PONDICHERRY UNIVERSITY

SCHOOL OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CSNS 616: CRYPTOGRAPHY LAB

NAME : **SANTANU MONDAL**

REGISTER NO. : **24MTNISPY0004**

COURSE : **M.Tech. NIS**

SEMESTER : **1st**

PONDICHERRY UNIVERSITY

SCHOOL OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

BONAFIDE CERTIFICATE

This is to certify that this is a Bonafide record of practical work done for **CSNS 616 Cryptography Lab** by **SANTANU MONDAL** bearing Register Number **24MTNISPY0004** of M.Tech (Network & Information Security) in Semester I during the year 2024-2025.

Faculty In-Charge

Submitted for the practical examination held on _____

Internal Examiner **External Examiner**

INDEX

SOURCE CODE:

//TCPServer:

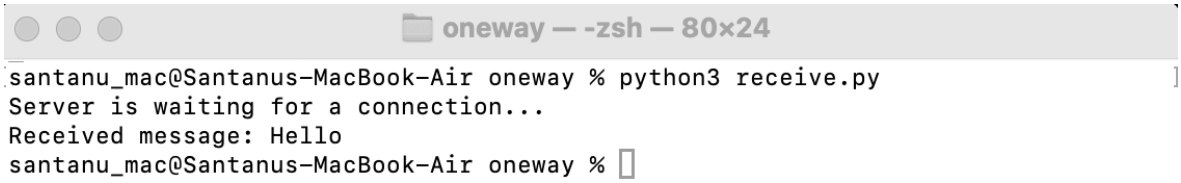
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('localhost', 5001))
s.listen(1)
print("Server is waiting for a connection...")
client, address = s.accept()
data = client.recv(1024).decode()
print(f"Received message: {data}")
response = "Received"
client.sendall(response.encode())
s.close()
```

//TCPClient:

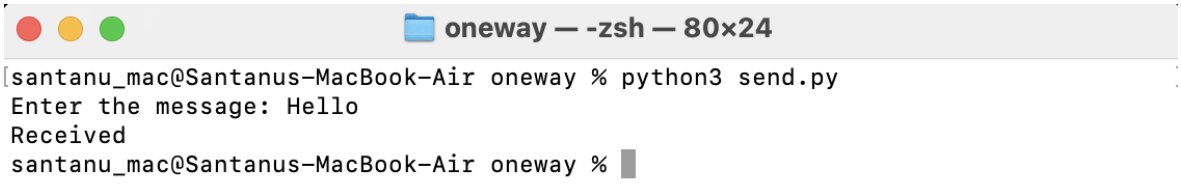
```
import socket
message = input("Enter the message: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 5001))
s.sendall(message.encode())
response = s.recv(1024).decode()
print(response)
s.close()
```

SAMPLE INPUT AND OUTPUT:

//TCPServer



//TCPClient



SOURCE CODE:

//TCPServer (Two Way):

```
import socket
import threading
def send():
    while True:
        message = input("Enter the message: ")
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('localhost', 5001))
        s.sendall(message.encode())
        response = s.recv(1024).decode()
        print(f"\n{response}")
        s.close()
def receive():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('localhost', 5002))
    s.listen(1)
    print("Server is waiting for a connection...")
    while True:
        client, address = s.accept()
        data = client.recv(1024).decode()
        print(f"\nReceived message: {data}")
        response = "Received"
        client.sendall(response.encode())
        client.close()
if __name__ == "__main__":
    threading.Thread(target=send).start()
    threading.Thread(target=receive).start()
```

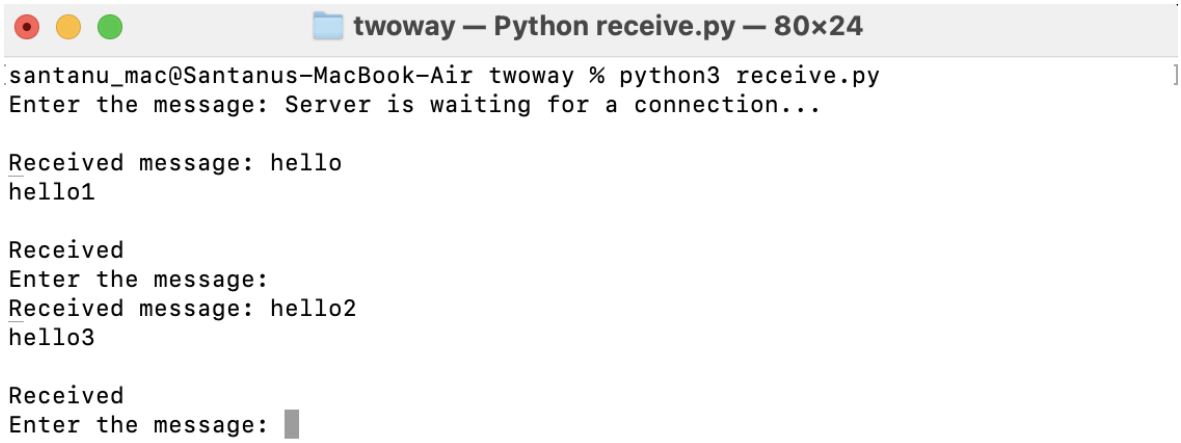
//TCPClient (Two Way):

```
import socket
import threading
def send():
    while True:
        message = input("Enter the message: ")
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('localhost', 5002))
        s.sendall(message.encode())
        response = s.recv(1024).decode()
        print(f"\n{response}")
        s.close()
def receive():
```

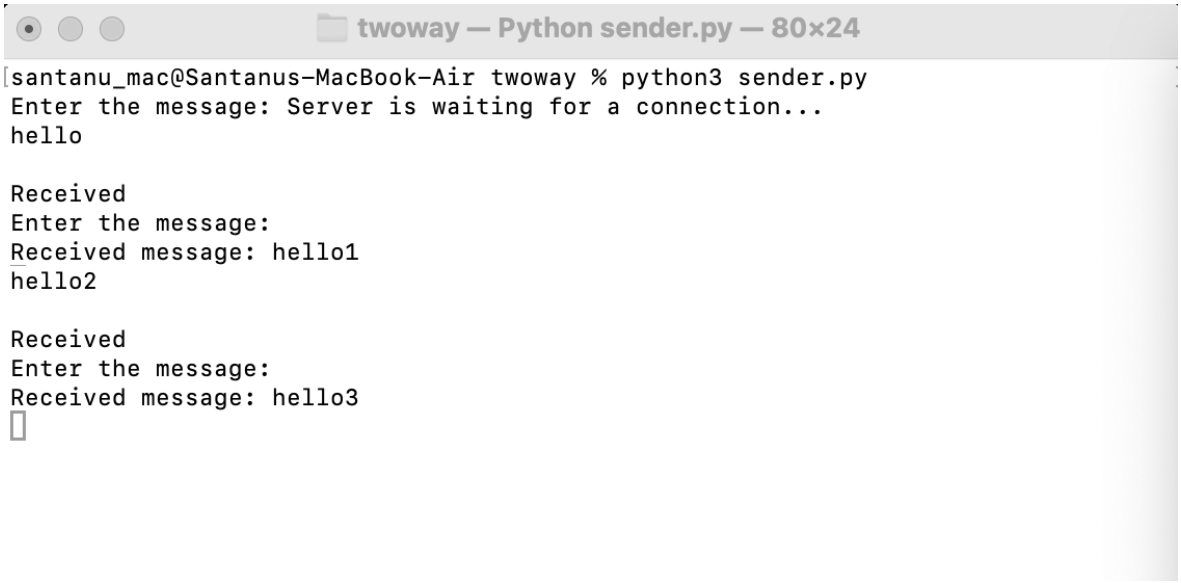
```
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('localhost', 5001))
    s.listen(1)
    print("Server is waiting for a connection...")
    while True:
        client, address = s.accept()
        data = client.recv(1024).decode()
        print(f"\nReceived message: {data}")
        response = "Received"
        client.sendall(response.encode())
        client.close()
if __name__ == "__main__":
    threading.Thread(target=send).start()
    threading.Thread(target=receive).start()
```

SAMPLE INPUT AND OUTPUT:

// TCPServer



//TCPClient



SOURCE CODE:

//Multicast Sender

```
import socket
import struct
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)

while True:
    group = input("Send message to Group 1 or Group 2: ")
    message = input("Enter your message: ").encode()

    if group == '1':
        sock.sendto(message, ('224.1.1.1', 5005))
    elif group == '2':
        sock.sendto(message, ('224.1.1.2', 5006))
    else:
        print("Invalid group selection!")
```

//Multicast Receiver Group 1:

```
import socket
import struct
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sock.bind(('0.0.0.0', 5005))

mreq = struct.pack("4sl", socket.inet_aton('224.1.1.1'), socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

while True:
    data, addr = sock.recvfrom(1024)
    print(f"Received message: {data.decode()}")
```

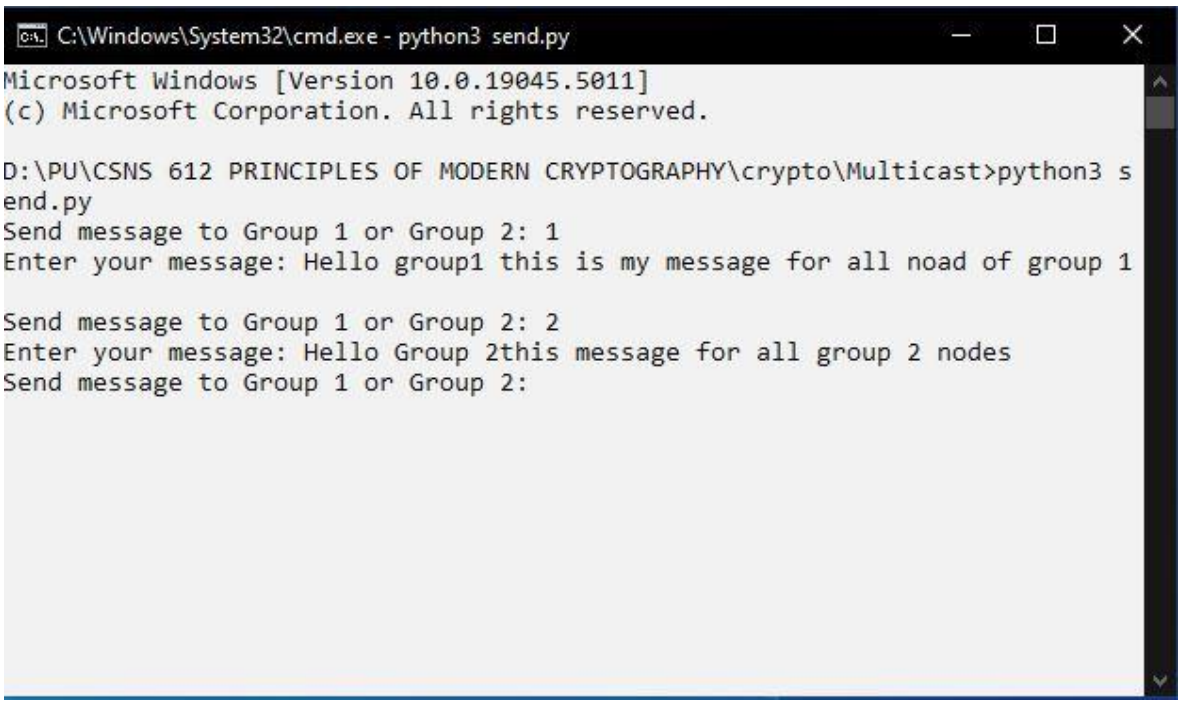
//Multicast Receiver Group 2:

```
import socket
import struct
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sock.bind(('0.0.0.0', 5006))
```

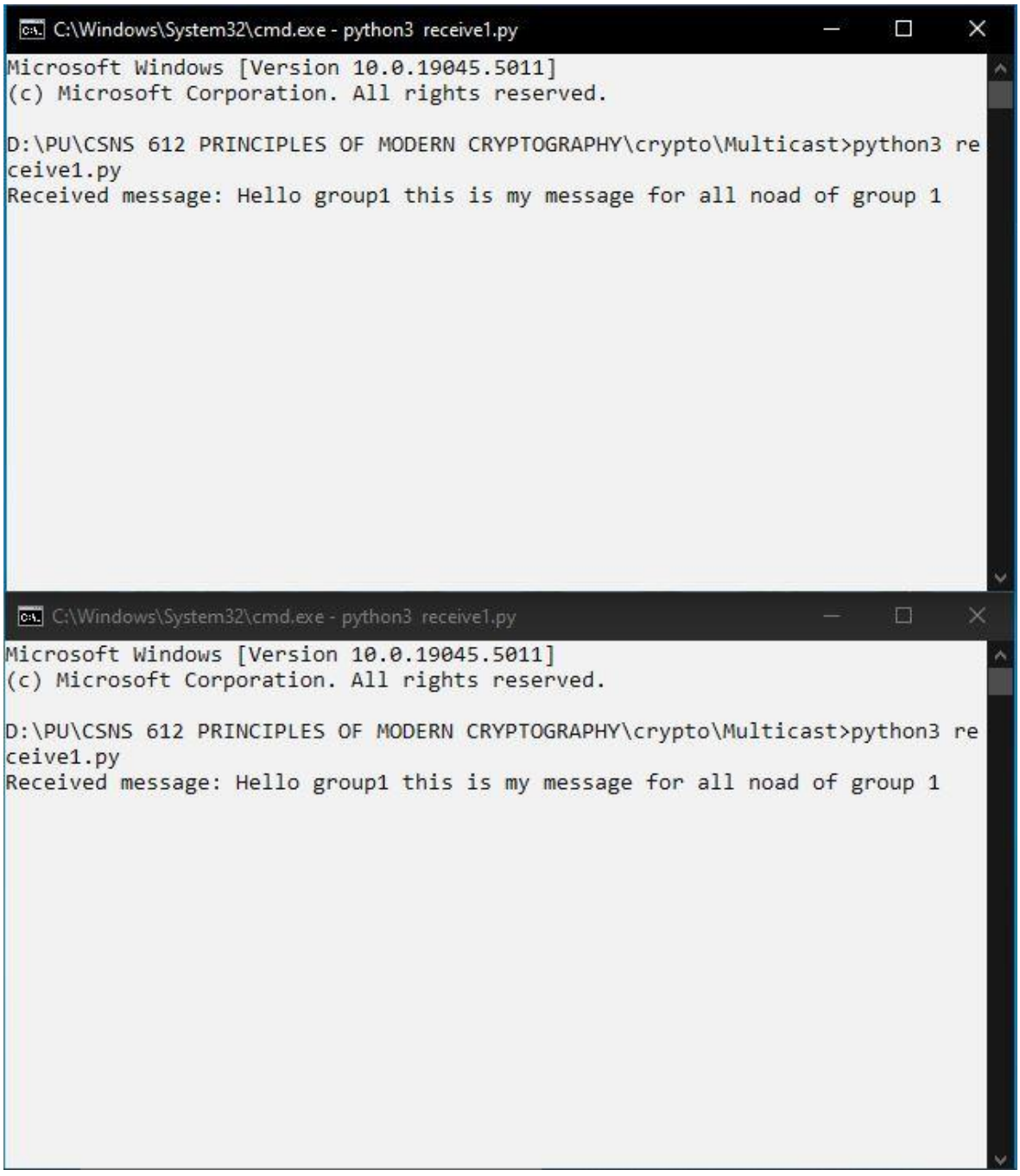
```
mreq = struct.pack("4sl", socket.inet_aton('224.1.1.2'), socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

while True:
    data, addr = sock.recvfrom(1024)
    print(f"Received message: {data.decode()}")
```

SAMPLE INPUT AND OUTPUT:

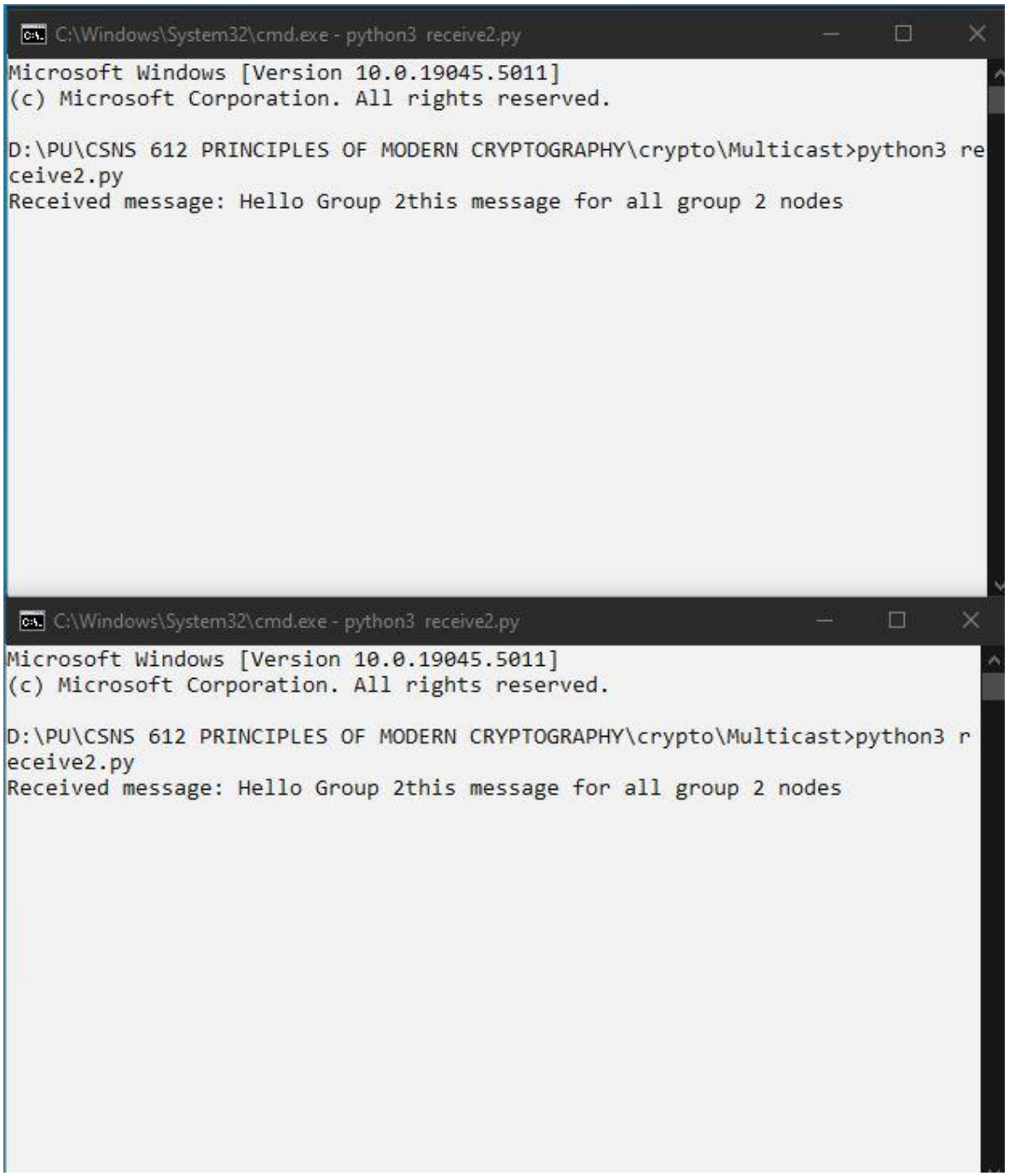
//Multicast Sender



//Multicast Receiver Group 1:



//Multicast Receiver Group 2:

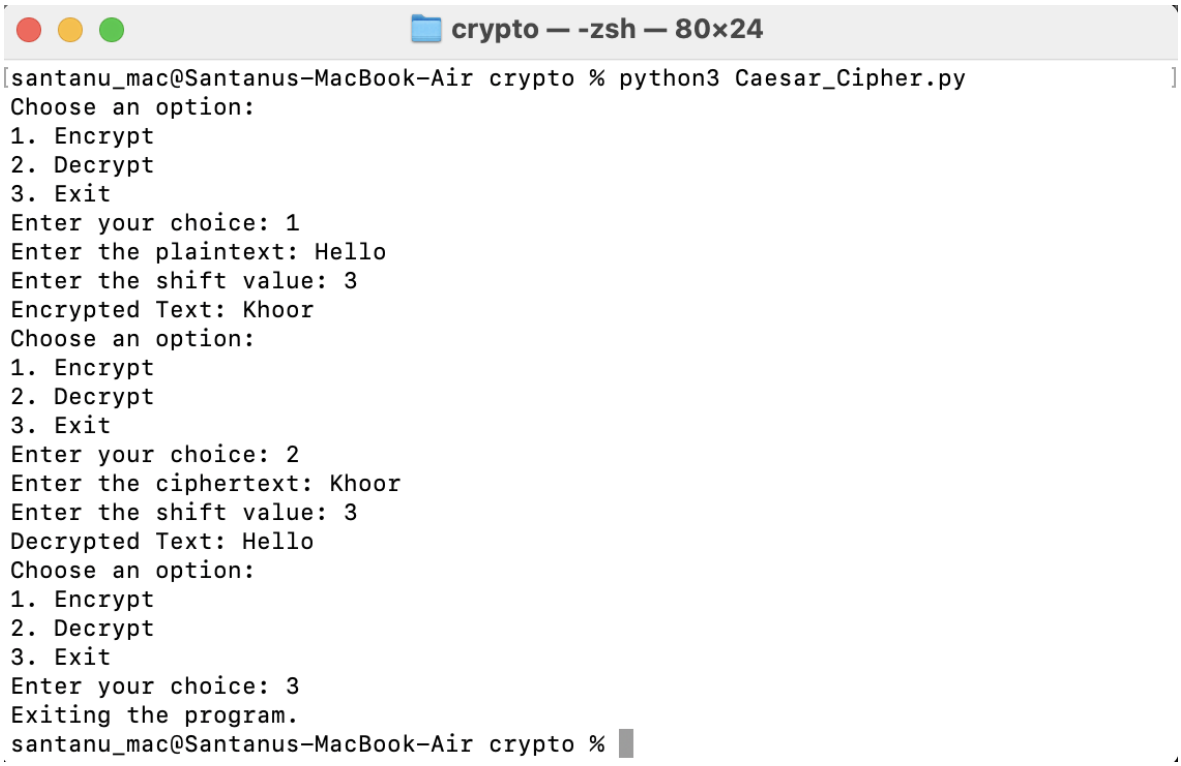


SOURCE CODE:

//Caesar Cipher

```
def encrypt(plaintext, shift):
    ciphertext = ""
    for ch in plaintext:
        if ch.isalpha():
            base = ord('a') if ch.islower() else ord('A')
            ch = chr((ord(ch) - base + shift) % 26 + base)
        ciphertext += ch

    return ciphertext

def decrypt(ciphertext, shift):
    return encrypt(ciphertext, 26 - shift)
def main():
    while True:
        print("Choose an option:")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plaintext = input("Enter the plaintext: ")
            shift = int(input("Enter the shift value: "))
            encrypted_text = encrypt(plaintext, shift)
            print("Encrypted Text:", encrypted_text)

        elif choice == '2':
            ciphertext = input("Enter the ciphertext: ")
            shift = int(input("Enter the shift value: "))
            decrypted_text = decrypt(ciphertext, shift)
            print("Decrypted Text:", decrypted_text)

        elif choice == '3':
            print("Exiting the program.")
            break

        else:
            print("Invalid choice! Please enter 1, 2, or 3.")

if __name__ == "__main__":
    main()
```

SAMPLE INPUT AND OUTPUT:



SOURCE CODE:

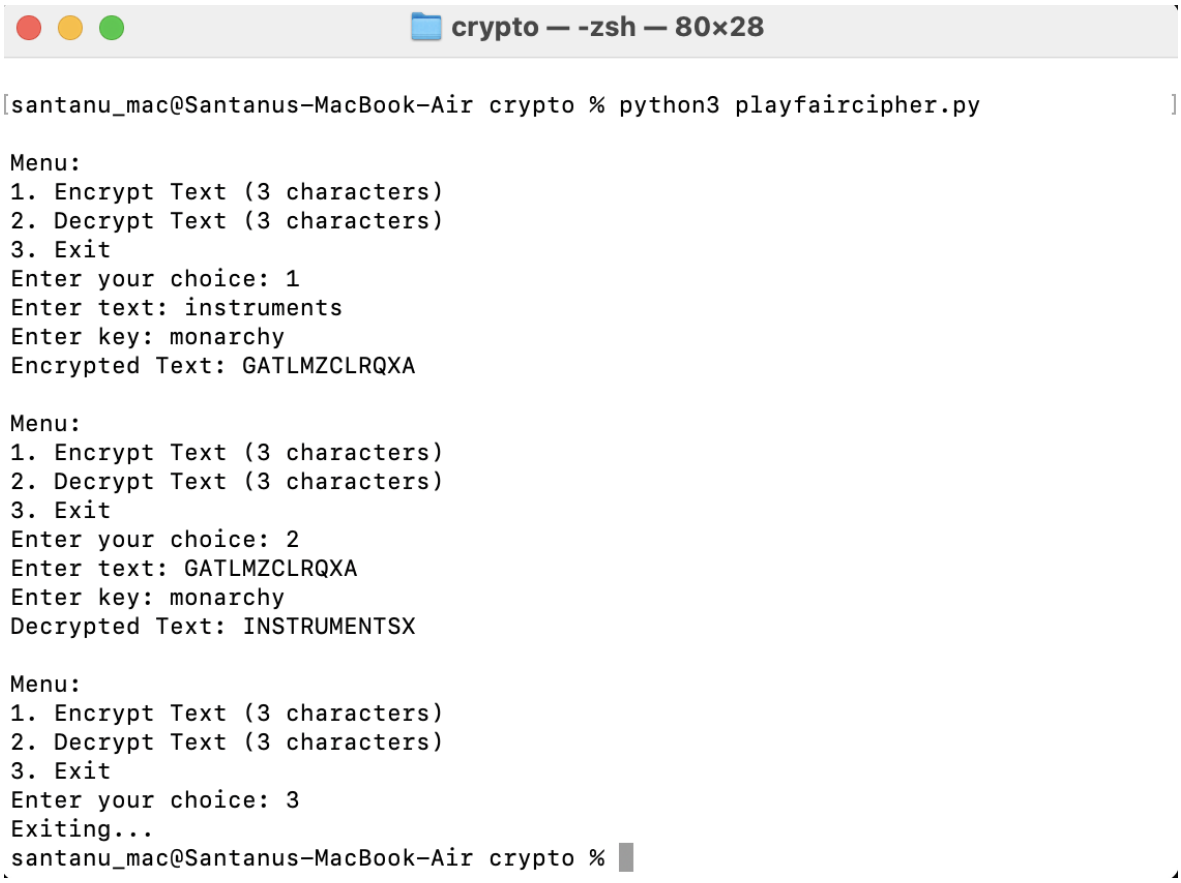
//Playfair Cipher

```
import re

def generate_key_matrix(key):
    matrix = []
    key = key.replace("J", "I")
    key = "".join(sorted(set(key), key=lambda x: key.index(x)))
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    key_matrix = key + "".join([char for char in alphabet if char not in key])
    for i in range(0, 25, 5):
        matrix.append(list(key_matrix[i:i+5]))
    return matrix

def find_position(matrix, char):
    for row in range(5):
        for col in range(5):
            if matrix[row][col] == char:
                return row, col
    return None

def prepare_text(text):
    text = re.sub(r'[^A-Z]', '', text.upper())
    text = text.replace("J", "I")
    prepared_text = ""
    i = 0
    while i < len(text):
        prepared_text += text[i]
        if i+1 < len(text) and text[i] == text[i+1]:
            prepared_text += "X"
        elif i+1 < len(text):
            prepared_text += text[i+1]
        i += 2
    if len(prepared_text) % 2 != 0:
        prepared_text += "X"
    return prepared_text

def encrypt(text, key):
    matrix = generate_key_matrix(key)
    prepared_text = prepare_text(text)
    result = ""
    for i in range(0, len(prepared_text), 2):
        char1, char2 = prepared_text[i], prepared_text[i+1]
```

```
        row1, col1 = find_position(matrix, char1)
        row2, col2 = find_position(matrix, char2)
        if row1 == row2:
            result += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:
            result += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
        else:
            result += matrix[row1][col2] + matrix[row2][col1]
    return result

def decrypt(text, key):
    matrix = generate_key_matrix(key)
    prepared_text = prepare_text(text)
    result = ""
    for i in range(0, len(prepared_text), 2):
        char1, char2 = prepared_text[i], prepared_text[i+1]
        row1, col1 = find_position(matrix, char1)
        row2, col2 = find_position(matrix, char2)
        if row1 == row2:
            result += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:
            result += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
        else:
            result += matrix[row1][col2] + matrix[row2][col1]
    return result

def main():
    while True:

        print("\nMenu:")
        print("1. Encrypt Text (3 characters)")
        print("2. Decrypt Text (3 characters)")
        print("3. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:
            text = input("Enter text: ").strip().upper()
            key = input("Enter key: ").strip().upper()
            print("Encrypted Text:", encrypt(text, key))

        elif choice == 2:
            text = input("Enter text: ").strip().upper()
            key = input("Enter key: ").strip().upper()
            print("Decrypted Text:", decrypt(text, key))
```

```
        elif choice == 3:
            print("Exiting...")
            break

        else:
            print("Invalid choice, please try again!")

if __name__ == "__main__":
    main()
```

SAMPLE INPUT AND OUTPUT:

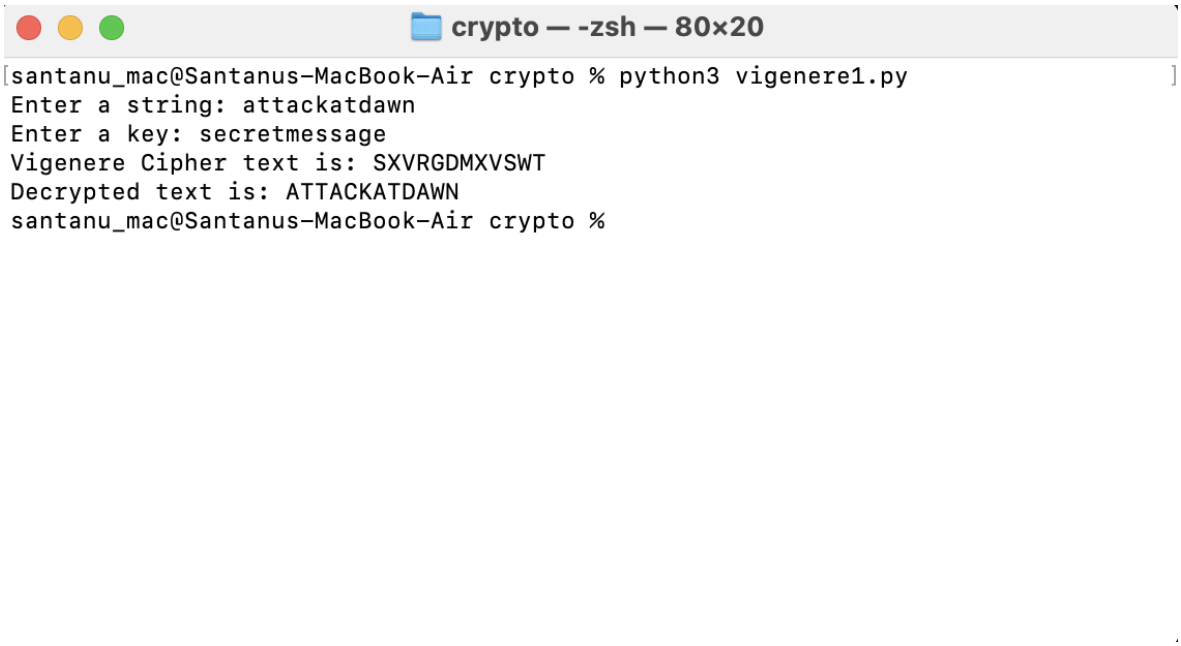


SOURCE CODE:

//Vigenere Cipher

```
def vigenere_cipher(text, key):
    text = text.replace(" ", "").upper()
    key = key.replace(" ", "").upper()
    encrypted = []
    decrypted = []

    for i, char in enumerate(text):
        text_num = ord(char) - ord('A')
        key_num = ord(key[i % len(key)]) - ord('A')

        encrypted_char = chr((text_num + key_num) % 26 + ord('A'))
        encrypted.append(encrypted_char)

        decrypted_char = chr((ord(encrypted_char) - ord('A') - key_num + 26) % 26 + ord('A'))
        decrypted.append(decrypted_char)

    return ''.join(encrypted), ''.join(decrypted)

text = input("Enter a string: ")
key = input("Enter a key: ")

cipher_text, decrypted_text = vigenere_cipher(text, key)
print("Vigenere Cipher text is:", cipher_text)
print("Decrypted text is:", decrypted_text)
```

SAMPLE INPUT AND OUTPUT:



SOURCE CODE:

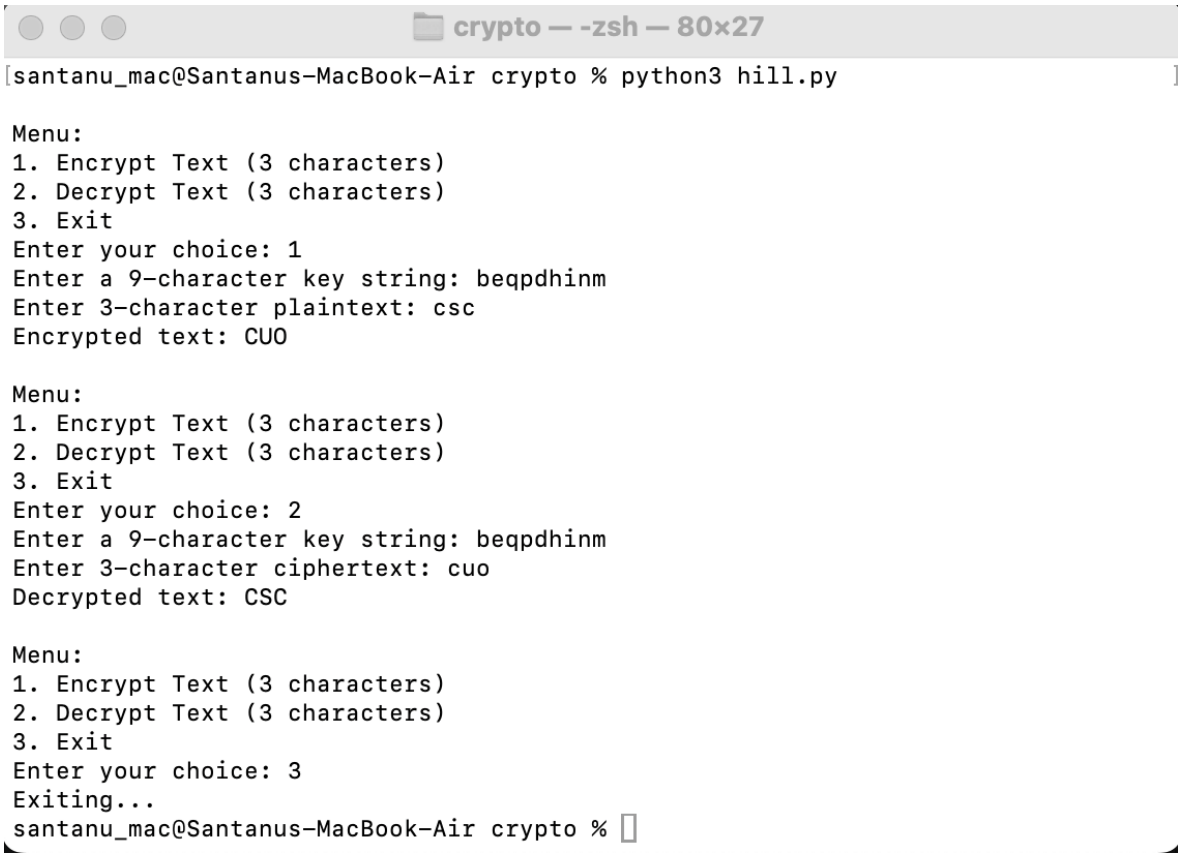
//Hill Cipher:

```
SIZE = 3

def letter_to_number(letter):
    return ord(letter.upper()) - ord('A')

def number_to_letter(number):
    return chr(number + ord('A'))

def mod26(a):
    return (a % 26 + 26) % 26

def encrypt(plaintext, key):
    plaintext_vector = [letter_to_number(plaintext[j]) for j in range(SIZE)]
    cipher_vector = [0] * SIZE

    for j in range(SIZE):
        for k in range(SIZE):
            cipher_vector[j] += key[j][k] * plaintext_vector[k]
        cipher_vector[j] = mod26(cipher_vector[j])

    ciphertext = ''.join(number_to_letter(cipher_vector[j]) for j in range(SIZE))
    return ciphertext

def string_to_key_matrix(key_string):
    key = [[0] * SIZE for _ in range(SIZE)]
    idx = 0
    for i in range(SIZE):
        for j in range(SIZE):
            key[i][j] = letter_to_number(key_string[idx])
            idx += 1
    return key

def find_determinant(matrix):
    det = (matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1]) -
           matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0]) +
           matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0]))
    return mod26(det)

def mod_inverse(a, m):
    a = a % m
    for x in range(1, m):
```

```
        if (a * x) % m == 1:
            return x
    return -1

def find_cofactor_matrix(matrix):
    cofactor = [[0] * SIZE for _ in range(SIZE)]
    cofactor[0][0] = mod26(matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1])
    cofactor[0][1] = mod26(-(matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0]))
    cofactor[0][2] = mod26(matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0])
    cofactor[1][0] = mod26(-(matrix[0][1] * matrix[2][2] - matrix[0][2] * matrix[2][1]))
    cofactor[1][1] = mod26(matrix[0][0] * matrix[2][2] - matrix[0][2] * matrix[2][0])
    cofactor[1][2] = mod26(-(matrix[0][0] * matrix[2][1] - matrix[0][1] * matrix[2][0]))
    cofactor[2][0] = mod26(matrix[0][1] * matrix[1][2] - matrix[0][2] * matrix[1][1])
    cofactor[2][1] = mod26(-(matrix[0][0] * matrix[1][2] - matrix[0][2] * matrix[1][0]))
    cofactor[2][2] = mod26(matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0])
    return cofactor

def transpose_matrix(matrix):
    return [[matrix[j][i] for j in range(SIZE)] for i in range(SIZE)]

def find_inverse_matrix(key):
    det = find_determinant(key)
    det_inv = mod_inverse(det, 26)
    if det_inv == -1:
        return None

    cofactor = find_cofactor_matrix(key)
    adjugate = transpose_matrix(cofactor)

    inverse = [[mod26(adjugate[i][j] * det_inv) for j in range(SIZE)] for i in range(SIZE)]
    return inverse

def decrypt(ciphertext, inverse_key):
    return encrypt(ciphertext, inverse_key)

def main():
    while True:
        print("\nMenu:")
        print("1. Encrypt Text (3 characters)")
        print("2. Decrypt Text (3 characters)")
        print("3. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            key_string = input("Enter a 9-character key string: ")
```

```
            key = string_to_key_matrix(key_string)
            while True:
                text = input("Enter 3-character plaintext: ")
                if len(text) == SIZE:
                    break
                print("Error: Please enter exactly 3 characters.")
            result = encrypt(text, key)
            print("Encrypted text:", result)

        elif choice == 2:
            key_string = input("Enter a 9-character key string: ")
            key = string_to_key_matrix(key_string)
            while True:
                text = input("Enter 3-character ciphertext: ")
                if len(text) == SIZE:
                    break
                print("Error: Please enter exactly 3 characters.")

            inverse_key = find_inverse_matrix(key)
            if inverse_key is None:
                print("Inverse of the key matrix does not exist!")
                continue

            result = decrypt(text, inverse_key)
            print("Decrypted text:", result)

        elif choice == 3:
            print("Exiting...")
            break

        else:
            print("Invalid choice, please try again!")

if __name__ == "__main__":
    main()
```

SAMPLE INPUT AND OUTPUT:



SOURCE CODE:

//DES

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = get_random_bytes(8)

def des_encrypt(plaintext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    padded_text = pad(plaintext, DES.block_size)
    encrypted_text = cipher.encrypt(padded_text)
    return encrypted_text

def des_decrypt(ciphertext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_text = cipher.decrypt(ciphertext)
    unpadded_text = unpad(decrypted_text, DES.block_size)
    return unpadded_text

while True:
    print("1. Encrypt")
    print("2. Decrypt")
    print("3. Exit")

    choice = int(input("Choose an option (1-3): "))

    if choice == 1:
        plaintext = input("Enter the text to encrypt: ").encode('utf-8')
        encrypted_text = des_encrypt(plaintext, key)
        print("Encrypted text (hex):", encrypted_text.hex())

    elif choice == 2:
        ciphertext_hex = input("Enter the hex string to decrypt: ")
        ciphertext = bytes.fromhex(ciphertext_hex)
        decrypted_text = des_decrypt(ciphertext, key)
        print("Decrypted text:", decrypted_text.decode('utf-8'))

    elif choice == 3:
        exit()

    else:
        print("Invalid option, please try again.")
```

SAMPLE INPUT AND OUTPUT:

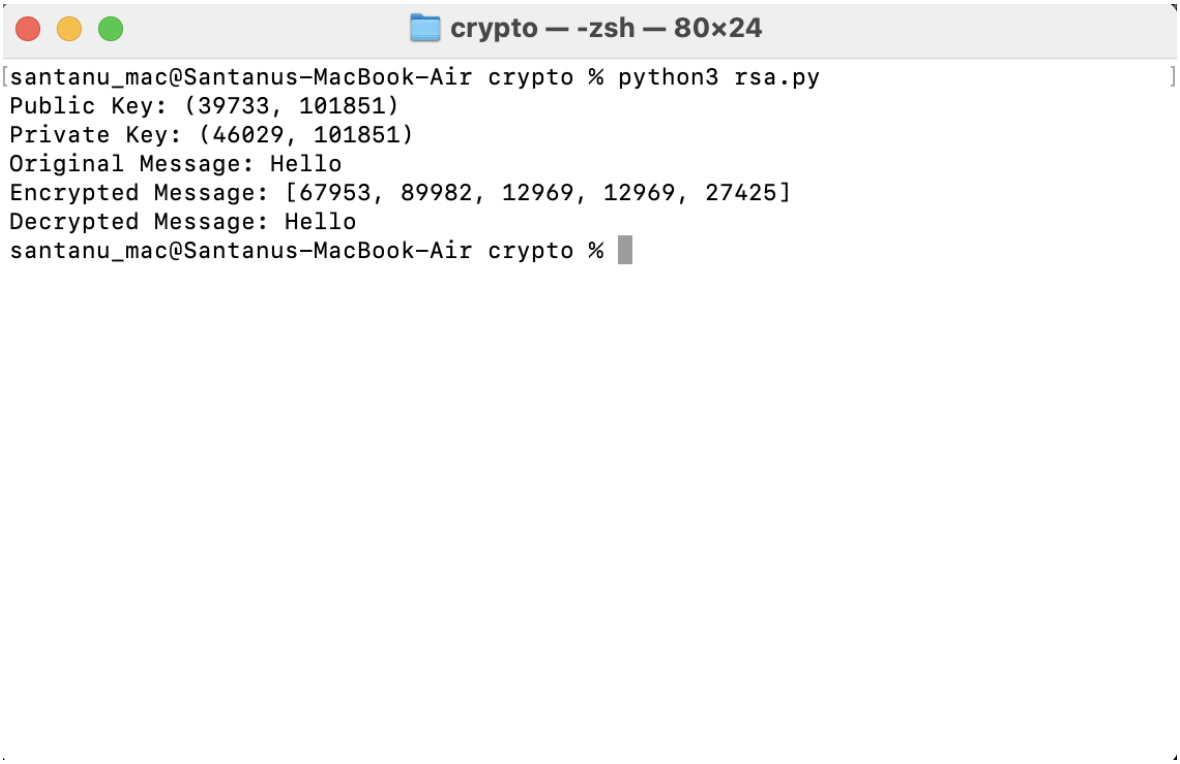
SOURCE CODE:

//RSA

```
import random
from sympy import mod_inverse, isprime

def generate_large_prime():
    while True:
        num = random.randint(100, 1000)
        if isprime(num):
            return num

def generate_keypair():
    # Step 1: Choose two prime numbers
    p = generate_large_prime()
    q = generate_large_prime()
    while p == q:
        q = generate_large_prime()

    # Step 2: Compute n = p * q
    n = p * q

    # Step 3: Compute phi(n) = (p-1)*(q-1)
    phi = (p - 1) * (q - 1)

    # Step 4: Choose e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1
    e = random.randint(2, phi - 1)
    while not isprime(e):  # In practice, we check if gcd(e, phi(n)) = 1
        e = random.randint(2, phi - 1)

    # Step 5: Compute d, the modular inverse of e mod phi(n)
    d = mod_inverse(e, phi)

    return ((e, n), (d, n))

def encrypt(plaintext, public_key):
    e, n = public_key
    encrypted_message = [pow(ord(char), e, n) for char in plaintext]
    return encrypted_message

def decrypt(ciphertext, private_key):
    d, n = private_key
    decrypted_message = ''.join([chr(pow(char, d, n)) for char in ciphertext])
    return decrypted_message
```

```
public_key, private_key = generate_keypair()
print("Public Key:", public_key)
print("Private Key:", private_key)

message = "Hello"
print("Original Message:", message)

encrypted_msg = encrypt(message, public_key)
print("Encrypted Message:", encrypted_msg)

decrypted_msg = decrypt(encrypted_msg, private_key)
print("Decrypted Message:", decrypted_msg)
```

SAMPLE INPUT AND OUTPUT:

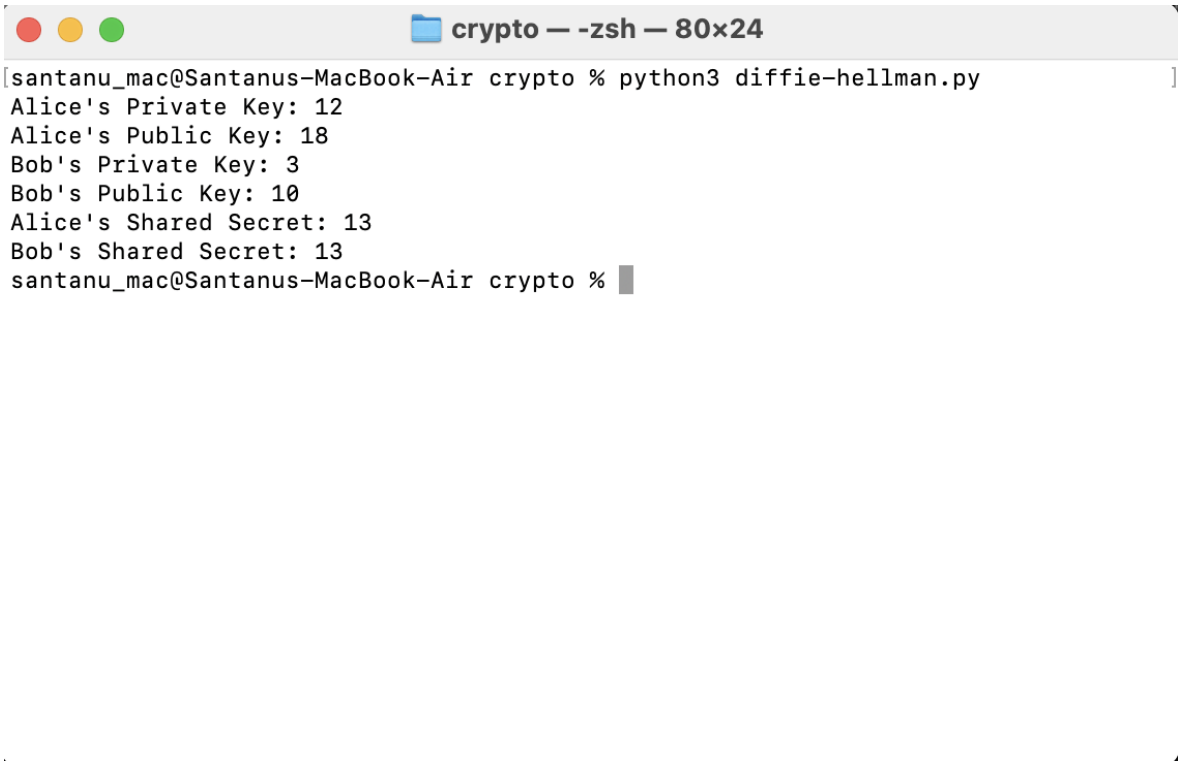


SOURCE CODE:

//Diffiee-Hellman

```
import random

def generate_private_key(prime):
    return random.randint(2, prime - 2)

def generate_public_key(private_key, prime, generator):
    return pow(generator, private_key, prime)

def compute_shared_secret(public_key, private_key, prime):
    return pow(public_key, private_key, prime)

prime = 23
generator = 5

# Alice's keys
alice_private_key = generate_private_key(prime)
alice_public_key = generate_public_key(alice_private_key, prime, generator)

# Bob's keys
bob_private_key = generate_private_key(prime)
bob_public_key = generate_public_key(bob_private_key, prime, generator)

# Exchange public keys and compute shared secrets
alice_shared_secret = compute_shared_secret(bob_public_key, alice_private_key, prime)
bob_shared_secret = compute_shared_secret(alice_public_key, bob_private_key, prime)

print("Alice's Private Key:", alice_private_key)
print("Alice's Public Key:", alice_public_key)
print("Bob's Private Key:", bob_private_key)
print("Bob's Public Key:", bob_public_key)
print("Alice's Shared Secret:", alice_shared_secret)
print("Bob's Shared Secret:", bob_shared_secret)
```

SAMPLE INPUT AND OUTPUT:

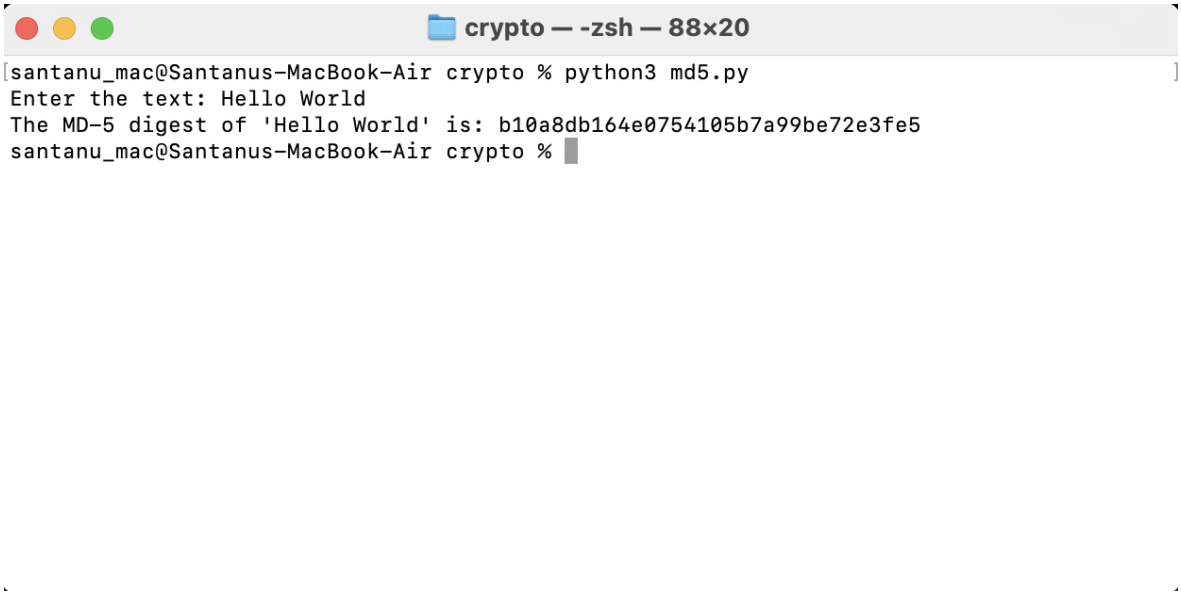


SOURCE CODE:

//MD5

```
import hashlib

text = input("Enter the text: ")
md5 = hashlib.md5()
md5.update(text.encode('utf-8'))
digest = md5.hexdigest()
print(f"The MD-5 digest of '{text}' is: {digest}")
```

SAMPLE INPUT AND OUTPUT:

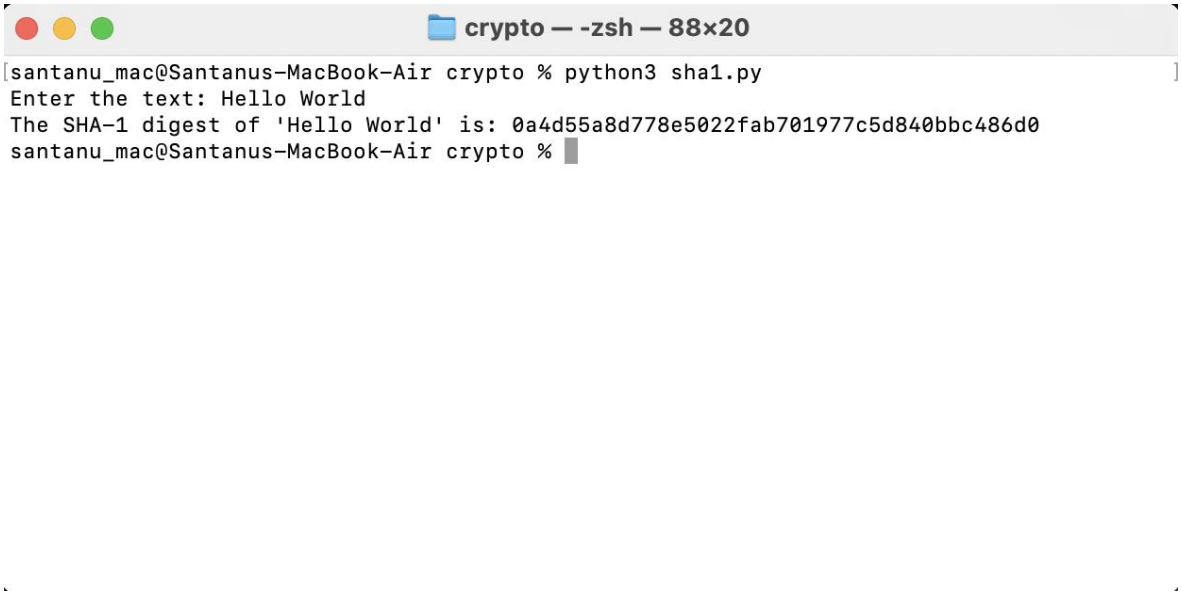


SOURCE CODE:

//SHA-1

```
import hashlib

text = input("Enter the text: ")
sha1 = hashlib.sha1()
sha1.update(text.encode('utf-8'))
digest = sha1.hexdigest()
print(f"The SHA-1 digest of '{text}' is: {digest}")
```

SAMPLE INPUT AND OUTPUT:



SOURCE CODE:

//Digital Signature

```
import hashlib
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa

def generate_keys():
    private_key = dsa.generate_private_key(key_size=2048, backend=default_backend())
    public_key = private_key.public_key()
    return private_key, public_key

def sign_message(private_key, message):
    message_hash = hashlib.sha256(message.encode()).digest()

    signature = private_key.sign(
        message_hash,
        hashes.SHA256()
    )
    return signature

def verify_signature(public_key, message, signature):
    message_hash = hashlib.sha256(message.encode()).digest()
    try:
        public_key.verify(signature, message_hash, hashes.SHA256())
        return True
    except Exception:
        return False

if __name__ == "__main__":
    private_key, public_key = generate_keys()

    message = input("Enter your message: ")

    signature = sign_message(private_key, message)
    print(f"Signature: {signature.hex()}")

    is_valid = verify_signature(public_key, message, signature)
    print(f"Is the signature valid? {is_valid}")
```

SAMPLE INPUT AND OUTPUT: