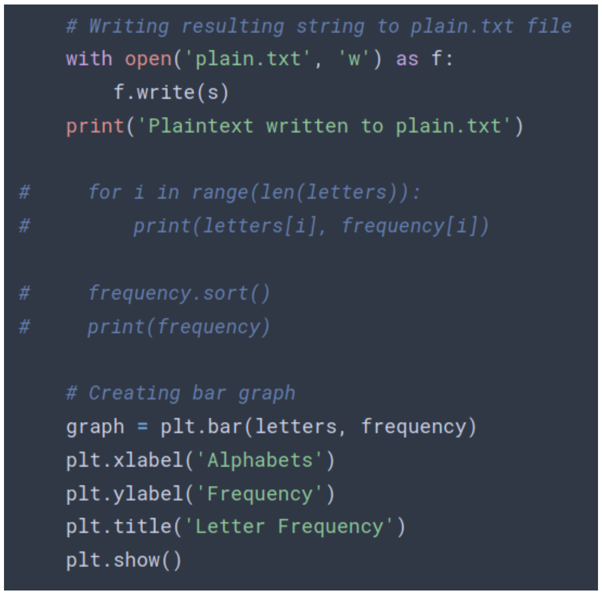


```python
s = ''
for i in text:
    if i.lower() == 'a':
        frequency[0] += 1
        s += 'x'
    elif i.lower() == 'b':
        frequency[1] += 1
        s += 'g'
    elif i.lower() == 'c':
        frequency[2] += 1
        s += 'p'
    elif i.lower() == 'd':
        frequency[3] += 1
        s += 'y'
    elif i.lower() == 'e':
        frequency[4] += 1
        s += 'h'
    elif i.lower() == 'f':
        frequency[5] += 1
        s += 'q'
    elif i.lower() == 'g':
        frequency[6] += 1
        s += 'z'
```

Full statement omitted

- Figure 2: “s” variable here is a string, that gets concatenated with a different mapped alphabet for every character in “text”, doing the substitution from the letter frequency analysis. This part requires a lot of human intervention, as basic lingual logic is required, so I had to manually find most of the values through executing the code multiple time with different values, that make sense from the frequency analysis, until I got it right.

3



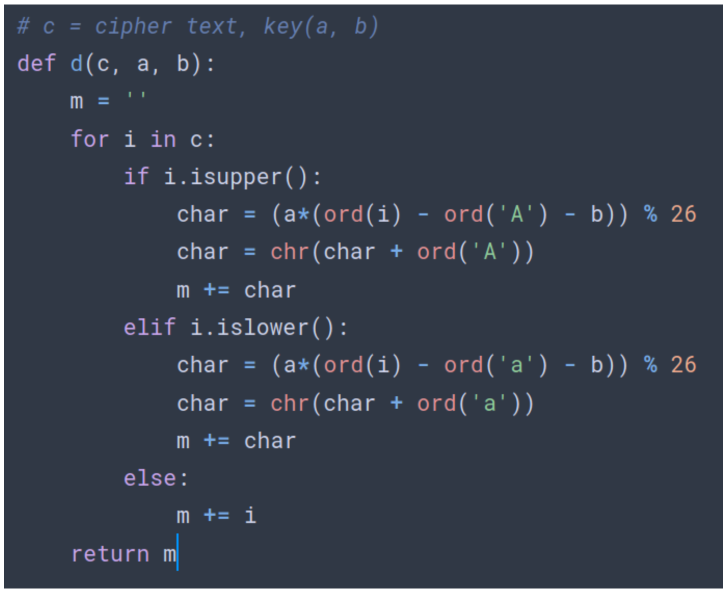
```
    # Writing resulting string to plain.txt file
    with open('plain.txt', 'w') as f:
        f.write(s)
    print('Plaintext written to plain.txt')

#     for i in range(len(letters)):
#         print(letters[i], frequency[i])

#     frequency.sort()
#     print(frequency)

    # Creating bar graph
    graph = plt.bar(letters, frequency)
    plt.xlabel('Alphabets')
    plt.ylabel('Frequency')
    plt.title('Letter Frequency')
    plt.show()
```

- Figure 3: Using the open() function to open file called “plain.txt”, it writes the resulting “s” variable, after the for loop, to the file. Creates the file if it doesn’t exist, and overwrites it, if it does exist.
- Figure 3: Commented part here is for lab demonstration.
- Figure 3: Creating “graph” variable to store bar graph object, created using matplotlib library. Uses “letters” list as x axis values, and “frequency” as y axis. Displays the bar graph at the end of execution.

affine.py:

This file stores the decryption function and brute forcing for the affine cipher part of the assignment.

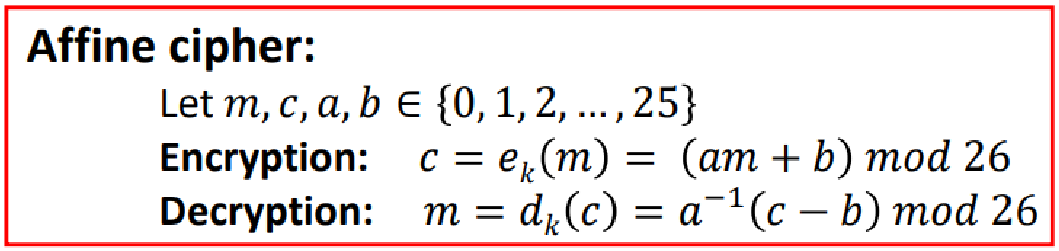
4



```
# c = cipher text, key(a, b)
def d(c, a, b):
    m = ''
    for i in c:
        if i.isupper():
            char = (a*(ord(i) - ord('A') - b)) % 26
            char = chr(char + ord('A'))
            m += char
        elif i.islower():
            char = (a*(ord(i) - ord('a') - b)) % 26
            char = chr(char + ord('a'))
            m += char
        else:
            m += i
    return m
```

- Figure 4: Function called “d()” created (stands for decryption), which takes in c = cipher text, and key(a, b). “m” variable created to store decrypted message. Uses for loop to loop through each character in cipher text. If character is not an upper or lowercase English character, simply concatenate character to “m”. Else if character is upper case, get ascii code using ord() function and – 65 (as uppercase “A” is 65 in ascii), and used the decryption function (Figure 5) to get value for substituted character, +65, and using chr() function to convert integer back to alphabet, and concatenating to “m”. Do the same for lowercase character but change subtraction and addition value to 97 (as lowercase “a” is 97 in ascii). Return resulting “m”.

5



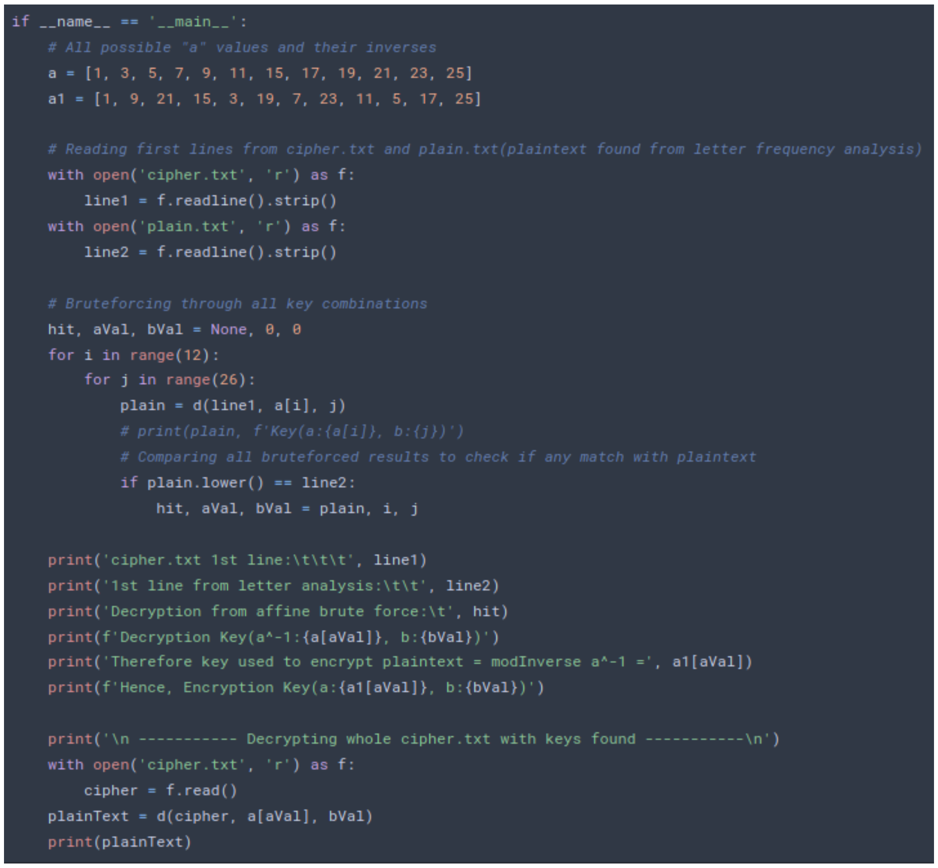
Affine cipher:

Let $m, c, a, b \in \{0, 1, 2, \dots, 25\}$

Encryption: $c = e_k(m) = (am + b)\ mod\ 26$

Decryption: $m = d_k(c) = a^{-1}(c - b)\ mod\ 26$

6

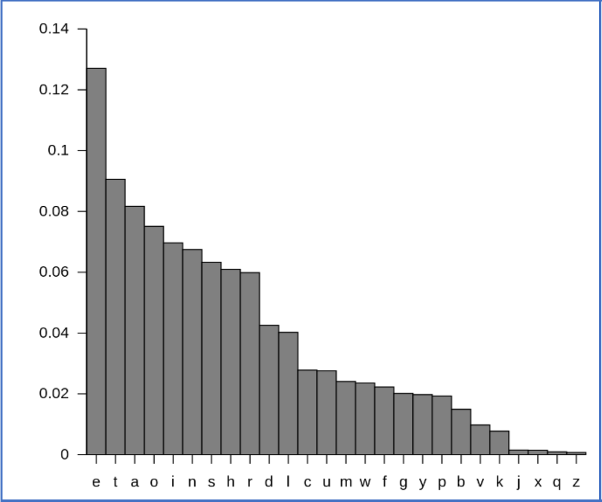
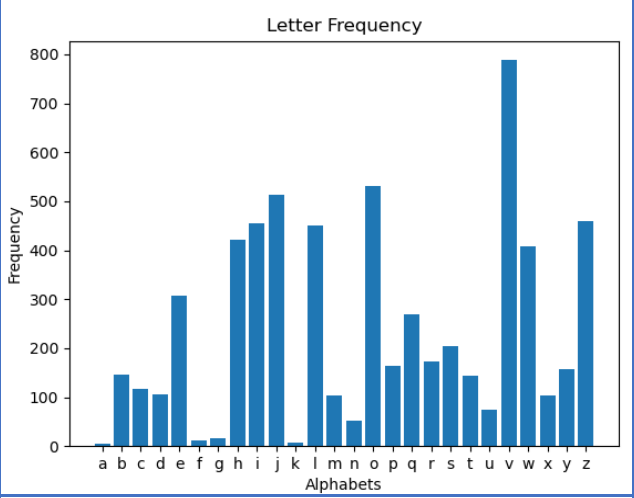


```python
if __name__ == '__main__':
    # All possible "a" values and their inverses
    a = [1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25]
    a1 = [1, 9, 21, 15, 3, 19, 7, 23, 11, 5, 17, 25]

    # Reading first lines from cipher.txt and plain.txt(plaintext found from letter frequency analysis)
    with open('cipher.txt', 'r') as f:
        line1 = f.readline().strip()
    with open('plain.txt', 'r') as f:
        line2 = f.readline().strip()

    # Bruteforcing through all key combinations
    hit, aVal, bVal = None, 0, 0
    for i in range(12):
        for j in range(26):
            plain = d(line1, a[i], j)
            # print(plain, f'Key(a:{a[i]}, b:{j})')
            # Comparing all bruteforced results to check if any match with plaintext
            if plain.lower() == line2:
                hit, aVal, bVal = plain, i, j

    print('cipher.txt 1st line:\t\t\t', line1)
    print('1st line from letter analysis:\t\t', line2)
    print('Decryption from affine brute force:\t', hit)
    print(f'Decryption Key(a^-1:{a[aVal]}, b:{bVal})')
    print('Therefore key used to encrypt plaintext = modInverse a^-1 =', a1[aVal])
    print(f'Hence, Encryption Key(a:{a1[aVal]}, b:{bVal})')

    print('\n ---------- Decrypting whole cipher.txt with keys found ----------\n')
    with open('cipher.txt', 'r') as f:
        cipher = f.read()
    plainText = d(cipher, a[aVal], bVal)
    print(plainText)
```

- Figure 6: “a” list created to store all possible values of a can have, since only these 12 values exist that satisfy modular inverses, and inverses for each index stored in list “a1”.
- Figure 6: Reading the only the first lines of cipher.txt file and plain.txt files, containing the cipher text and result got from letter frequency analysis attack from the previous part, respectively.
- Figure 6: Since there can only 12x26 possible key combinations for affine cipher, using two for loops with these values, and using the “d()” function created to brute force all possibilities, using the first line from cipher.txt as “c” variable (only one line used for quicker brute forcing).
- Figure 6: Comparing the resulting possibilities to first line from plain.txt, to check if the result is intelligible English. If it is, store the resulting in “hit” variable, storing current index for a value in “aVal” and current b value in “bVal” variables respectively.
- Figure 6: Printing out the single line results, with the key. Then reading the whole cipher from cipher.txt and using it as the “c” variable in the “d()” function, with the keys stored in index “aVal” and “bVal” variables and printing the result.

(b) Substitution Table:

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

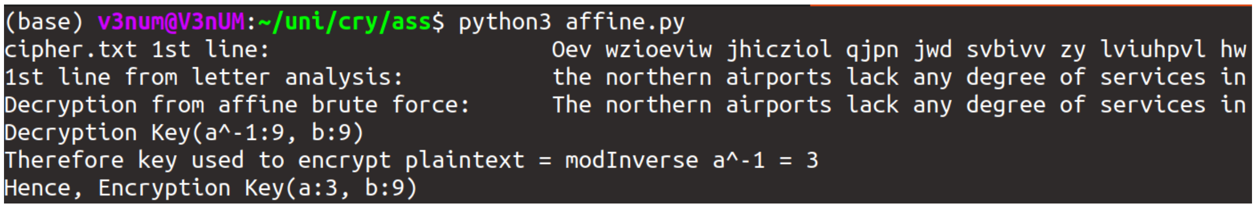
	
Normal Letter frequency	cipher.txt Letter frequency

(c) Key From Brute Force:

Decryption key got from brute forcing the cipher.txt file was Key(a^{-1}=9, b=9), therefore Encryption Key(a=3, b=9).

Figure 7: Shows result from executing affine.py, showing the key used to get the correct decryption, after comparison with 1st line from plain.txt.

7



```
(base) v3num@V3nUM:~/uni/cry/ass$ python3 affine.py
cipher.txt 1st line:                    Oev wzioeviw jhicziol qjpn jwd svbivv zy lviuhpvl hw
1st line from letter analysis:          the northern airports lack any degree of services in
Decryption from affine brute force:     The northern airports lack any degree of services in
Decryption Key(a^-1:9, b:9)
Therefore key used to encrypt plaintext = modInverse a^-1 = 3
Hence, Encryption Key(a:3, b:9)
```

Full Result Omitted

Question 2:

(a) Success, Lessons Learnt, and Difficulties:

Successful Plaintext Recovery?

The plaintext was successfully recovered after the encryption using the specific DES I made, proving that it works theoretically and practically if decryption is done completely, and exactly, inverse to the encryption.

Lessons Learnt:

I learnt through theory that the Data Encryption Standard process was very systematical. When I completed its implementation, this became very clear. As I got closer to encrypting a block, the realization set in that understanding is the key to success with this, and since then, the project became a very logical and step-by-step journey, where each section is consistently dependent on another. An example is that the initial permutation and final permutation tables are inverses and are REQUIRED to be used to specifics. Other lessons include:

- The mathematics that makes substitution boxes effective.
- Efficiency of covering blocks of bits instead of single bits.
- Coherent functioning of every step performed in a round, and how decryption perfectly fits the pieces to decipher text.

Difficulties:

This was my first time working with a somewhat complex encryption algorithm, taking into account the key scheduling, multiple rounds, different forms of permutations, and proof of decryption. Even after all these, due to the systematic and logical nature of DES, I didn't find too much problem understanding how to program it, though it's not perfect. Here are some problems that I did face:

- Working with bits: The first problem was figuring out how to work with bits in python. Though this was an easy problem in hindsight, it was my first time doing so. The solution was to use the bin() function and treat the binaries as python strings. This was perfect, since strings in python are very easy to work with, in terms of concatenation and indexing, making things like the permutation simpler.
- Keeping track of components: This was probably the biggest issue with regards to the implementation. There are an abundant number of things that are required to be EXACT for the DES to work and missing even a small part would lead to incorrect functions, leading to problems down the line. An example is that I missed the part where you are supposed to switch the right and left side of the block before the final permutation. This led to incorrect encryption and become an issue when it came to decryption.

- Decryption: Even after following the steps defined for the encryption in the theory of DES, the decryption took some time to get correct. As stated in the previous point, the numerous cogs the DES had made it difficult to keep up, which ended with uncertainty, with me wondering if my encryption was correct in the first place. After a lot of trial and error and going back and forth between working on different components, I got it working as intended.

(b) Key Initialized to 0 bits:

Theory:

When a key is initialized to 0 bits, it means all subkeys for rounds will be the same, them being blocks of 48-bit 0s. This leads to a fundamental DES problem, when it comes to decryption, making it an extremely weak key.

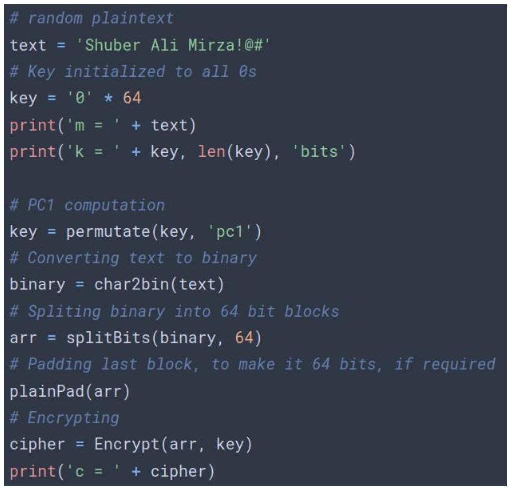
The theory behind decryption in the DES is that through going in the exact opposite manner to the encryption rounds, the ciphertext is decoded. This requires programming the algorithm to have key scheduling such that the subkey of round$_{e16}$ would be equal to subkey of round$_{d1}$ and so on, until all rounds are complete.

But if we initialized the key to all 0s and we know the subkeys of each round, it is theoretically possible to ENCRYPT the ciphertext again to return back to the plaintext.

Practical:

I created a new file called “key0.py” to demonstrate the theory explained for a key with all 0 bits. It inherits all methods from the main “des.py” file.

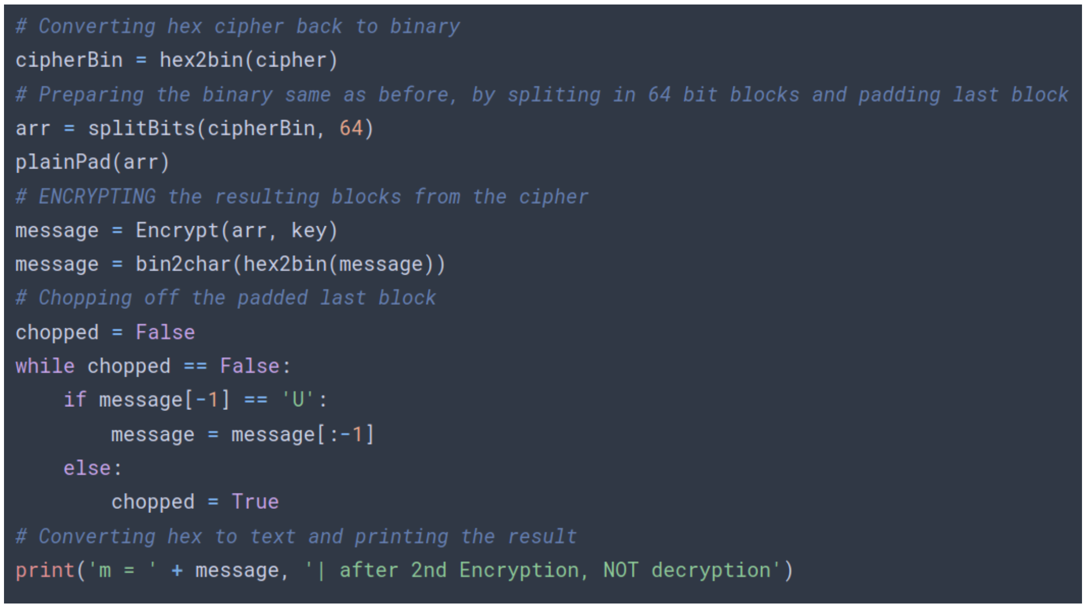
8



```
# random plaintext
text = 'Shuber Ali Mirza!@#'
# Key initialized to all 0s
key = '0' * 64
print('m = ' + text)
print('k = ' + key, len(key), 'bits')

# PC1 computation
key = permutate(key, 'pc1')
# Converting text to binary
binary = char2bin(text)
# Spliting binary into 64 bit blocks
arr = splitBits(binary, 64)
# Padding last block, to make it 64 bits, if required
plainPad(arr)
# Encrypting
cipher = Encrypt(arr, key)
print('c = ' + cipher)
```

- Figure 8: Initialized the key to 64-bit block of 0s
- Figure 8: Performed pc1 to make it 56 bits (though it's useless in this case)
- Figure 8: Followed the same methods used in "des.py" file to convert plaintext to binary, split into 64-bit blocks, pad last block if required, and encrypting the blocks using the key

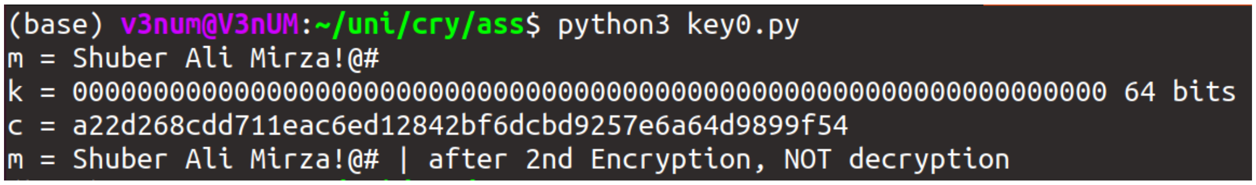
9



```
# Converting hex cipher back to binary
cipherBin = hex2bin(cipher)
# Preparing the binary same as before, by spliting in 64 bit blocks and padding last block
arr = splitBits(cipherBin, 64)
plainPad(arr)
# ENCRYPTING the resulting blocks from the cipher
message = Encrypt(arr, key)
message = bin2char(hex2bin(message))
# Chopping off the padded last block
chopped = False
while chopped == False:
    if message[-1] == 'U':
        message = message[:-1]
    else:
        chopped = True
# Converting hex to text and printing the result
print('m = ' + message, '| after 2nd Encryption, NOT decryption')
```

- Figure 9: Since returning value from the "Encrypt()" function is in hexadecimal, first converting that back to binary.
- Figure 9: Performing the same process with binary from hex as with binary from the plaintext.
- Figure 9: ENCRYPTING the binary from the cipher again, giving a new hexadecimal value.
- Figure 9: Converting new hex value back to English characters and chopping off the padding done to the binaries.

10



```
(base) v3num@V3nUM:~/uni/cry/ass$ python3 key0.py
m = Shuber Ali Mirza!@#
k = 0000000000000000000000000000000000000000000000000000000000000000 64 bits
c = a22d268cdd711eac6ed12842bf6dcbd9257e6a64d9899f54
m = Shuber Ali Mirza!@# | after 2nd Encryption, NOT decryption
```

- Figure 10: After running the file, output demonstrates that original plaintext was returned after second call to the "Encrypt()" function.

This proves that initializing key to all 0s results in a weak key for DES.