Practical 1: Traditional symmetric Cipher

CAESAR CIPHER :

Cipher Description:

The Caesar cipher, also known as an additive cipher, is one of the simplest and most widely known encryption techniques. In this cipher, each letter in the plaintext is shifted a certain number of places down or up the alphabet.

Formulae:

Encryption

For a given plaintext letter PP, the corresponding ciphertext letter CC is calculated as: **C=(P+k)mod 26C=(P+k)mod26**

Decryption:

For a given ciphertext letter CC, the corresponding plaintext letter PP is calculated as: **P=(C−k)mod 26P=(C−k)mod26**

Strengths and Weaknesses:

Strengths:

1. **Simplicity**: The Caesar cipher is extremely simple to understand and implement. This makes it useful for educational purposes to introduce concepts of encryption and decryption.
2. **Speed**: Due to its simplicity, encryption and decryption using the Caesar cipher are very fast.

Weaknesses:

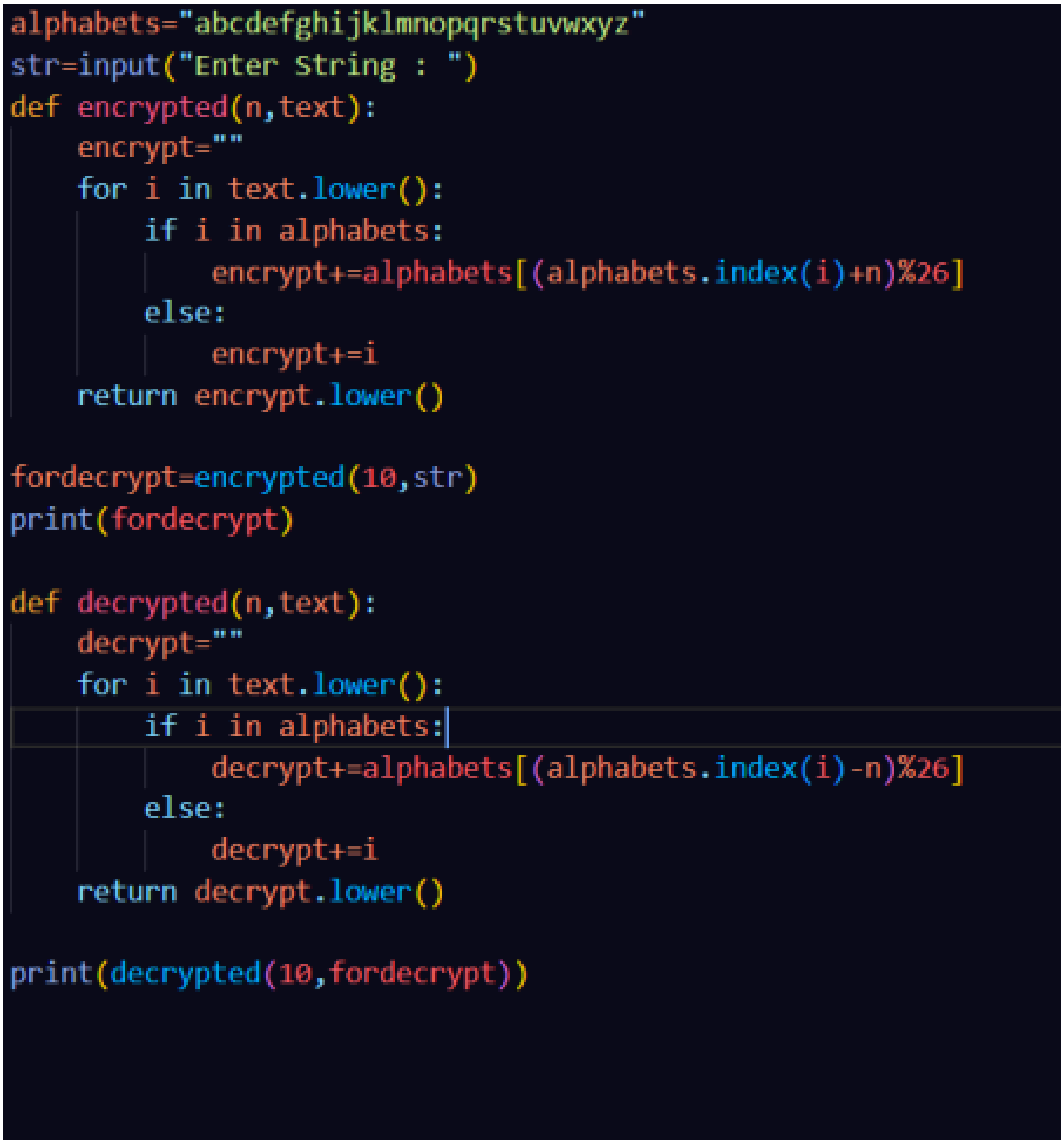
1. **Limited Key Space**: The Caesar cipher has only 25 possible keys (excluding the key 0, which would result in no encryption). This makes it highly vulnerable to brute force attacks, as an attacker can simply try all possible keys.
2. **Preservation of Letter Frequency**: The Caesar cipher does not alter the frequency distribution of letters in the plaintext. Therefore, frequency analysis can easily be used to break the cipher, especially for longer messages.
3. **Lack of Complexity**: The simplicity of the Caesar cipher means it does not provide sufficient security for any real-world application.

Possible Attacks:

1. **Brute Force Attack**: Because there are only 25 possible keys, an attacker can try each key in turn to decrypt the ciphertext. The correct key will result in a readable plaintext.
2. **Frequency Analysis**: This is a more sophisticated attack that exploits the fact that the Caesar cipher does not alter the frequency of letters. By analyzing the frequency
of letters in the ciphertext and comparing them to the expected frequency of letters in the language of the plaintext (e.g., English), an attacker can deduce the key and decrypt the message.

3. **Known Plaintext Attack**: If the attacker has access to a segment of plaintext and its corresponding ciphertext, they can easily determine the key by comparing the two.

INPUT:



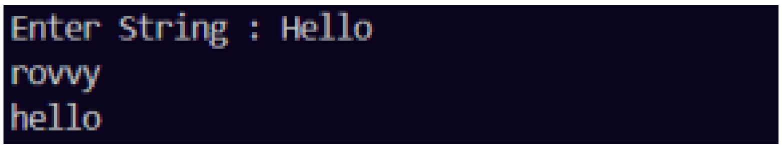
```python
alphabets="abcdefghijklmnopqrstuvwxyz"
str=input("Enter String : ")
def encrypted(n,text):
    encrypt=""
    for i in text.lower():
        if i in alphabets:
            encrypt+=alphabets[(alphabets.index(i)+n)%26]
        else:
            encrypt+=i
    return encrypt.lower()

fordecrypt=encrypted(10,str)
print(fordecrypt)

def decrypted(n,text):
    decrypt=""
    for i in text.lower():
        if i in alphabets:
            decrypt+=alphabets[(alphabets.index(i)-n)%26]
        else:
            decrypt+=i
    return decrypt.lower()

print(decrypted(10,fordecrypt))
```

OUTPUT:



```
Enter String : Hello
rovvy
hello
```

Autokey cipher:

Cipher Description:

The Autokey cipher is a polyalphabetic substitution cipher, which uses a keyword and the plaintext to create a running key for encryption and decryption. Unlike the Vigenère cipher, which uses a repeating keyword, the Autokey cipher appends the plaintext itself to the end of the keyword.

Formulae:

Encryption

For a given plaintext letter PiPi and a running key letter KiKi, the corresponding ciphertext letter CiCi is calculated as:**Ci=(Pi+Ki)mod 26Ci=(Pi+Ki)mod26**

Decryption

For a given ciphertext letter CiCi and a running key letter KiKi, the corresponding plaintext letter PiPi is calculated as:**Pi=(Ci−Ki+26)mod 26Pi=(Ci−Ki+26)mod26**

Strengths and Weaknesses:

Strengths:

1. **Enhanced Security**: The Autokey cipher is more secure than the Caesar cipher and Vigenère cipher due to its use of a running key that includes the plaintext, making frequency analysis more difficult.
2. **Polyalphabetic Nature**: The cipher uses multiple alphabets, reducing the risk of simple frequency analysis attacks.

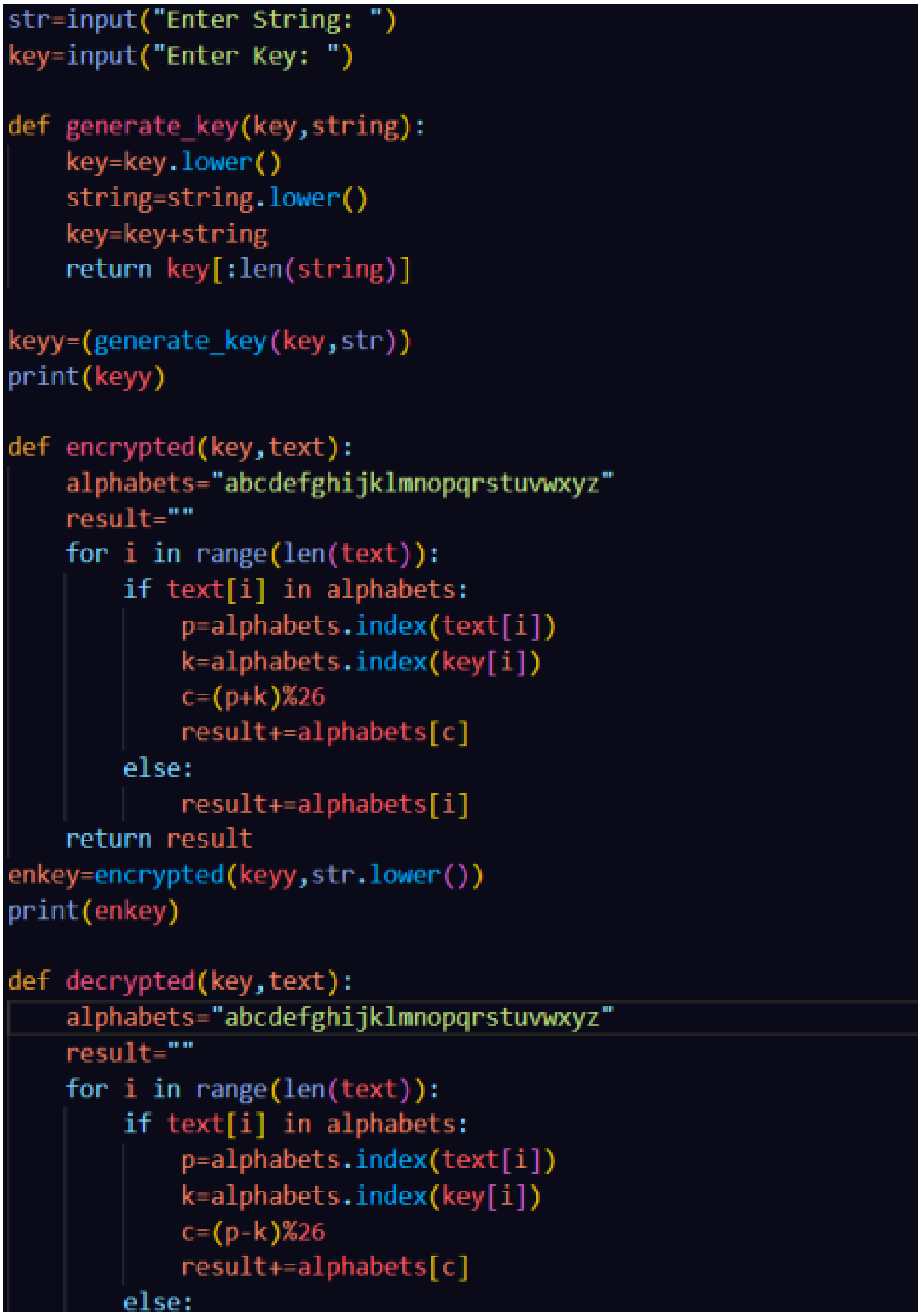
Weaknesses:

1. **Known Plaintext Attack**: If part of the plaintext is known, the corresponding portion of the key can be easily derived, compromising the security of the cipher.
2. **Complexity**: The cipher is more complex to implement and understand compared to simpler ciphers like the Caesar cipher.

Possible Attacks:

1. **Known Plaintext Attack**: If an attacker has a portion of the plaintext and its corresponding ciphertext, they can derive part of the running key and potentially decrypt the rest of the message.
2. **Frequency Analysis**: Though more resistant than simpler ciphers, frequency analysis can still be applied if the keyword is short or if the plaintext has predictable patterns.

Code:



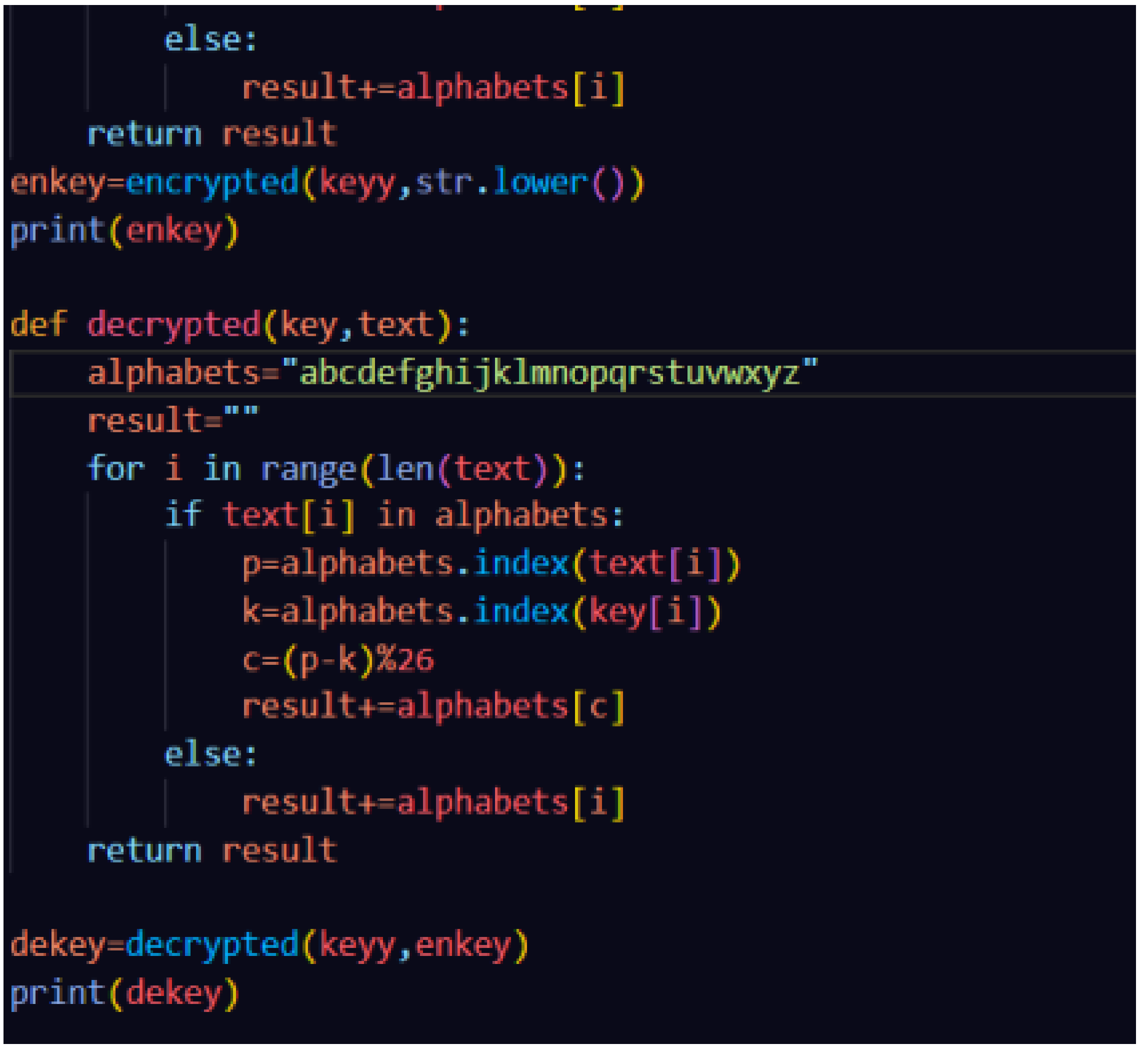
```python
str=input("Enter String: ")
key=input("Enter Key: ")

def generate_key(key,string):
    key=key.lower()
    string=string.lower()
    key=key+string
    return key[:len(string)]

keyy=(generate_key(key,str))
print(keyy)

def encrypted(key,text):
    alphabets="abcdefghijklmnopqrstuvwxyz"
    result=""
    for i in range(len(text)):
        if text[i] in alphabets:
            p=alphabets.index(text[i])
            k=alphabets.index(key[i])
            c=(p+k)%26
            result+=alphabets[c]
        else:
            result+=alphabets[i]
    return result
enkey=encrypted(keyy,str.lower())
print(enkey)

def decrypted(key,text):
    alphabets="abcdefghijklmnopqrstuvwxyz"
    result=""
    for i in range(len(text)):
        if text[i] in alphabets:
            p=alphabets.index(text[i])
            k=alphabets.index(key[i])
            c=(p-k)%26
            result+=alphabets[c]
        else:
```

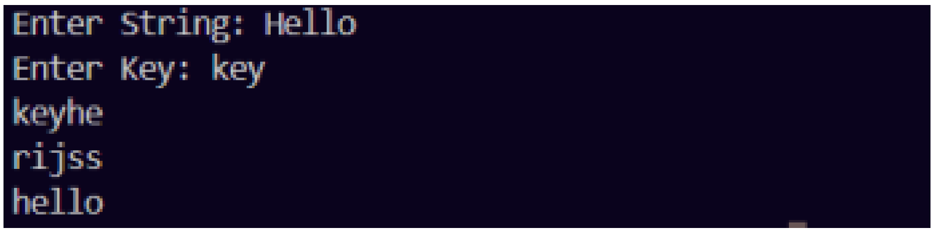


```
        else:
            result+=alphabets[i]
    return result
enkey=encrypted(keyy,str.lower())
print(enkey)

def decrypted(key,text):
    alphabets="abcdefghijklmnopqrstuvwxyz"
    result=""
    for i in range(len(text)):
        if text[i] in alphabets:
            p=alphabets.index(text[i])
            k=alphabets.index(key[i])
            c=(p-k)%26
            result+=alphabets[c]
        else:
            result+=alphabets[i]
    return result

dekey=decrypted(keyy,enkey)
print(dekey)
```

Output:



```
Enter String: Hello
Enter Key: key
keyhe
rijss
hello
```

Vigenère Cipher

Description:

The Vigenère cipher is a method of encrypting alphabetic text using a simple form of polyalphabetic substitution. It uses a keyword, where each letter of the keyword shifts corresponding letters of the plaintext by the alphabetical position of the keyword letter.

Formulae:

Encryption:

For a given plaintext letter PiPi and a keyword letter KiKi, the corresponding ciphertext letter CiCi is calculated as:**Ci=(Pi+Ki)mod 26Ci=(Pi+Ki)mod26**

Decryption:

For a given ciphertext letter CiCi and a keyword letter KiKi, the corresponding plaintext letter PiPi is calculated as:**Pi=(Ci−Ki+26)mod 26Pi=(Ci−Ki+26)mod26**

Strengths and Weaknesses:

Strengths:

1. **Increased Security**: The Vigenère cipher is more secure than simple substitution ciphers like the Caesar cipher because it uses a keyword to create multiple shifted alphabets.
2. **Polyalphabetic**: It employs multiple Caesar ciphers in sequence with different shifts based on the letters of a keyword.

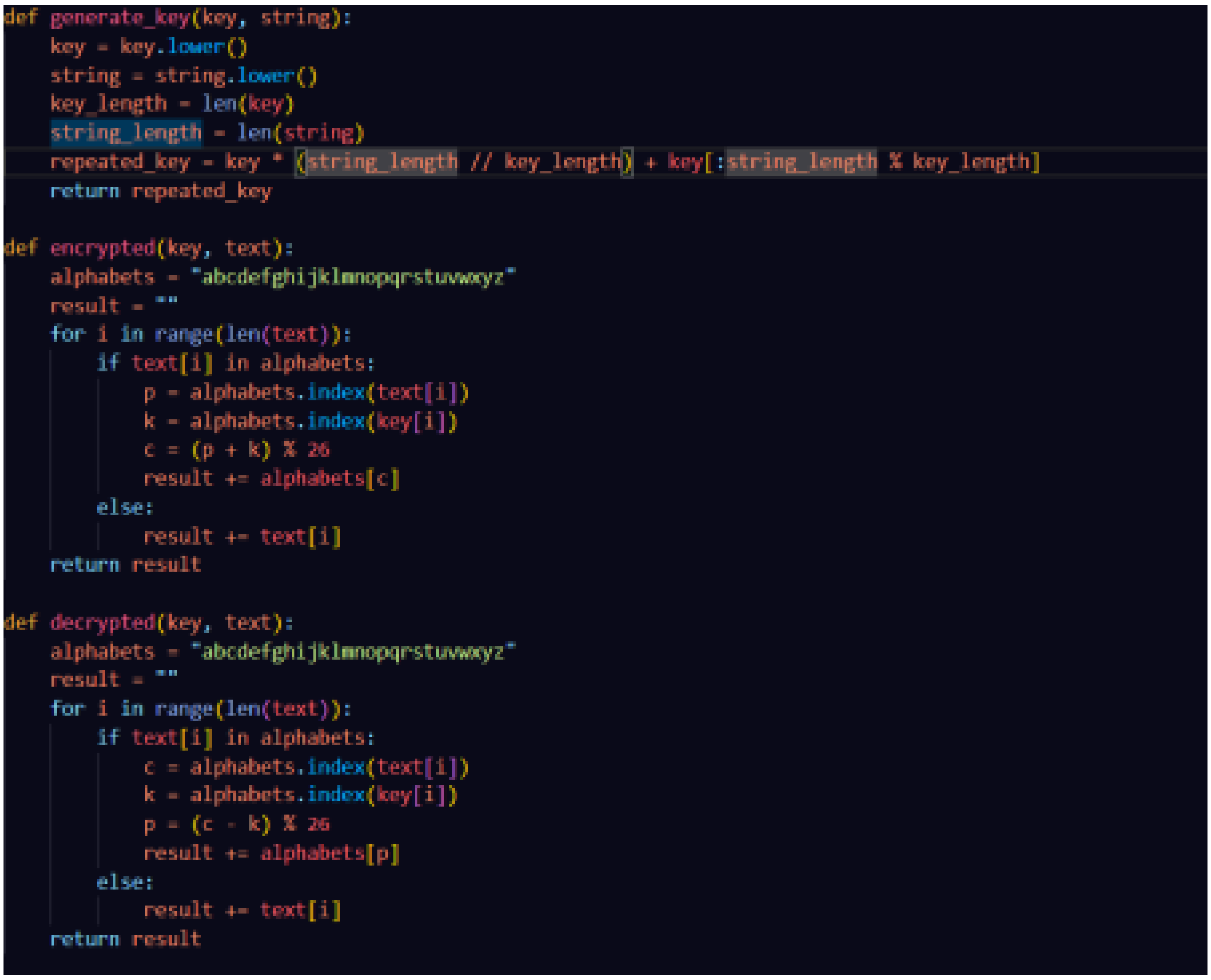
Weaknesses:

1. **Repetition of Keyword**: If the keyword is shorter than the plaintext, it repeats, which can introduce patterns that might be exploited.
2. **Frequency Analysis**: If the keyword length is known or guessed, the cipher can be broken into separate Caesar ciphers, each of which can be attacked using frequency analysis.

Possible Attacks:

1. **Kasiski Examination**: This method attempts to find repeating sequences of letters in the ciphertext that might indicate the length of the keyword.
2. **Frequency Analysis**: Once the keyword length is determined, the ciphertext can be divided into groups, each corresponding to a single alphabetic substitution. Frequency analysis can then be applied to each group individually.

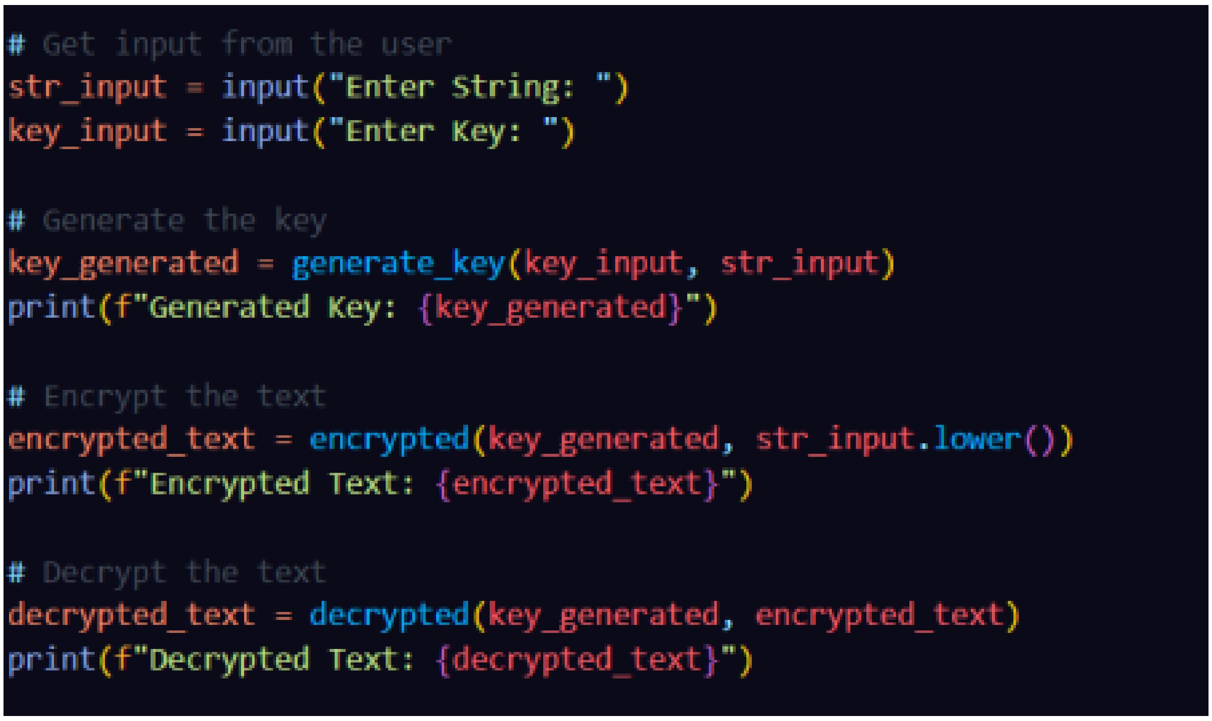
Code :



```
def generate_key(key, string):
    key = key.lower()
    string = string.lower()
    key_length = len(key)
    string_length = len(string)
    repeated_key = key * (string_length // key_length) + key[:string_length % key_length]
    return repeated_key

def encrypted(key, text):
    alphabets = "abcdefghijklmnopqrstuvwxyz"
    result = ""
    for i in range(len(text)):
        if text[i] in alphabets:
            p = alphabets.index(text[i])
            k = alphabets.index(key[i])
            c = (p + k) % 26
            result += alphabets[c]
        else:
            result += text[i]
    return result

def decrypted(key, text):
    alphabets = "abcdefghijklmnopqrstuvwxyz"
    result = ""
    for i in range(len(text)):
        if text[i] in alphabets:
            c = alphabets.index(text[i])
            k = alphabets.index(key[i])
            p = (c - k) % 26
            result += alphabets[p]
        else:
            result += text[i]
    return result
```



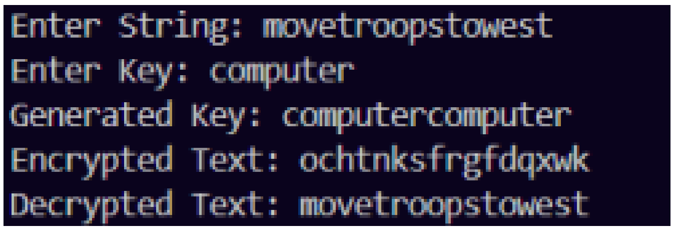
```
# Get input from the user
str_input = input("Enter String: ")
key_input = input("Enter Key: ")

# Generate the key
key_generated = generate_key(key_input, str_input)
print(f"Generated Key: {key_generated}")

# Encrypt the text
encrypted_text = encrypted(key_generated, str_input.lower())
print(f"Encrypted Text: {encrypted_text}")

# Decrypt the text
decrypted_text = decrypted(key_generated, encrypted_text)
print(f"Decrypted Text: {decrypted_text}")
```

Output:



```
Enter String: movetroopstowest
Enter Key: computer
Generated Key: computercomputer
Encrypted Text: ochtnksfrgfdqxwk
Decrypted Text: movetroopstowest
```