XOR KEY:
An XOR key is a sequence of bits used in the XOR cipher to encrypt and decrypt data. The XOR operation is applied between each bit of the plaintext and the corresponding bit of the key to produce the ciphertext. The same key is used for both encryption and decryption because the XOR operation is commutative and reversible.

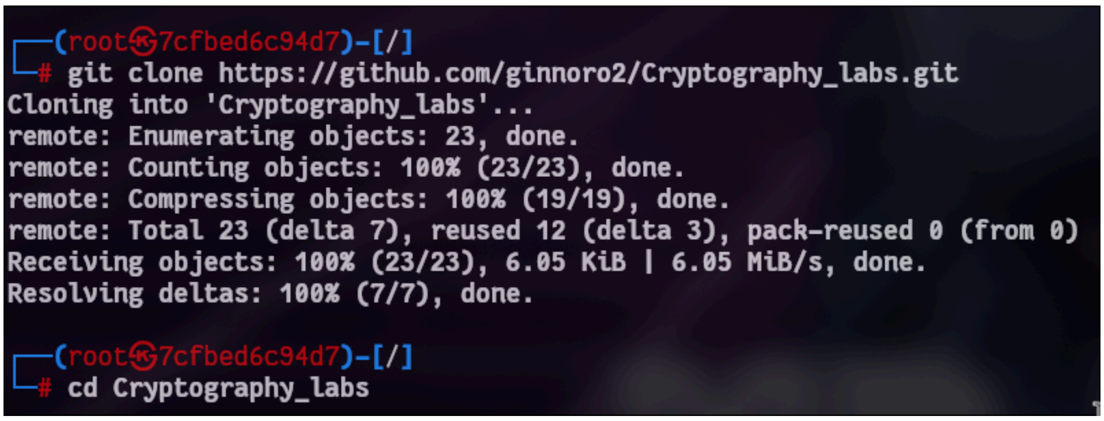
For example, if you have a plaintext message and a key, you can encrypt the message by performing the XOR operation on each bit of the message with the corresponding bit of the key. To decrypt the message, you perform the XOR operation again with the same key.

Here is an example of XOR encryption and decryption:

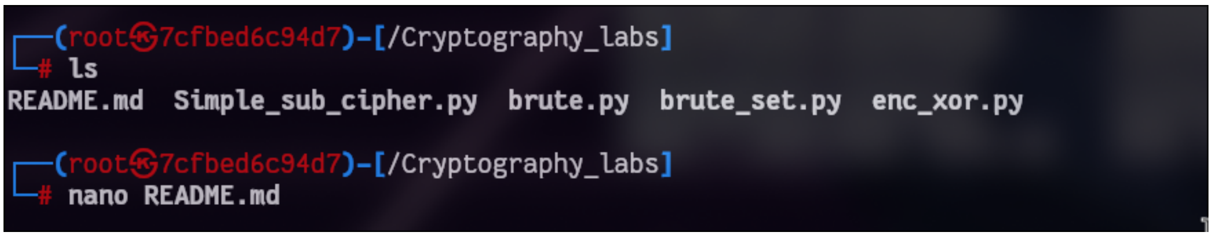
Plaintext: "My" (in binary: 01001101 01111001)
Key: "sec" (in binary: 11100000 01100001 01101101)
Result: 10101101 00011000 01001101 (after performing XOR)
To decrypt, you would use the same key to reverse the XOR operation, resulting in the original plaintext.

It's important to note that if the key is shorter than the plaintext, it is typically repeated to match the length of the plaintext. However, using a short, repeated key makes the encryption vulnerable to frequency analysis attacks.

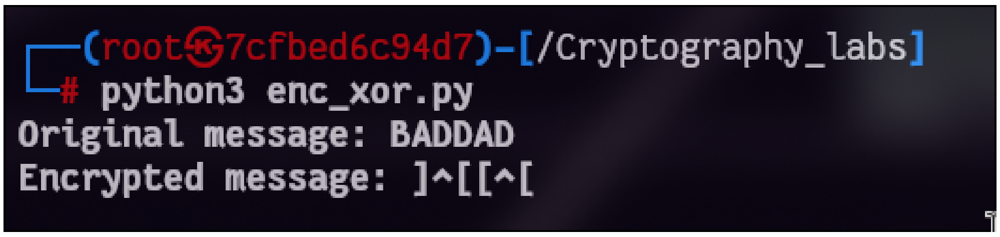
Encryption: P⊕K=C
Decryption: C⊕K=P
Where P is the plaintext, K is the key, and C is the ciphertext



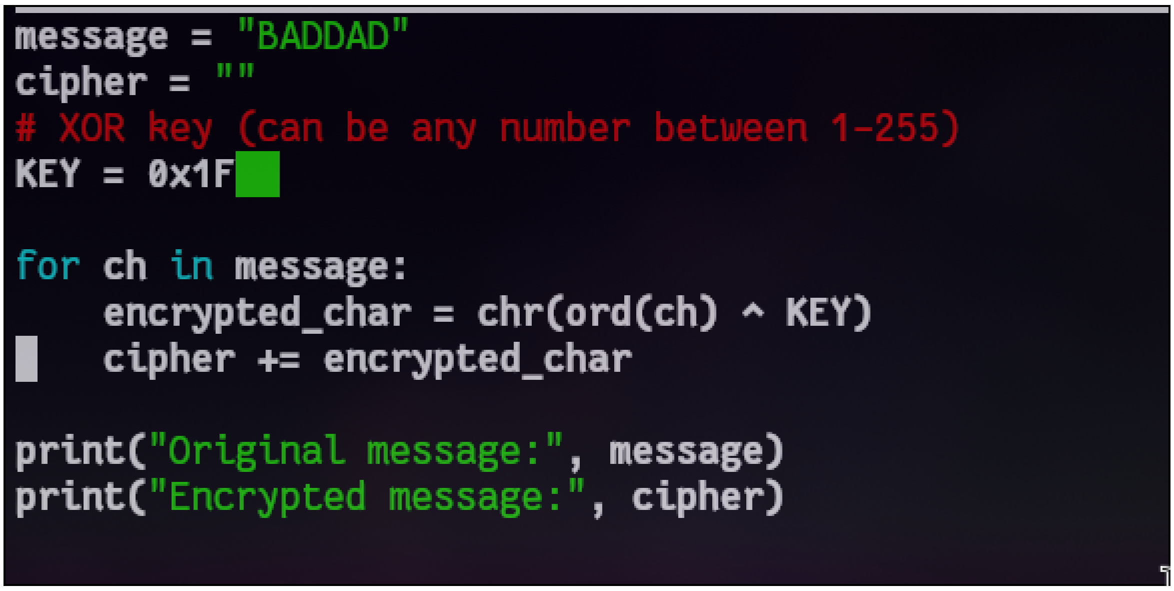
```
┌──(root㉿7cfbed6c94d7)-[/]
└─# git clone https://github.com/ginnoro2/Cryptography_labs.git
Cloning into 'Cryptography_labs'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 23 (delta 7), reused 12 (delta 3), pack-reused 0 (from 0)
Receiving objects: 100% (23/23), 6.05 KiB | 6.05 MiB/s, done.
Resolving deltas: 100% (7/7), done.

┌──(root㉿7cfbed6c94d7)-[/]
└─# cd Cryptography_labs
```



```
┌──(root㉿7cfbed6c94d7)-[/Cryptography_labs]
└─# ls
README.md  Simple_sub_cipher.py  brute.py  brute_set.py  enc_xor.py

┌──(root㉿7cfbed6c94d7)-[/Cryptography_labs]
└─# nano README.md
```



```
┌──(root㉿7cfbed6c94d7)-[/Cryptography_labs]
└─# python3 enc_xor.py
Original message: BADDAD
Encrypted message: ]^[[^[
```

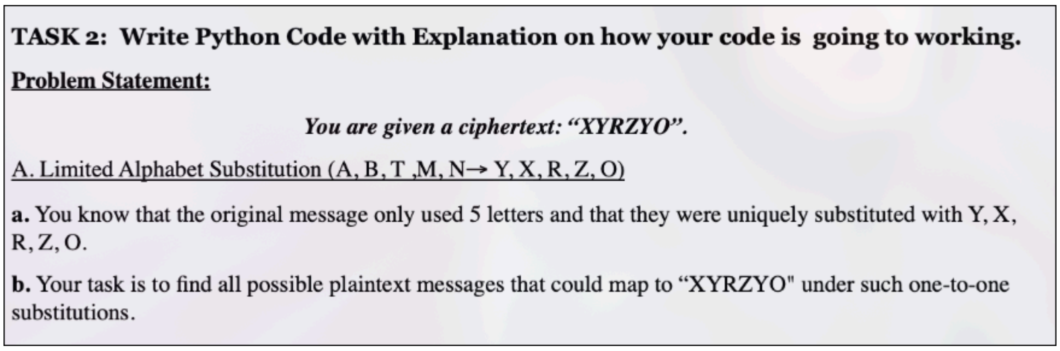


```
message = "BADDAD"
cipher = ""
# XOR key (can be any number between 1-255)
KEY = 0x1F

for ch in message:
    encrypted_char = chr(ord(ch) ^ KEY)
    cipher += encrypted_char

print("Original message:", message)
print("Encrypted message:", cipher)
```

How the Code Works
message = "BADDAD": The plaintext message you want to encrypt.
cipher = "": An empty string to store the encrypted message.
KEY = 0x1F: The encryption key, written in hexadecimal (0x1F = 31 in decimal). The key can be any integer between 1 and 255.

The for loop iterates over each character (ch) in the message.
ord(ch): Converts the character to its ASCII integer value.
ord(ch) ^ KEY: Applies the bitwise XOR operation between the character's ASCII value and the key. XOR (exclusive OR) outputs 1 only if the input bits are different. This operation is the core of the XOR cipher.
chr(...): Converts the result back to a character.
cipher += encrypted_char: Appends the encrypted character to the cipher string.

LIMITED ALPHABET SUBSTITUTION:



TASK 2: Write Python Code with Explanation on how your code is going to working.

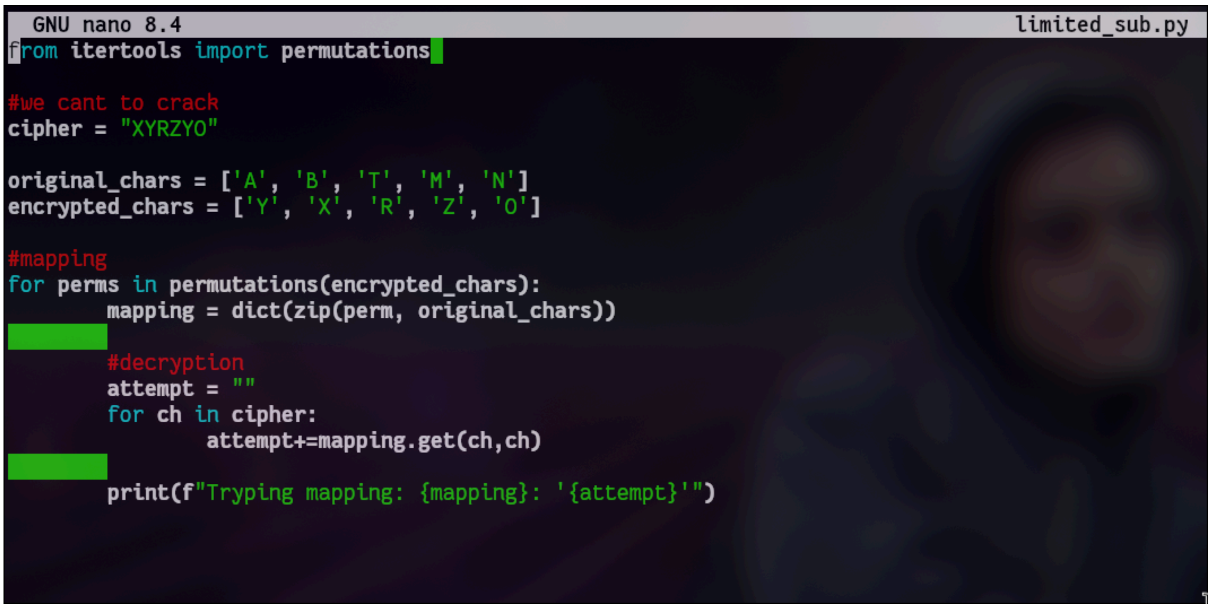
Problem Statement:

You are given a ciphertext: "XYRZYO".

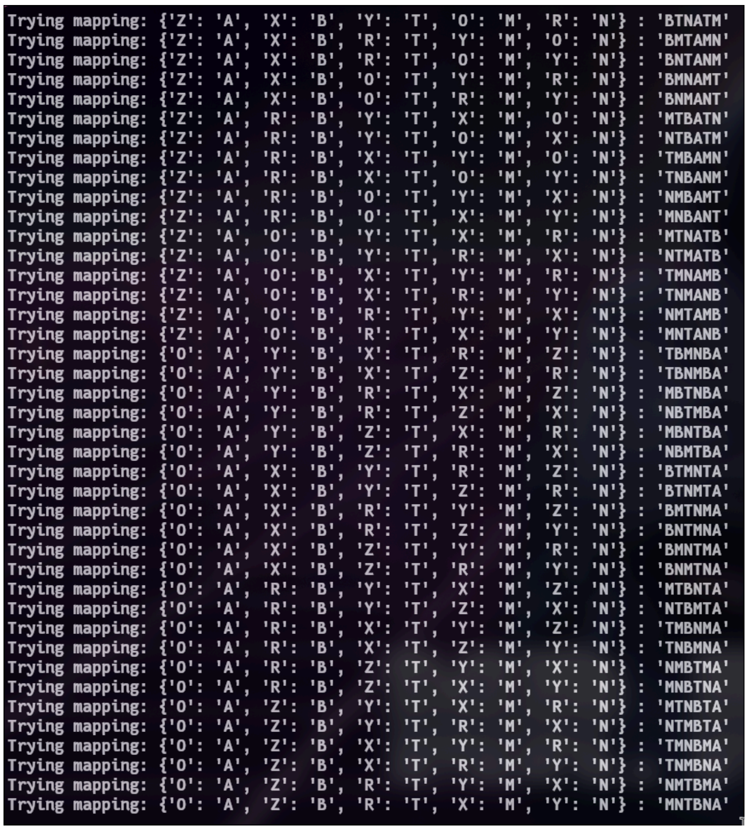
A. Limited Alphabet Substitution (A, B, T ,M, N→ Y, X, R, Z, O)

a. You know that the original message only used 5 letters and that they were uniquely substituted with Y, X, R, Z, O.

b. Your task is to find all possible plaintext messages that could map to "XYRZYO" under such one-to-one substitutions.



```
GNU nano 8.4                                                      limited_sub.py
from itertools import permutations

#we cant to crack
cipher = "XYRZYO"

original_chars = ['A', 'B', 'T', 'M', 'N']
encrypted_chars = ['Y', 'X', 'R', 'Z', 'O']

#mapping
for perms in permutations(encrypted_chars):
        mapping = dict(zip(perm, original_chars))

        #decryption
        attempt = ""
        for ch in cipher:
                attempt+=mapping.get(ch,ch)

        print(f"Trypinq mapping: {mapping}: '{attempt}'")
```



```
Trying mapping: {'Z': 'A', 'X': 'B', 'Y': 'T', 'O': 'M', 'R': 'N'} : 'BTNATM'
Trying mapping: {'Z': 'A', 'X': 'B', 'R': 'T', 'Y': 'M', 'O': 'N'} : 'BMTAMN'
Trying mapping: {'Z': 'A', 'X': 'B', 'R': 'T', 'O': 'M', 'Y': 'N'} : 'BNTANM'
Trying mapping: {'Z': 'A', 'X': 'B', 'O': 'T', 'Y': 'M', 'R': 'N'} : 'BMNAMT'
Trying mapping: {'Z': 'A', 'X': 'B', 'O': 'T', 'R': 'M', 'Y': 'N'} : 'BNMANT'
Trying mapping: {'Z': 'A', 'R': 'B', 'Y': 'T', 'X': 'M', 'O': 'N'} : 'MTBATN'
Trying mapping: {'Z': 'A', 'R': 'B', 'Y': 'T', 'O': 'M', 'X': 'N'} : 'NTBATM'
Trying mapping: {'Z': 'A', 'R': 'B', 'X': 'T', 'Y': 'M', 'O': 'N'} : 'TMBAMN'
Trying mapping: {'Z': 'A', 'R': 'B', 'X': 'T', 'O': 'M', 'Y': 'N'} : 'TNBANM'
Trying mapping: {'Z': 'A', 'R': 'B', 'O': 'T', 'Y': 'M', 'X': 'N'} : 'NMBAMT'
Trying mapping: {'Z': 'A', 'R': 'B', 'O': 'T', 'X': 'M', 'Y': 'N'} : 'MNBANT'
Trying mapping: {'Z': 'A', 'O': 'B', 'Y': 'T', 'X': 'M', 'R': 'N'} : 'MTNATB'
Trying mapping: {'Z': 'A', 'O': 'B', 'Y': 'T', 'R': 'M', 'X': 'N'} : 'NTMATB'
Trying mapping: {'Z': 'A', 'O': 'B', 'X': 'T', 'Y': 'M', 'R': 'N'} : 'TMNAMB'
Trying mapping: {'Z': 'A', 'O': 'B', 'X': 'T', 'R': 'M', 'Y': 'N'} : 'TNMANB'
Trying mapping: {'Z': 'A', 'O': 'B', 'R': 'T', 'Y': 'M', 'X': 'N'} : 'NMTAMB'
Trying mapping: {'Z': 'A', 'O': 'B', 'R': 'T', 'X': 'M', 'Y': 'N'} : 'MNTANB'
Trying mapping: {'O': 'A', 'Y': 'B', 'X': 'T', 'R': 'M', 'Z': 'N'} : 'TBMNBA'
Trying mapping: {'O': 'A', 'Y': 'B', 'X': 'T', 'Z': 'M', 'R': 'N'} : 'TBNMBA'
Trying mapping: {'O': 'A', 'Y': 'B', 'R': 'T', 'X': 'M', 'Z': 'N'} : 'MBTNBA'
Trying mapping: {'O': 'A', 'Y': 'B', 'R': 'T', 'Z': 'M', 'X': 'N'} : 'NBTMBA'
Trying mapping: {'O': 'A', 'Y': 'B', 'Z': 'T', 'X': 'M', 'R': 'N'} : 'MBNTBA'
Trying mapping: {'O': 'A', 'Y': 'B', 'Z': 'T', 'R': 'M', 'X': 'N'} : 'NBMTBA'
Trying mapping: {'O': 'A', 'X': 'B', 'Y': 'T', 'R': 'M', 'Z': 'N'} : 'BTMNTA'
Trying mapping: {'O': 'A', 'X': 'B', 'Y': 'T', 'Z': 'M', 'R': 'N'} : 'BTNMTA'
Trying mapping: {'O': 'A', 'X': 'B', 'R': 'T', 'Y': 'M', 'Z': 'N'} : 'BMTNMA'
Trying mapping: {'O': 'A', 'X': 'B', 'R': 'T', 'Z': 'M', 'Y': 'N'} : 'BNTMNA'
Trying mapping: {'O': 'A', 'X': 'B', 'Z': 'T', 'Y': 'M', 'R': 'N'} : 'BMNTMA'
Trying mapping: {'O': 'A', 'X': 'B', 'Z': 'T', 'R': 'M', 'Y': 'N'} : 'BNMTNA'
Trying mapping: {'O': 'A', 'R': 'B', 'Y': 'T', 'X': 'M', 'Z': 'N'} : 'MTBNTA'
Trying mapping: {'O': 'A', 'R': 'B', 'Y': 'T', 'Z': 'M', 'X': 'N'} : 'NTBMTA'
Trying mapping: {'O': 'A', 'R': 'B', 'X': 'T', 'Y': 'M', 'Z': 'N'} : 'TMBNMA'
Trying mapping: {'O': 'A', 'R': 'B', 'X': 'T', 'Z': 'M', 'Y': 'N'} : 'TNBMNA'
Trying mapping: {'O': 'A', 'R': 'B', 'Z': 'T', 'Y': 'M', 'X': 'N'} : 'NMBTMA'
Trying mapping: {'O': 'A', 'R': 'B', 'Z': 'T', 'X': 'M', 'Y': 'N'} : 'MNBTNA'
Trying mapping: {'O': 'A', 'Z': 'B', 'Y': 'T', 'X': 'M', 'R': 'N'} : 'MTNBTA'
Trying mapping: {'O': 'A', 'Z': 'B', 'Y': 'T', 'R': 'M', 'X': 'N'} : 'NTMBTA'
Trying mapping: {'O': 'A', 'Z': 'B', 'X': 'T', 'Y': 'M', 'R': 'N'} : 'TMNBMA'
Trying mapping: {'O': 'A', 'Z': 'B', 'X': 'T', 'R': 'M', 'Y': 'N'} : 'TNMBNA'
Trying mapping: {'O': 'A', 'Z': 'B', 'R': 'T', 'Y': 'M', 'X': 'N'} : 'NMTBMA'
Trying mapping: {'O': 'A', 'Z': 'B', 'R': 'T', 'X': 'M', 'Y': 'N'} : 'MNTBNA'
```

How does the code work?
The first line imports a function called permutations from Python's itertools library, which lets you generate all possible orderings of a list. Then, the main idea is that the loop goes through every possible way to match the encrypted letters to the original letters. zip() is a built-in Python function that takes two or more iterables (like lists or tuples) and pairs their elements together into tuples. Here, perm is a tuple

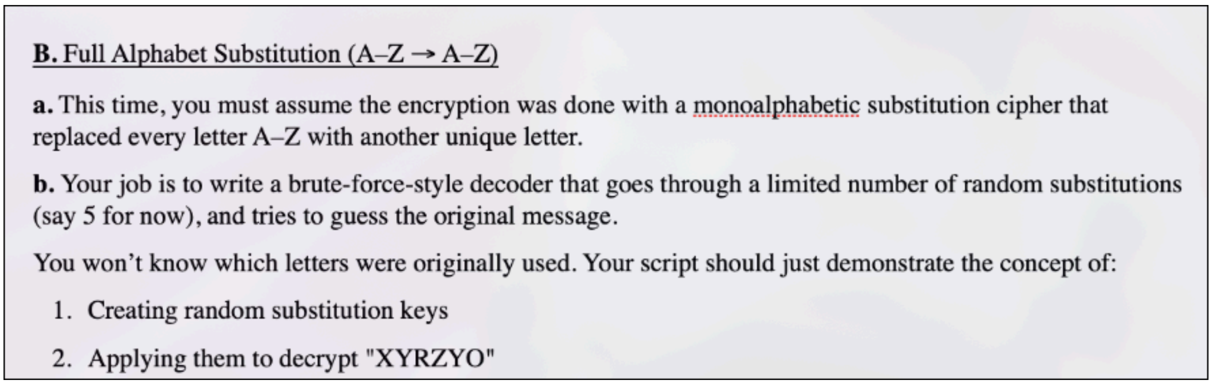
representing one permutation of the encrypted characters (e.g., ('X', 'Y', 'Z') or ('Y', 'Z', 'X')).

zip(perm, original_chars) pairs each element of perm with the corresponding element of original_chars.
Example:
perm = ('Y', 'Z', 'X')
original_chars = ['A', 'B', 'D']
zip(perm, original_chars)
('Y', 'A'), ('Z', 'B'), ('X', 'D')

dict() converts the zipped pairs into a dictionary.
The first element of each tuple becomes a key, and the second element becomes the corresponding value.
mapping = dict(zip(('Y', 'Z', 'X'), ['A', 'B', 'D']))
mapping = {'Y': 'A', 'Z': 'B', 'X': 'D'}

The expression mapping.get(ch, ch) uses the Python dictionary get() method, which works as follows:
It tries to find the value associated with the key ch in the dictionary mapping.
If the key ch exists in the dictionary, it returns the corresponding value.

FULL ALPHABETS RANDOM SUBSTITUTION:



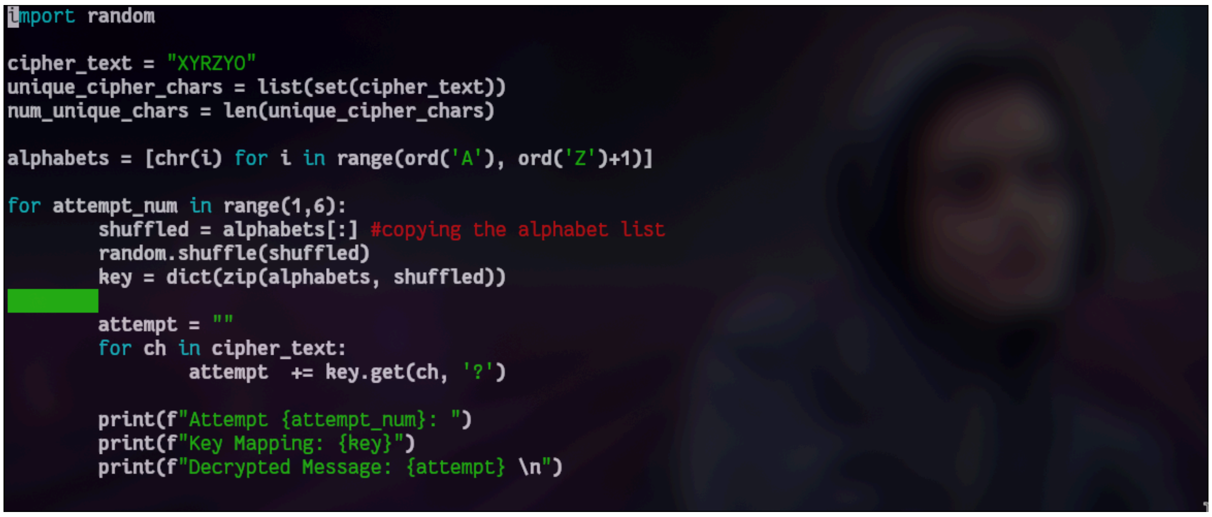
B. Full Alphabet Substitution (A–Z → A–Z)

a. This time, you must assume the encryption was done with a monoalphabetic substitution cipher that replaced every letter A–Z with another unique letter.

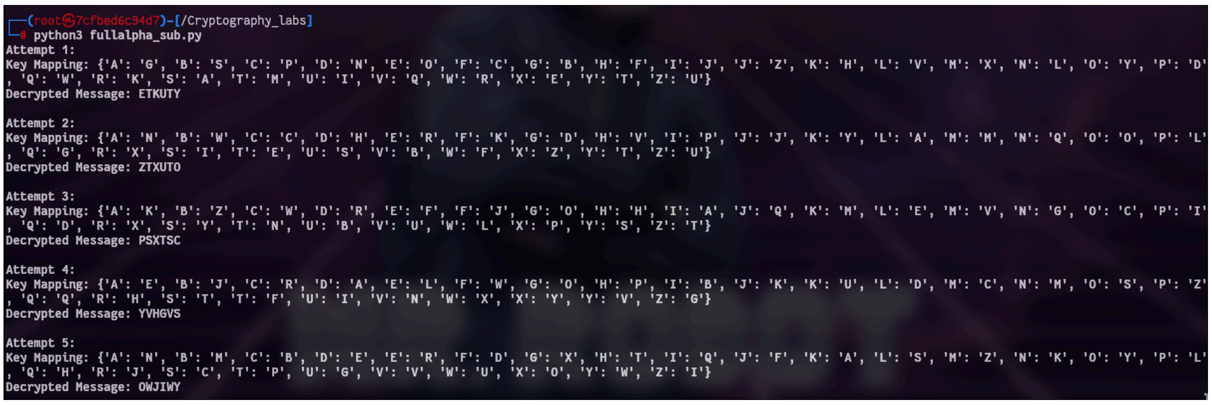
b. Your job is to write a brute-force-style decoder that goes through a limited number of random substitutions (say 5 for now), and tries to guess the original message.

You won't know which letters were originally used. Your script should just demonstrate the concept of:

1. Creating random substitution keys
2. Applying them to decrypt "XYRZYO"



```
import random

cipher_text = "XYRZYO"
unique_cipher_chars = list(set(cipher_text))
num_unique_chars = len(unique_cipher_chars)

alphabets = [chr(i) for i in range(ord('A'), ord('Z')+1)]

for attempt_num in range(1,6):
        shuffled = alphabets[:] #copying the alphabet list
        random.shuffle(shuffled)
        key = dict(zip(alphabets, shuffled))

        attempt = ""
        for ch in cipher_text:
                attempt  += key.get(ch, '?')

        print(f"Attempt {attempt_num}: ")
        print(f"Key Mapping: {key}")
        print(f"Decrypted Message: {attempt} \n")
```



```
┌──(root㉿7cfbed6c94d7)-[/Cryptography_labs]
└─# python3 fullalpha_sub.py
Attempt 1:
Key Mapping: {'A': 'G', 'B': 'S', 'C': 'P', 'D': 'N', 'E': 'O', 'F': 'C', 'G': 'B', 'H': 'F', 'I': 'J', 'J': 'Z', 'K': 'H', 'L': 'V', 'M': 'X', 'N': 'L', 'O': 'Y', 'P': 'D'
, 'Q': 'W', 'R': 'K', 'S': 'A', 'T': 'M', 'U': 'I', 'V': 'Q', 'W': 'R', 'X': 'E', 'Y': 'T', 'Z': 'U'}
Decrypted Message: ETKUTY

Attempt 2:
Key Mapping: {'A': 'N', 'B': 'W', 'C': 'C', 'D': 'H', 'E': 'R', 'F': 'K', 'G': 'D', 'H': 'V', 'I': 'P', 'J': 'J', 'K': 'Y', 'L': 'A', 'M': 'M', 'N': 'Q', 'O': 'O', 'P': 'L'
, 'Q': 'G', 'R': 'X', 'S': 'I', 'T': 'E', 'U': 'S', 'V': 'B', 'W': 'F', 'X': 'Z', 'Y': 'T', 'Z': 'U'}
Decrypted Message: ZTXUTO

Attempt 3:
Key Mapping: {'A': 'K', 'B': 'Z', 'C': 'W', 'D': 'R', 'E': 'F', 'F': 'J', 'G': 'O', 'H': 'H', 'I': 'A', 'J': 'Q', 'K': 'M', 'L': 'E', 'M': 'V', 'N': 'G', 'O': 'C', 'P': 'I'
, 'Q': 'D', 'R': 'X', 'S': 'Y', 'T': 'N', 'U': 'B', 'V': 'U', 'W': 'L', 'X': 'P', 'Y': 'S', 'Z': 'T'}
Decrypted Message: PSXTSC

Attempt 4:
Key Mapping: {'A': 'E', 'B': 'J', 'C': 'R', 'D': 'A', 'E': 'L', 'F': 'W', 'G': 'O', 'H': 'P', 'I': 'B', 'J': 'K', 'K': 'U', 'L': 'D', 'M': 'C', 'N': 'M', 'O': 'S', 'P': 'Z'
, 'Q': 'Q', 'R': 'H', 'S': 'T', 'T': 'F', 'U': 'I', 'V': 'N', 'W': 'X', 'X': 'Y', 'Y': 'V', 'Z': 'G'}
Decrypted Message: YVHGVS

Attempt 5:
Key Mapping: {'A': 'N', 'B': 'M', 'C': 'B', 'D': 'E', 'E': 'R', 'F': 'D', 'G': 'X', 'H': 'T', 'I': 'Q', 'J': 'F', 'K': 'A', 'L': 'S', 'M': 'Z', 'N': 'K', 'O': 'Y', 'P': 'L'
, 'Q': 'H', 'R': 'J', 'S': 'C', 'T': 'P', 'U': 'G', 'V': 'V', 'W': 'U', 'X': 'O', 'Y': 'W', 'Z': 'I'}
Decrypted Message: OWJIWY
```

Explanation:

The code creates an alphabet list once and, for five attempts, shuffles it to generate a random substitution key mapping each letter A-Z to a new letter. It then decrypts the ciphertext letter-by-letter using this key and prints both the mapping and the decrypted message. This approach helps explore different possible decryptions when the key is unknown.