**7.2.25 and 8.2.25 Lab program**

**1. Write a program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used**

**in reverse order. Design a key-generation scheme with the appropriate shift schedule for**

**the decryption process.**

**Code:**

ip = [58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7]

fp = [40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25]

pc1 = [57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18, 10, 2,

59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36, 63, 55, 47, 39,

31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37,

29, 21, 13, 5, 28, 20, 12, 4]

pc2 = [14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10, 23, 19, 12, 4,

26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55, 30, 40,

51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32]

shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

def permute(block, table):

return [block[i-1] for i in table]

def lshift(bits, n):

return bits[n:] + bits[:n]

def gkey(key):

key = permute(key, pc1)

l, r = key[:28], key[28:]

keys = []

for s in shifts:

l, r = lshift(l, s), lshift(r, s)

keys.append(perm ute(l + r, pc2))

return keys[::-1]

def feistel(r, k):

return [x ^ y for x, y in zip(r, k)]

def ddec(ct, key):

ct = permute(ct, ip)

l, r = ct[:32], ct[32:]

keys = gkey(key)

for i in range(16):

nr = [x ^ y for x, y in zip(l, feistel(r, keys[i]))]

l, r = r, nr

return permute(r + l, fp)

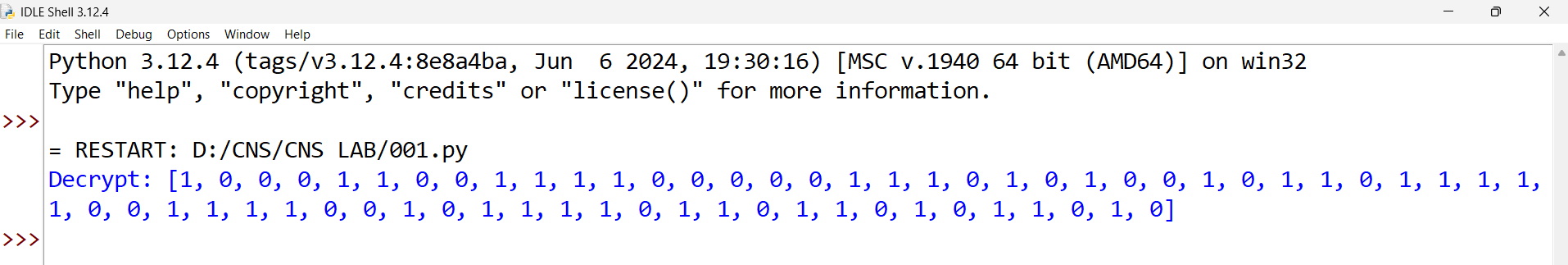
ct = [0, 1, 1, 0, 1, 0, 0, 1] \* 8

key = [1, 0, 1, 0, 0, 1, 1, 1] \* 8

pt = ddec(ct, key)

print("Decrypt:", pt)

**output:**



**2. Write a program for encryption in the cipher block chaining (CBC) mode using an**

**algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the**

**definition of CBC. Which of the two would you choose:**

1. **For security? b. For performance?**

**Code:**

def xor(b1, b2):

return [x ^ y for x, y in zip(b1, b2)]

def perm(b):

return b[::-1]

def dese(b, k):

return xor(perm(b), k)

def tdese(b, k):

e1 = dese(b, k[0])

d1 = dese(e1, k[1])

e2 = dese(d1, k[2])

return e2

def tcbc(pt, k, iv):

blks = [pt[i:i+64] for i in range(0, len(pt), 64)]

ct = []

prev = iv

for b in blks:

b = xor(b, prev)

c = tdese(b, k)

ct.append(c)

prev = c

return ct

pt = [0, 1, 1, 0, 1, 0, 0, 1] \* 8

k = [

[1, 0, 1, 0, 0, 1, 1, 1] \* 8,

[0, 1, 0, 1, 1, 0, 0, 1] \* 8,

[1, 1, 0, 0, 1, 0, 1, 0] \* 8

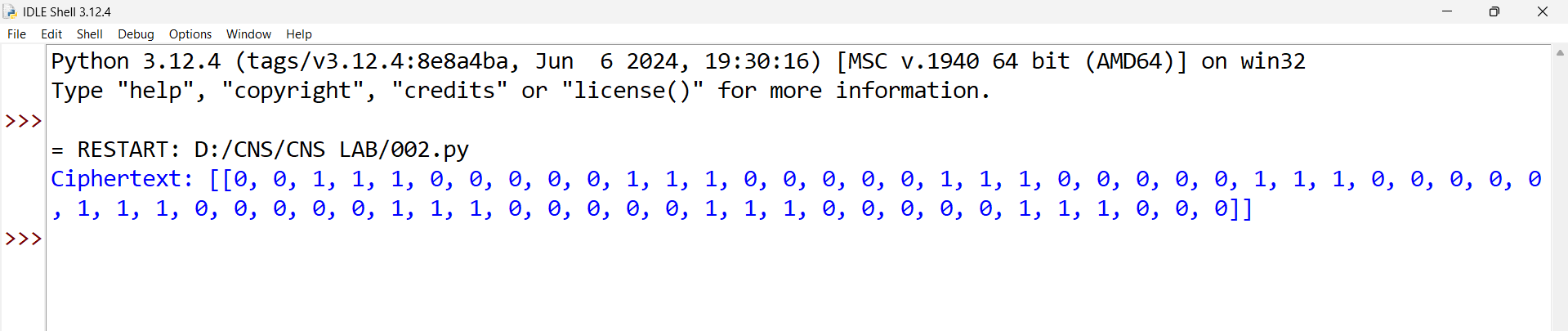
]

iv = [1, 0, 0, 1, 1, 0, 1, 0] \* 8

ct = tcbc(pt, k, iv)

print("Ciphertext:", ct)

**Output:**



**3. Write a program for ECB, CBC, and CFB modes, the plaintext must be a sequence of**

**one or more complete data blocks (or, for CFB mode, data segments). In other words, for**

**these three modes, the total number of bits in the plaintext must be a positive multiple of**

**the block (or segment) size. One common method of padding, if needed, consists of a 1**

**bit followed by as few zero bits, possibly none, as are necessary to complete the final**

**block. It is considered good practice for the sender to pad every message, including**

**messages in which the final message block is already complete. What is the motivation**

**for including a padding block when padding is not needed?**

**Code:**

def xor(b1, b2):

return [x ^ y for x, y in zip(b1, b2)]

def perm(b):

return b[::-1]

def dese(b, k):

return xor(perm(b), k)

def pad(pt, blk):

plen = len(pt)

padbits = blk - (plen % blk)

return pt + [1] + [0] \* (padbits - 1)

def ecbe(pt, k):

pt = pad(pt, 64)

blks = [pt[i:i+64] for i in range(0, len(pt), 64)]

return [dese(b, k) for b in blks]

def cbce(pt, k, iv):

pt = pad(pt, 64)

blks = [pt[i:i+64] for i in range(0, len(pt), 64)]

ct, prev = [], iv

for b in blks:

b = xor(b, prev)

c = dese(b, k)

ct.append(c)

prev = c

return ct

def cfbe(pt, k, iv, seg):

pt = pad(pt, seg)

ct, prev = [], iv

for i in range(0, len(pt), seg):

segb = pt[i:i+seg]

enciv = dese(prev, k)[:seg]

c = xor(segb, enciv)

ct.append(c)

prev = c

return ct

pt = [0, 1, 1, 0, 1, 0, 0, 1] \* 8

k = [1, 0, 1, 0, 0, 1, 1, 1] \* 8

iv = [1, 0, 0, 1, 1, 0, 1, 0] \* 8

ctecb = ecbe(pt, k)

ctcbc = cbce(pt, k, iv)

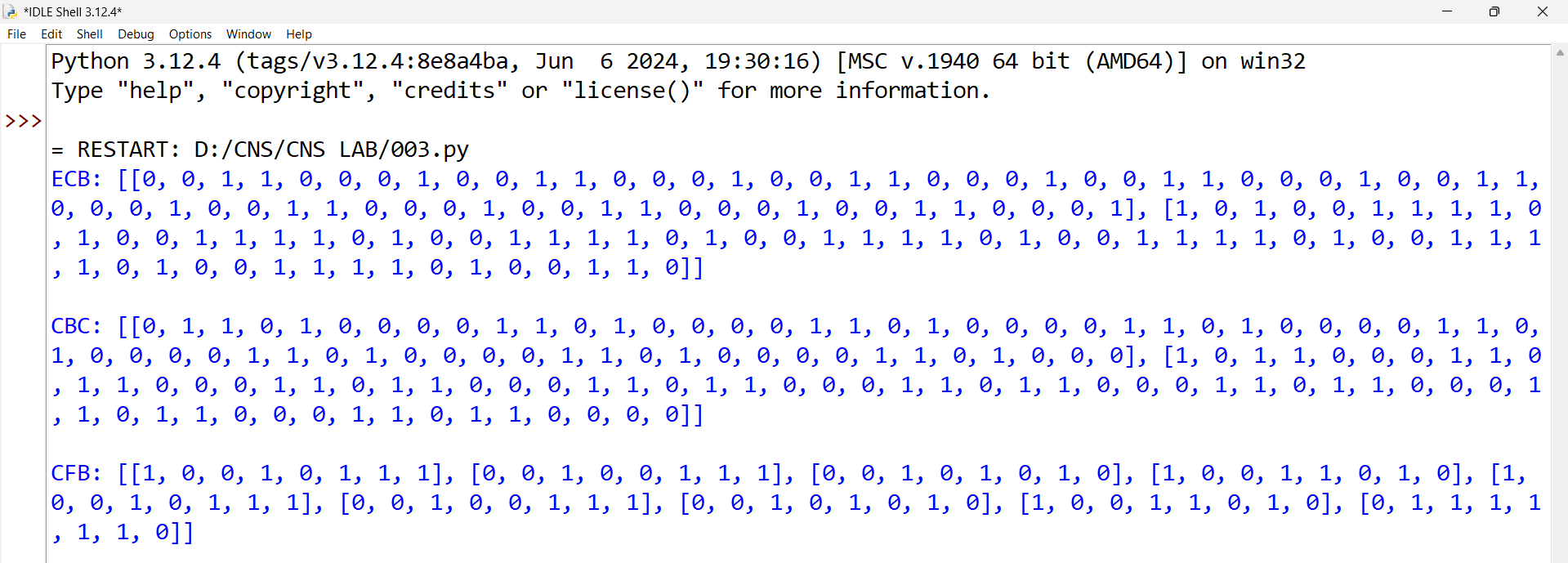
ctcfb = cfbe(pt, k, iv, 8)

print("ECB:", ctecb, "\n")

print("CBC:", ctcbc, "\n")

print("CFB:", ctcfb, "\n")

**Output:**



**4. Write a program for Encrypt and decrypt in cipher block chaining mode using one of the**

**following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-**

**DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001**

**0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of**

**1111 0100 0000 1011. Decryption should work correspondingly.**

**Code:**

def xor(b1, b2):

return [x ^ y for x, y in zip(b1, b2)]

def se(b, k):

return xor(b[::-1], k)

def sd(b, k):

return xor(b[::-1], k)

def pad(pt, blk):

plen = len(pt)

padbits = blk - (plen % blk)

return pt + [1] + [0] \* (padbits - 1)

def cbce(pt, k, iv):

pt = pad(pt, 8)

blks = [pt[i:i+8] for i in range(0, len(pt), 8)]

ct, prev = [], iv

for b in blks:

b = xor(b, prev)

c = sd(b, k)

ct.append(c)

prev = c

return ct

def cbcd(ct, k, iv):

pt, prev = [], iv

for c in ct:

d = sd(c, k)

p = xor(d, prev)

pt.append(p)

prev = c

return pt

pt = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1]

k = [0, 1, 1, 1, 1, 1, 1, 1, 0, 1]

iv = [1, 0, 1, 0, 1, 0, 1, 0]

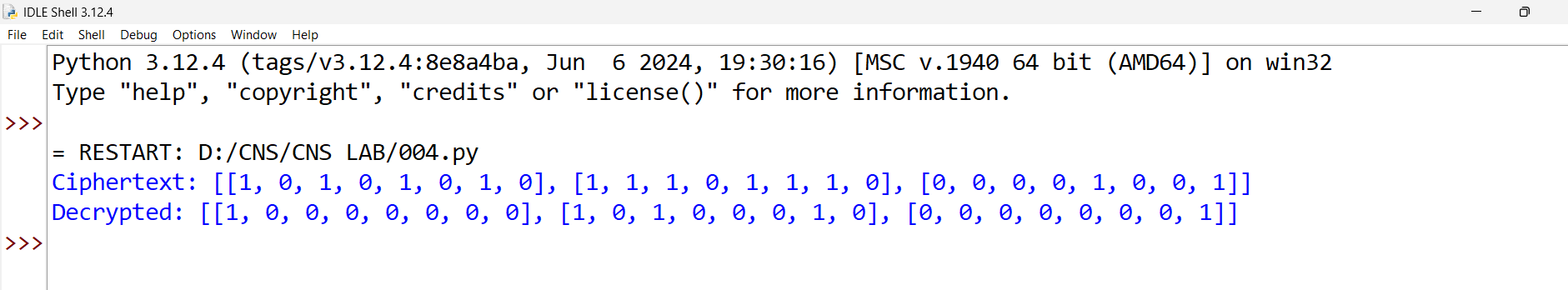
ctcbc = cbce(pt, k, iv)

ptcbc = cbcd(ctcbc, k, iv)

print("Ciphertext:", ctcbc)

print("Decrypted:", ptcbc)

**Output:**



**5. Write a program for RSA system, the public key of a given user is e = 31, n = 3599. What**

**is the private key of this user? Hint: First use trial-and-error to determine p and q; then**

**use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).**

**code:**

def gcd(a, b):

while b:

a, b = b, a % b

return a

def exteuclid(a, b):

x0, x1, y0, y1 = 1, 0, 0, 1

while b:

q = a // b

a, b = b, a % b

x0, x1 = x1, x0 - q \* x1

y0, y1 = y1, y0 - q \* y1

return x0

def modinv(e, phi):

d = exteuclid(e, phi)

return d % phi

def findpq(n):

for p in range(2, int(n\*\*0.5) + 1):

if n % p == 0:

q = n // p

return p, q

return None, None

e = 31

n = 3599

p, q = findpq(n)

phi = (p - 1) \* (q - 1)

d = modinv(e, phi)

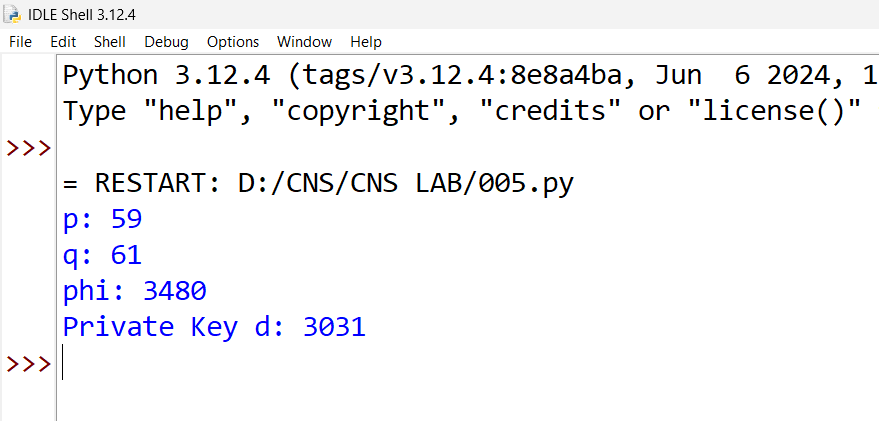
print("p:", p)

print("q:", q)

print("phi:", phi)

print("Private Key d:", d)

**output:**



**6. Write a program for Diffie-Hellman protocol, each participant selects a secret number x**

**and sends the other participant ax mod q for some public number a. What would happen**

**if the participants sent each other xa for some public number a instead? Give at least one**

**method Alice and Bob could use to agree on a key. Can Eve break your system without**

**finding the secret numbers? Can Eve find the secret numbers?**

**Code:**

def pmod(b, e, m):

r = 1

while e > 0:

if e % 2 == 1:

r = (r \* b) % m

b = (b \* b) % m

e //= 2

return r

q = 23

a = 5

xa = 6

xb = 15

ya = pmod(a, xa, q)

yb = pmod(a, xb, q)

ka = pmod(yb, xa, q)

kb = pmod(ya, xb, q)

print("Alice key:", ka)

print("Bob key:", kb)

**output:**

