Devo decriptare questa stringa:

QWJhIHZ6b2VidHI2bmdyIHB1ciB6ciBhciBucHBiZXRi

QWJhIHZ6b2VidHI2bmdylHB1ciB6ciBhciBucHBiZXRi

Creo uno script python in cui importo le librerie delle tecniche di criptazione che mi interessa utilizzare.

Quali tecniche di criptazione scelgo?

Non ho una chiave, quindi non è criptato ma codificato/cifrato.

Cerco su internet i metodi più comuni, provo BASE64.

Da wikipedia:
Base64 è un sistema di codifica che consente la traduzione di dati binari (contenenti sequenze di 8 bit) in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi.

```python
import base64

# La tua stringa codificata in Base64
encoded_string = input("inserisci la stringa:\n")


# Decodifica Base64
decoded_bytes = base64.b64decode(encoded_string)  # Decodifica in bytes
decoded_string = decoded_bytes.decode('utf-8')    # Converte in testo leggibile

print("Testo decodificato:", decoded_string)
```

Con base64 importo la libreria base64, il resto sono metodi presenti nella documentazione.

E qui l'output:

```
marco@Fisso:~/python$ python3 decrypt.py
inserisci la stringa:
QWJhIHZ6b2VidHl2bmdyIHB1ciB6ciBhciBucHBiZXRi
Testo decodificato: Aba vzoebtyvngr pur zr ar nppbetb
```

Ci siamo quasi. Provo con dei cifrari generici e vedo tutti gli output. Guardando su internet si chiamano ROT cipher. Li provo tutti quanti, sono solo 25 (come le lettere dell'alfabeto)

Come funzionano questi cifrari? Sposto le lettere in avanti di un numero corrispondente al numero del ROT. Quindi in ROT1 la A diventa B e così via.

La differenza tra le lettere si chiama in inglese offset, lo spostamento da una all'altra è lo shift.

Creo quindi una funzione (apply_rot_cipher) che per ogni lettera fa lo shift necessario. La funzione prende in input text e shift. Text è la stringa acquisita tramite input utente

Lavoro quindi sui singoli caratteri. Uso ord(char) per ottenere il valore ascii corrispondente. Ad esempio A ha valore 65,B ha valore 66, a ha valore 97 e b ha valore 98

Quindi mi basta spostarmi in avanti delle lettere necessarie. Per fare questo il nuovo carattere avrà valore pari a questa formula:

```
    # Applica lo shift e rimappa all'interno dell'alfabeto
    new_char = chr((ord(char) - offset + shift) % 26 + offset)
```

Il modulo di 26 serve a rimanere entro l'intervallo desiderato. Vediamo cosa succede per la lettera W con ROT13.

char(W) è 87, tolgo l'offset (ovvero char(A) nel caso di lettere maiuscole cioè 65) e sommo 13. Il risultato sarà 35, che è un carattere che va oltre alle 26 lettere dell'alfabeto. Facendo il modulo ottengo il resto della divisione intera, cioè 9. Sommo 9 all'offset (65) e ottengo 74, che corrisponde alla lettera I (i maiuscola).

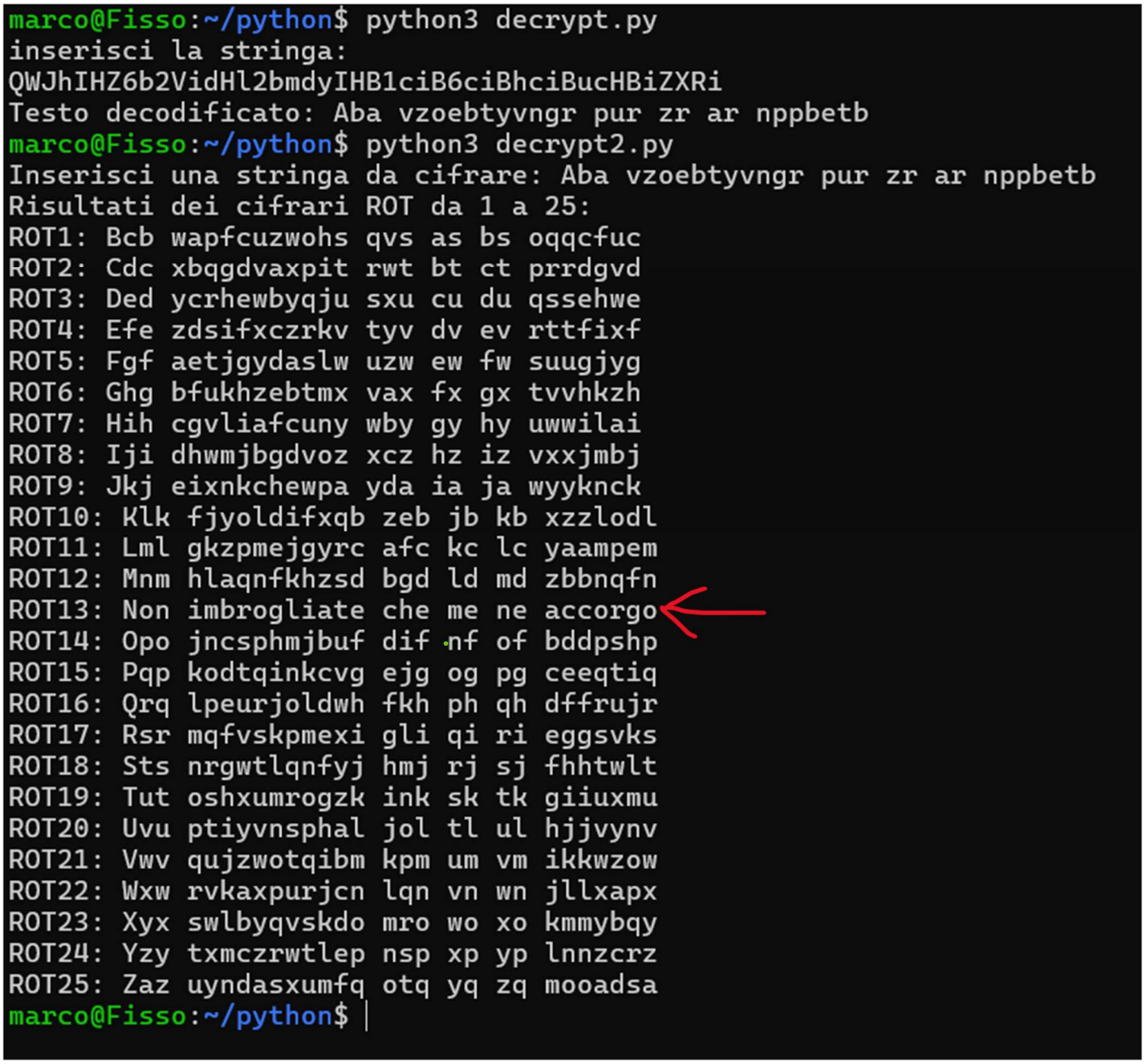
Questa operazione è inserita dentro ad un ciclo for che compone una stringa risultato a cui ad ogni step aggiunge o la lettera passata dentro rot13 oppure il carattere non alfabetico

```
def apply_rot_cipher(text, shift):
        #Applica il cifrario ROT con uno specifico shift.
    result = ""
    for char in text:
        if char.isalpha():
            # Determina se è maiuscola o minuscola
            offset = ord('A') if char.isupper() else ord('a')
            # Applica lo shift e rimappa all'interno dell'alfabeto
            new_char = chr((ord(char) - offset + shift) % 26 + offset)
            result += new_char
        else:
            # Mantiene caratteri non alfabetici invariati
            result += char
    return result

# Chiedi all'utente di inserire una stringa
input_string = input("Inserisci una stringa da cifrare: ")

# Applica i cifrari da ROT1 a ROT25
print("Risultati dei cifrari ROT da 1 a 25:")
for i in range(1, 26):
    rot_result = apply_rot_cipher(input_string, i)
    print(f"ROT{i}: {rot_result}")
```

Stampo quindi i 25 risultati



Ed ecco qui il risultato, codificato prima in rot13 e poi in base64
Non imbrogliate che me ne accorgo

Esercizio di oggi Criptazione e Firmatura con OpenSSLe Python: Obiettivi dell'esercizio:

- Generare chiavi RSA.
- Estrarre la chiave pubblica da chiave privata.
- Criptare e decriptare messaggi.
- Firmare e verificare messaggi. Strumenti utilizzati:
- OpenSSLper la generazione delle chiavi.
- Libreria cryptographyin Python.

Seguo le indicazioni e installo openssl, python3-pip e uso pip3 per installare cryptography.

Uso openssl per generare una chiave privata RSA 2048 bit sotto forma di file.pem

Creo i files encdec.py e firma.py

```
┌──(kali㉿kali)-[~]
└─$ touch encdec.py

┌──(kali㉿kali)-[~]
└─$ nano -m encdec.py

┌──(kali㉿kali)-[~]
└─$ touch firma.py

┌──(kali㉿kali)-[~]
└─$ nano -m firma.py
```

Guardiamo il codice e cerchiamo di capire cosa fa.

```
 GNU nano 8.1                                                encdec.py
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization
import base64

# Carica la chiave privata
with open('private_key.pem', 'rb') as key_file:
        private_key = serialization.load_pem_private_key(
                key_file.read(),
                password=None)


# Carica la chiave pubblica
with open('public_key.pem', 'rb') as key_file:
        public_key= serialization.load_pem_public_key(
                key_file.read())

message ='Ciao, Epicode spacca!'

# Criptazione con la chiave pubblica
encrypted= public_key.encrypt(message.encode(), padding.PKCS1v15())


# Decriptazione con la chiave privata
decrypted= private_key.decrypt( encrypted, padding.PKCS1v15())
print("Messaggio originale:", message)

print("Messaggio criptato:", base64.b64encode(encrypted).decode('utf-8') )
print("Messaggio decriptato:", decrypted.decode('utf-8') )
```

Questa parte di codice:

```
# Carica la chiave privata
with open('private_key.pem', 'rb') as key_file:
        private_key = serialization.load_pem_private_key(
                key_file.read(),
                password=None)
```

Carica il file private_key.pem creato precedentemente

Quest'altro invece carica la chiave pubblica:

```
# Carica la chiave pubblica
with open('public_key.pem', 'rb') as key_file:
        public_key= serialization.load_pem_public_key(
                key_file.read())
```

Il messaggio viene criptato con la chiave pubblica

```
message ='Ciao, Epicode spacca!'

# Criptazione con la chiave pubblica
encrypted= public_key.encrypt(message.encode(), padding.PKCS1v15())
```

E poi decriptato con la chiave privata:

```
# Decriptazione con la chiave privata
decrypted= private_key.decrypt( encrypted, padding.PKCS1v15())
print("Messaggio originale:", message)
```

Infine questa parte di codice:

```
print("Messaggio criptato:", base64.b64encode(encrypted).decode('utf-8') )
print("Messaggio decriptato:", decrypted.decode('utf-8') )
```

stampa:

Una rappresentazione codificata del messaggio criptato.

Il messaggio decodificato dopo la decriptazione, in un formato leggibile.

```
┌──(kali㉿kali)-[~]
└─$ python3 encdec.py
Messaggio originale: Ciao, Epicode spacca!
Messaggio criptato: e7kd2GFjkvLAA9×2Aq9769uHH80Rp409pRYiqMfhQvDTeAQlNEOOuo5PMOHSoBzsfKzQDMfJITy1V4U9TQ55T31rwwullTaOJ8G
fhLU85xgoW3JI6TJFbjI0WJyalLWkaSnYowqBJqzGHgjBNBE1jjGI7J5km+Vl+m1IqvpxGSklwHP8oligk5vzfqBXGeVKRfPpROcAkZ5Jo9RWamwMcllnZ2
NMcoxfo3Pbrm3u/S/Jy6zlZaMjdm+N7pmAKnZUBSxomxp9WDs43ohbKXeSFscF5VC1WJVZ3ZrIyv9yylJFANHAQ1TN5SRj1FTlaBupQdk7IfDscUr8B+h+X
GFRHw==
Messaggio decriptato: Ciao, Epicode spacca!
```

L'esercizio sulla firma è analogo

```python
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
import base64




# Carica la chiave privata
with open('private_key.pem', 'rb') as key_file:
        private_key = serialization.load_pem_private_key(
                key_file.read(),
                password=None)

# Carica la chiave pubblica
with open('public_key.pem', 'rb') as key_file:
        public_key = serialization.load_pem_public_key(key_file.read() )
message = 'Ciao, Epicode spacca!'

# Firma con chiave privata
signed = private_key.sign( message.encode(), padding.PKCS1v15(), hashes.SHA256() )

# Verifica della firma con la chiave pubblica
try:
        encrypted_b64 = base64.b64encode(signed).decode('utf-8')
        public_key.verify(signed, message.encode(), padding.PKCS1v15(), hashes.SHA256() )
        print("Base64 della firma:", encrypted_b64)
        print("Messaggio originale da confrontare:", message)
        print("La firma è valida.")
except Exception as e:
        print("La firma non è valida.", str(e))
```

```
┌──(kali㉿kali)-[~]
└─$ python3 firma.py
Base64 della firma: AFKgQFFbWgWidMunZukZo6heGPDAcm7nCIuwAr2eq6BvB7e/bRCqtl8FO7IbFecYd7LYmUCLgJGSGh3649461YJvCrP0Pg28t5X
pX1pGudYFLIkPwpGqPgVuZZsOvmNRvBVpvOZa//0i9G6WbYCVVI+FvCXtumkJCGmhJdKfMGSMBveJ3HgmYmWpIcXThdWb4Dc9vT0ZJMwzVOKAvVkz+PjMyL
vNLCPDgZAfFoaFUa1EZ9eEYDus0fWyNpcHsNvSEYYDULKGhWcUnpPwXYQLQGMgyWvjMPUWhFCXoBzWtei+hy4oAd6qaw4uVyF05JHjaAn+LekDGOQP4+/zy
Jsxtg==
Messaggio originale da confrontare: Ciao, Epicode spacca!
La firma è valida.
```