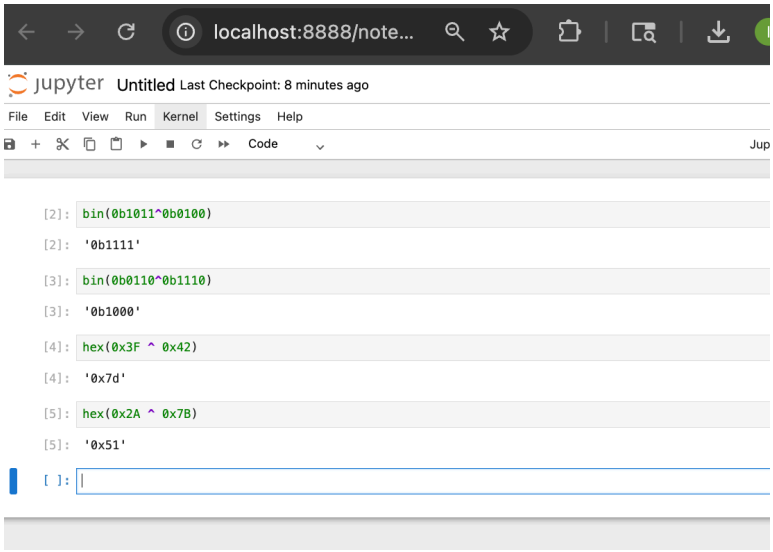
IOT InClass_Exercise-7
Lakshminath Reddy Alamuru
SUID: 367982169
Email ID: lalamuru@syr.edu

1. Tested the operation using python in the Jupyter notebook



Statement1	Operation	Statement2	Output
0b1011	XOR	0b100	0b1111
0b0110	XOR	0b1110	0b1001
x3F	XOR	X42	x7d
x2A	XOR	x7B	X51

2. Written the code below for the task-2 for the encryption and decryption and got the values as expected.

[19]:
```python
import random

def generate_keystream(key, length):
    """Generates a pseudorandom keystream using the given key."""
    random.seed(key)  # initialize PRNG with the seed
    keystream = bytes([random.randint(0, 255) for _ in range(length)])
    return keystream

def encrypt(plaintext: bytes, key: int) -> bytes:
    """Encrypt plaintext using XOR with a generated keystream."""
    keystream = generate_keystream(key, len(plaintext))
    ciphertext = bytes([p ^ k for p, k in zip(plaintext, keystream)])
    return ciphertext

def decrypt(ciphertext: bytes, key: int) -> bytes:
    """Decrypt ciphertext (same operation as encryption)."""
    return encrypt(ciphertext, key)

# Example data
key = 0b01011101  # Seed for PRNG
plaintext = b'\xB5'  # 0b10110101 as one byte

# Encryption
ciphertext = encrypt(plaintext, key)

# Decryption
decrypted_text = decrypt(ciphertext, key)

print("Plaintext (bits) :", format(plaintext[0], '08b'))
print("Ciphertext (bits):", format(ciphertext[0], '08b'))
print("Decrypted (bits) :", format(decrypted_text[0], '08b'))
```

```
Plaintext (bits) : 10110101
Ciphertext (bits): 01001000
Decrypted (bits) : 10110101
```

How Long Key Streams Are Generated?

In real cryptographic stream ciphers (like RC4, ChaCha20, etc.), the key stream isn't manually defined, it's generated by a CSPRNG (Cryptographically Secure Pseudo-Random Number Generator) that expands a short key (the seed) into a long, unpredictable bit stream

3. For task-3, written the code and complied for the plaintext.

[20]:
```python
import os

def generate_keystream(length: int) -> bytes:
    """Generate a secure pseudorandom keystream using OS CSPRNG."""
    return os.urandom(length)  # cryptographically secure random bytes

def encrypt(plaintext: bytes, keystream: bytes) -> bytes:
    """Encrypt plaintext using XOR with the keystream."""
    return bytes([p ^ k for p, k in zip(plaintext, keystream)])

def decrypt(ciphertext: bytes, keystream: bytes) -> bytes:
    """Decrypt ciphertext (same operation as encryption)."""
    return encrypt(ciphertext, keystream)

# Example plaintext
plaintext = b"HELLO"  # 5 bytes of plaintext

# Step 1: Generate secure keystream of equal length
keystream = generate_keystream(len(plaintext))

# Step 2: Encrypt plaintext
ciphertext = encrypt(plaintext, keystream)

# Step 3: Decrypt ciphertext (using same keystream)
decrypted_text = decrypt(ciphertext, keystream)

# Display results
print("Plaintext     :", plaintext)
print("Keystream (hex):", keystream.hex())
print("Ciphertext (hex):", ciphertext.hex())
print("Decrypted Text :", decrypted_text)
```

```
Plaintext     : b'HELLO'
Keystream (hex): e67947e5cf
Ciphertext (hex): ae3c0ba980
Decrypted Text : b'HELLO'
```

4. Yes, it can do that if the same keystream `K` is reused and the attacker knows (or can guess) `P1`, they can compute `K = P1 ⊕ C1` and then recover any other message encrypted with the same `K` by `P2 = C2 ⊕ K`.

[21]:
```python
# Demonstration of known-plaintext attack when the same keystream is reused
import os

# Two plaintexts of equal length (required for this simple demo)
P1 = b'HELLO123'   # attacker knows or guesses this
P2 = b'SECRET42'   # target message attacker wants to recover

# Attacker and sender mistakenly reuse the same keystream K
K = os.urandom(len(P1))    # keystream (should NEVER be reused in real systems)

# Sender encrypts both messages with same keystream
C1 = bytes([p ^ k for p, k in zip(P1, K)])
C2 = bytes([p ^ k for p, k in zip(P2, K)])

# --- Attacker side (knows P1 and C1) ---
# Recover keystream:
K_recovered = bytes([p ^ c for p, c in zip(P1, C1)])   # K_recovered == K

# Use recovered keystream to decrypt C2:
P2_recovered = bytes([c ^ k for c, k in zip(C2, K_recovered)])

# Print results (binary and ascii)
def bstr(x): return ' '.join(format(b, '08b') for b in x)

print("P1 (ASCII)        :", P1)
print("P1 (binary)       :", bstr(P1))
print("C1 (binary)       :", bstr(C1))
print("Recovered K (bin) :", bstr(K_recovered))
print()
print("C2 (binary)       :", bstr(C2))
print("P2 recovered (ASCII):", P2_recovered)
print("P2 recovered (binary):", bstr(P2_recovered))

# Verify correctness
print("\nKeystream match?   ", K_recovered == K)
print("P2 match?          ", P2_recovered == P2)
```

```
P1 (ASCII)        : b'HELLO123'
P1 (binary)       : 01001000 01000101 01001100 01001100 01001111 00110001 00110010 00110011
C1 (binary)       : 11010010 00001111 01100000 01111101 01000000 11100010 00011000 10010010
Recovered K (bin) : 10011010 01001010 00101100 00110001 00001111 11010011 00101010 10100001

C2 (binary)       : 11001001 00001111 01101111 01100011 01001010 10000111 00011110 10010011
P2 recovered (ASCII): b'SECRET42'
P2 recovered (binary): 01010011 01000101 01000011 01010010 01000101 01010100 00110100 00110010

Keystream match?    True
P2 match?           True
```

[ ]: