**# Caesar Cipher encryption-decryption in Python**

def encrypt(text, shift):

result = ""

for char in text:

if char.isupper():

result += chr((ord(char) + shift - 65) % 26 + 65)

elif char.islower():

result += chr((ord(char) + shift - 97) % 26 + 97)

else:

result += char

return result

def decrypt(cipher, shift):

return encrypt(cipher, -shift)

if \_\_name\_\_ == "\_\_main\_\_":

message = input("Enter the message: ")

shift = int(input("Enter shift value (e.g., 3): "))

encrypted = encrypt(message, shift)

decrypted = decrypt(encrypted, shift)

print("\nEncrypted Message:", encrypted)

print("Decrypted Message:", decrypted)

**#implement Mono-alphabetic Cipher Encryption – Decryption**

import string

def create\_key():

# This is a sample key; in real use you could randomize it

plain = string.ascii\_lowercase

cipher = "QWERTYUIOPASDFGHJKLZXCVBNM".lower()

return dict(zip(plain, cipher)), dict(zip(cipher, plain))

def encrypt(text, key\_map):

result = ""

for char in text:

if char.islower():

result += key\_map.get(char, char)

elif char.isupper():

result += key\_map.get(char.lower(), char).upper()

else:

result += char

return result

def decrypt(cipher\_text, reverse\_key\_map):

result = ""

for char in cipher\_text:

if char.islower():

result += reverse\_key\_map.get(char, char)

elif char.isupper():

result += reverse\_key\_map.get(char.lower(), char).upper()

else:

result += char

return result

if \_\_name\_\_ == "\_\_main\_\_":

plain\_to\_cipher, cipher\_to\_plain = create\_key()

message = input("Enter the message: ")

encrypted\_msg = encrypt(message, plain\_to\_cipher)

**#DES algorithm in python**

from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

import base64

def get\_key():

return b'8bytekey'

def encrypt(plain\_text, key):

cipher = DES.new(key, DES.MODE\_ECB) # Using ECB mode

padded\_text = pad(plain\_text.encode(), DES.block\_size)

encrypted\_bytes = cipher.encrypt(padded\_text)

return base64.b64encode(encrypted\_bytes).decode()

def decrypt(cipher\_text, key):

cipher = DES.new(key, DES.MODE\_ECB)

encrypted\_bytes = base64.b64decode(cipher\_text)

decrypted\_padded = cipher.decrypt(encrypted\_bytes)

return unpad(decrypted\_padded, DES.block\_size).decode()

if \_\_name\_\_ == "\_\_main\_\_":

key = get\_key()

message = input("Enter the message to encrypt: ")

encrypted = encrypt(message, key)

decrypted = decrypt(encrypted, key)

print("\nEncrypted:", encrypted)

print("Decrypted:", decrypted)

**#AES algorithm in python**

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

def aes\_encrypt(plaintext, key):

iv = get\_random\_bytes(AES.block\_size)

cipher = AES.new(key, AES.MODE\_CBC, iv)

ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block\_size))

return ciphertext, iv

def aes\_decrypt(ciphertext, key, iv):

cipher = AES.new(key, AES.MODE\_CBC, iv)

plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

return plaintext.decode()

if \_\_name\_\_ == "\_\_main\_\_":

key = get\_random\_bytes(32)

plaintext = "This is a secret message."

print(f"Original Plaintext: {plaintext}")

ciphertext, iv = aes\_encrypt(plaintext, key)

print(f"Ciphertext (hex): {ciphertext.hex()}")

decrypted\_text = aes\_decrypt(ciphertext, key, iv)

print(f"Decrypted Plaintext: {decrypted\_text}")

**#RC4 algorithm**

def rc4\_initialize(key):

S = list(range(256)) # State array initialization

j = 0

for i in range(256):

j = (j + S[i] + key[i % len(key)]) % 256

S[i], S[j] = S[j], S[i]

return S

def rc4\_generate\_keystream(S, length):

i = j = 0

keystream = []

for \_ in range(length):

i = (i + 1) % 256

j = (j + S[i]) % 256

S[i], S[j] = S[j], S[i]

keystream.append(S[(S[i] + S[j]) % 256])

return keystream

def rc4\_encrypt\_decrypt(data, key):

S = rc4\_initialize(key)

keystream = rc4\_generate\_keystream(S, len(data))

result = [data[i] ^ keystream[i] for i in range(len(data))]

return result

if \_\_name\_\_ == "\_\_main\_\_":

plaintext = "Hello, RC4!".encode

key = "SecretKey".encode()

print(f"Original Plaintext: {plaintext.decode()}")

ciphertext = rc4\_encrypt\_decrypt(plaintext, key)

print(f"Ciphertext (hex): {''.join(format(x, '02x') for x in ciphertext)}")

decrypted\_text = rc4\_encrypt\_decrypt(ciphertext, key)

print(f"Decrypted Plaintext: {bytes(decrypted\_text).decode()}")

**#RC5 algorithm**

import struct

W = 32 # Word size in bits

R = 12 # Number of rounds

B = 16 # Key size in bytes

P = 0xB7E15163 # Magic constant P

Q = 0x9E3779B9 # Magic constant Q

def rotate\_left(value, shift, bits=W):

return ((value << shift) & (2\*\*bits - 1)) | (value >> (bits - shift))

def rotate\_right(value, shift, bits=W):

return ((value >> shift) & (2\*\*bits - 1)) | (value << (bits - shift) & (2\*\*bits - 1))

def key\_schedule(key):

c = len(key) // (W // 8)

L = list(struct.unpack('<' + 'I' \* c, key.ljust(c \* (W // 8), b'\x00')))

t = 2 \* (R + 1)

S = [P]

for i in range(1, t):

S.append((S[i - 1] + Q) & (2\*\*W - 1))

i = j = A = B = 0

for \_ in range(3 \* max(t, c)):

A = S[i] = rotate\_left((S[i] + A + B) & (2\*\*W - 1), 3)

B = L[j] = rotate\_left((L[j] + A + B) & (2\*\*W - 1), (A + B) % W)

i = (i + 1) % t

**#half RC5 algorithm**

j = (j + 1) % c

return S

def rc5\_encrypt(plaintext, S):

A, B = struct.unpack('<II', plaintext)

A = (A + S[0]) & (2\*\*W - 1)

B = (B + S[1]) & (2\*\*W - 1)

for i in range(1, R + 1):

A = (rotate\_left(A ^ B, B % W) + S[2 \* i]) & (2\*\*W - 1)

B = (rotate\_left(B ^ A, A % W) + S[2 \* i + 1]) & (2\*\*W - 1)

return struct.pack('<II', A, B)

def rc5\_decrypt(ciphertext, S):

A, B = struct.unpack('<II', ciphertext)

for i in range(R, 0, -1):

B = rotate\_right((B - S[2 \* i + 1]) & (2\*\*W - 1), A % W) ^ A

A = rotate\_right((A - S[2 \* i]) & (2\*\*W - 1), B % W) ^ B

B = (B - S[1]) & (2\*\*W - 1)

A = (A - S[0]) & (2\*\*W - 1)

return struct.pack('<II', A, B)

if \_\_name\_\_ == "\_\_main\_\_":

key = b"SecretKey123456" # Must be exactly B bytes long

plaintext\_block = b"PlainTxt" # Must be exactly two words long (8 bytes)

print(f"Original Plaintext Block: {plaintext\_block}")

subkeys = key\_schedule(key)

ciphertext\_block = rc5\_encrypt(plaintext\_block.ljust(8, b'\x00'), subkeys)

print(f"Ciphertext Block: {ciphertext\_block.hex()}")

decrypted\_block = rc5\_decrypt(ciphertext\_block, subkeys)

print(f"Decrypted Plaintext Block: {decrypted\_block.strip().decode()}")

**#Blowfish in py**

from Crypto.Cipher import Blowfish

from Crypto.Util.Padding import pad, unpad

def blowfish\_encrypt(plaintext, key):

cipher = Blowfish.new(key, Blowfish.MODE\_ECB)

padded\_plaintext = pad(plaintext.encode(), Blowfish.block\_size)

ciphertext = cipher.encrypt(padded\_plaintext)

return ciphertext

def blowfish\_decrypt(ciphertext, key):

cipher = Blowfish.new(key, Blowfish.MODE\_ECB)

decrypted\_plaintext = unpad(cipher.decrypt(ciphertext), Blowfish.block\_size)

return decrypted\_plaintext.decode()

if \_\_name\_\_ == "\_\_main\_\_":

key = b"SecretKey123

plaintext = "Hello, Blowfish!"

print(f"Original Plaintext: {plaintext}")

ciphertext = blowfish\_encrypt(plaintext, key)

print(f"Ciphertext (hex): {ciphertext.hex()}")

decrypted\_text = blowfish\_decrypt(ciphertext, key)

print(f"Decrypted Plaintext: {decrypted\_text}")

**#hill cipher encryption**

import numpy as np

def mod\_inverse(a, m):

a = a % m

for x in range(1, m):

if (a \* x) % m == 1:

return x

return None

def preprocess\_text(text):

return ''.join([c.upper() for c in text if c.isalpha()])

def text\_to\_numbers(text):

return [ord(c) - ord('A') for c in text]

def numbers\_to\_text(numbers):

return ''.join([chr(num % 26 + ord('A')) for num in numbers])

def hill\_encrypt(plaintext, key\_matrix):

size = key\_matrix.shape[0]

plaintext = preprocess\_text(plaintext)

while len(plaintext) % size != 0:

plaintext += 'X'

numbers = text\_to\_numbers(plaintext)

ciphertext = []

for i in range(0, len(numbers), size):

block = np.array(numbers[i:i+size])

result = np.dot(key\_matrix, block) % 26

ciphertext.extend(result)

return numbers\_to\_text(ciphertext)

if \_\_name\_\_ == "\_\_main\_\_":

key\_matrix = np.array([[3, 3],

[2, 5]])

plaintext = input("Enter plaintext: ")

encrypted = hill\_encrypt(plaintext, key\_matrix)

print("\nEncrypted text:", encrypted)

**#polyalphabetic cipher**

def generate\_key(plaintext, key):

key = key.upper()

plaintext = plaintext.upper()

key\_extended = (key \* (len(plaintext) // len(key))) + key[:len(plaintext) % len(key)]

return key\_extende

def encrypt(plaintext, key):

key\_extended = generate\_key(plaintext, key)

ciphertext = []

for i in range(len(plaintext)):

if plaintext[i].isalpha():

shift = ord(key\_extended[i]) - ord('A')

encrypted\_char = chr((ord(plaintext[i].upper()) - ord('A') + shift) % 26 + ord('A'))

ciphertext.append(encrypted\_char)

else:

ciphertext.append(plaintext[i]) # Non-alphabet characters remain unchanged

return ''.join(ciphertext)

def decrypt(ciphertext, key):

key\_extended = generate\_key(ciphertext, key)

plaintext = []

for i in range(len(ciphertext)):

if ciphertext[i].isalpha():

shift = ord(key\_extended[i]) - ord('A')

**#half polyalphabetic cipher**

decrypted\_char = chr((ord(ciphertext[i].upper()) - ord('A') - shift) % 26 + ord('A'))

plaintext.append(decrypted\_char)

else:

plaintext.append(ciphertext[i]) # Non-alphabet characters remain unchanged

return ''.join(plaintext)

if \_\_name\_\_ == "\_\_main\_\_":

key = input("Enter key: ")

plaintext = input("Enter plaintext: ")

encrypted\_text = encrypt(plaintext, key)

print(f"Encrypted text: {encrypted\_text}")

decrypted\_text = decrypt(encrypted\_text, key)

print(f"Decrypted text: {decrypted\_text}")

**#Deffie-hellman exchange**

def power\_mod(base, exponent, modulus):

result = 1

base = base % modulus # Ensure base is within modulus

while exponent > 0:

if exponent % 2 == 1:

result = (result \* base) % modulus

exponent = exponent // 2

base = (base \* base) % modulus

return result

def diffie\_hellman(p, g, a, b):

p - prime modulus (public)

g - base (public)

a - private key of party A (private)

b - private key of party B (private)

A = power\_mod(g, a, p)

B = power\_mod(g, b, p)

shared\_secret\_A = power\_mod(B, a, p)

shared\_secret\_B = power\_mod(A, b, p)

assert shared\_secret\_A == shared\_secret\_B, "Shared secrets don't match!"

return shared\_secret\_A

if \_\_name\_\_ == "\_\_main\_\_": g

p = 23 # A small prime number

g = 5 # A primitive root modulo p

a = 6 # Party A's private key

b = 15 # Party B's private key

print(f"Public parameters: p = {p}, g = {g}")

print(f"Party A's private key: {a}")

print(f"Party B's private key: {b}")

shared\_secret = diffie\_hellman(p, g, a, b)

print(f"Shared secret key: {shared\_secret}")