

# CSNS 616: CRYPTOGRAPHY LAB

**NAME : SANTANU MONDAL**

**REGISTER NO. : 24MTNISPY0004**

**COURSE : M.Tech. NIS**

**SEMESTER : 1st**



# 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**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EXP.**  **NO.** | **DATE** | **TITLE** | **PAGE**  **NO.** | **SIGNATURE** |
| 1 | 09/09/24 | Transmission Control Protocol using One Way Communication | 01 – 04 |  |
| 2 | 09/09/24 | Transmission Control Protocol using Two Way Communication | 05 – 09 |  |
| 3 | 16/09/24 | Multicast | 10 – 16 |  |
| 4 | 16/09/24 | Caesar Cipher | 17 – 19 |  |
| 5 | 23/09/24 | Playfair Cipher | 20 – 24 |  |
| 6 | 14/10/24 | Hill Cipher | 25 – 30 |  |
| 7 | 07/10/24 | Vigenere Cipher | 31 – 33 |  |
| 8 | 21/10/24 | DES | 34 – 36 |  |
| 9 | 21/10/24 | RSA Algorithm | 37 – 41 |  |
| 10 | 21/10/24 | Diffiee-Hellman | 42 – 44 |  |
| 11 | 21/10/24 | MD5 | 45 – 46 |  |
| 12 | 21/10/24 | SHA-1 | 47 – 48 |  |
| 13 | 28/10/24 | Digital Signature Standard | 49 – 52 |  |

**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()

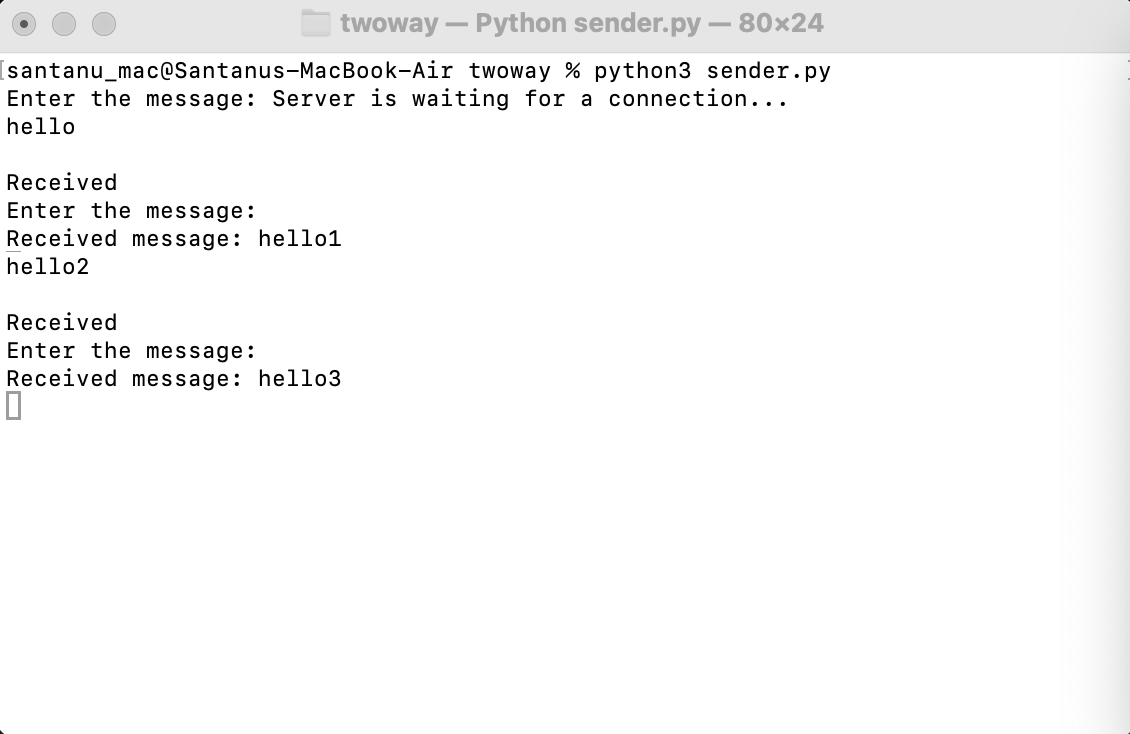
    threading.Thread(target=receive).start()

**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)

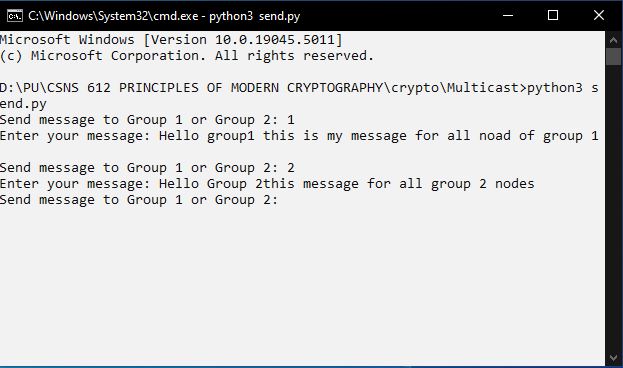
while True:

    data, addr = sock.recvfrom(1024)

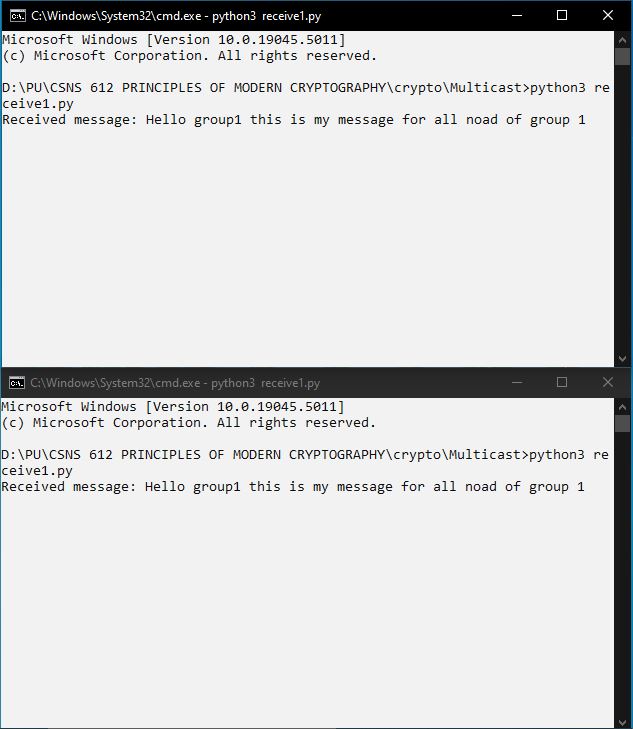
    print(f"Received message: {data.decode()}")

**SAMPLE INPUT AND OUTPUT:**

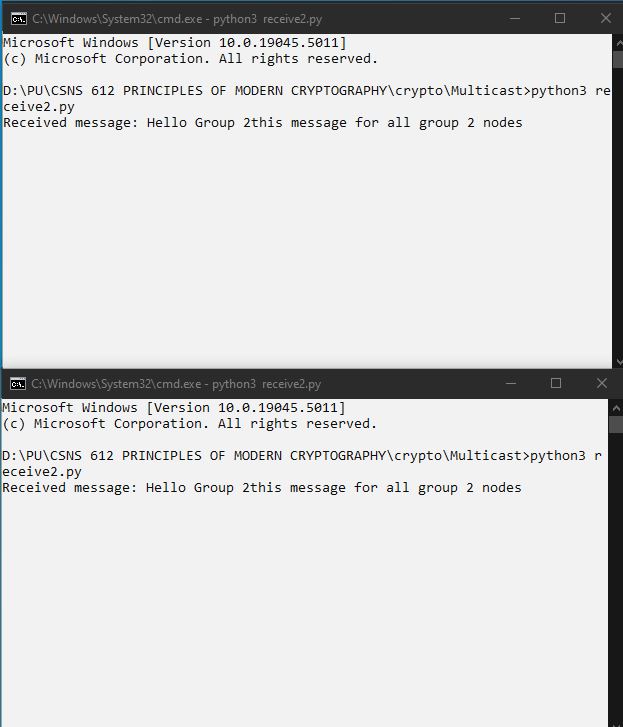
//Multicast Sender



//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

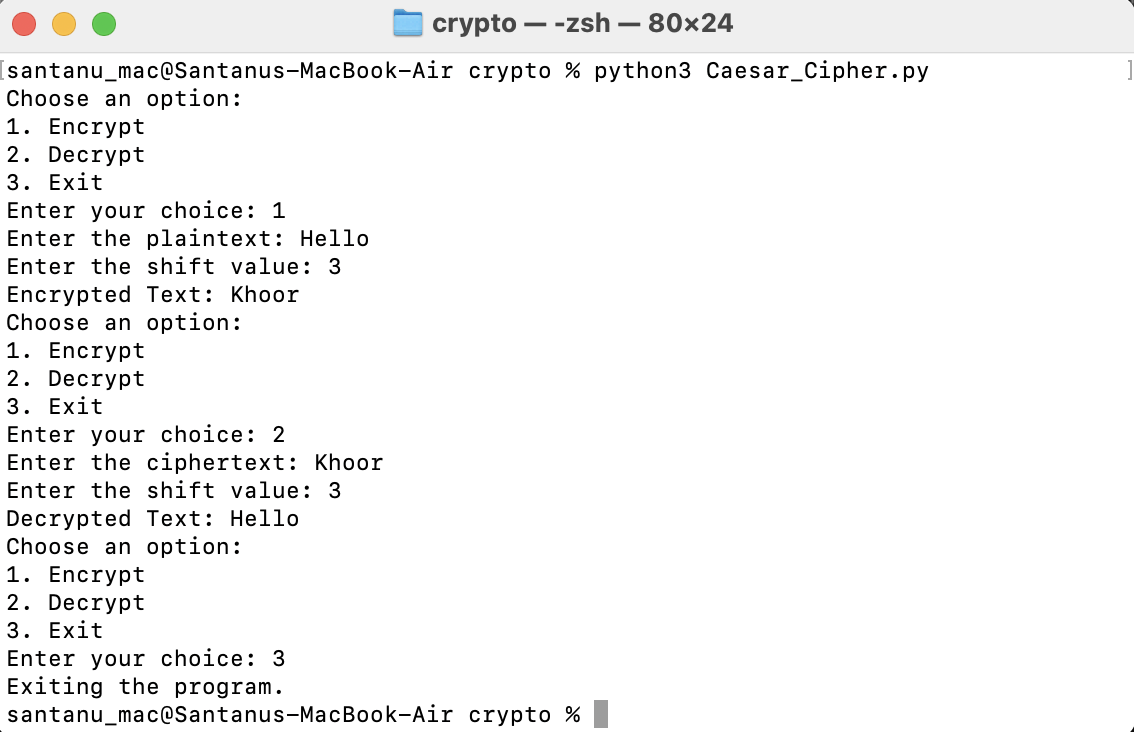
        else:

            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

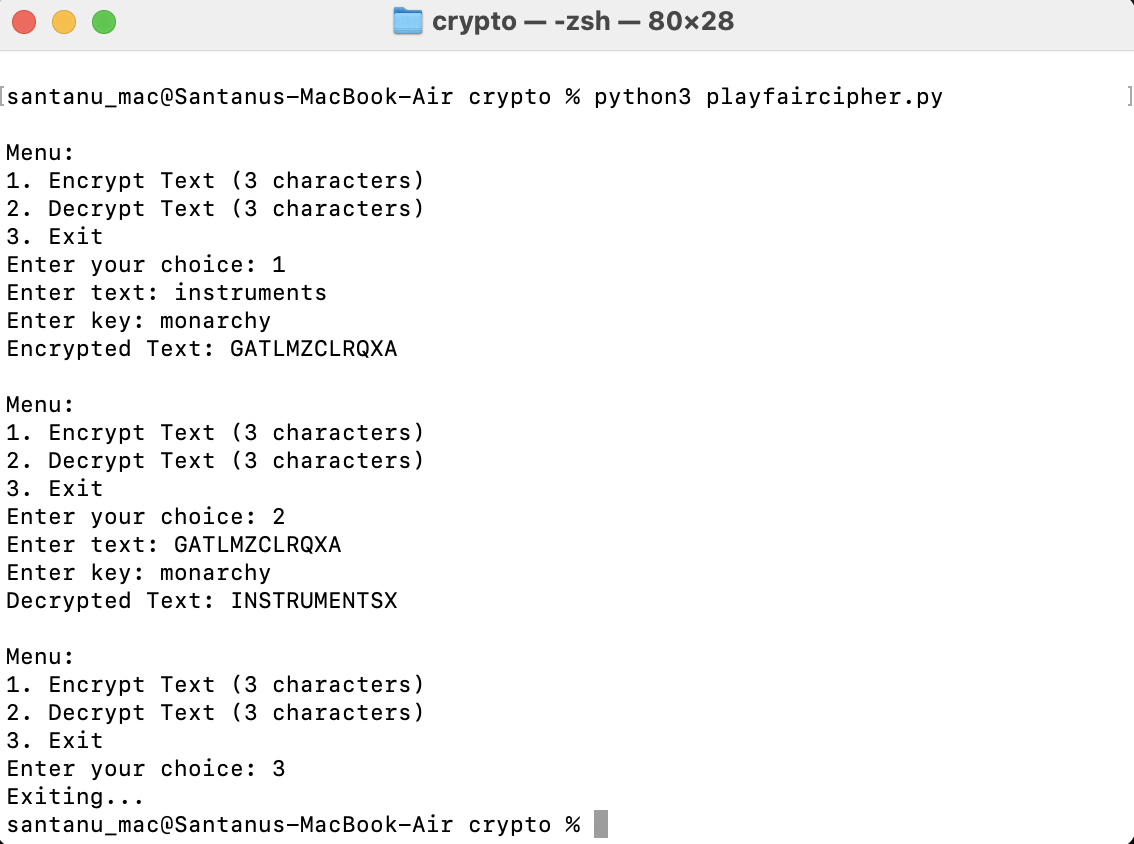
        else:

            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: ")

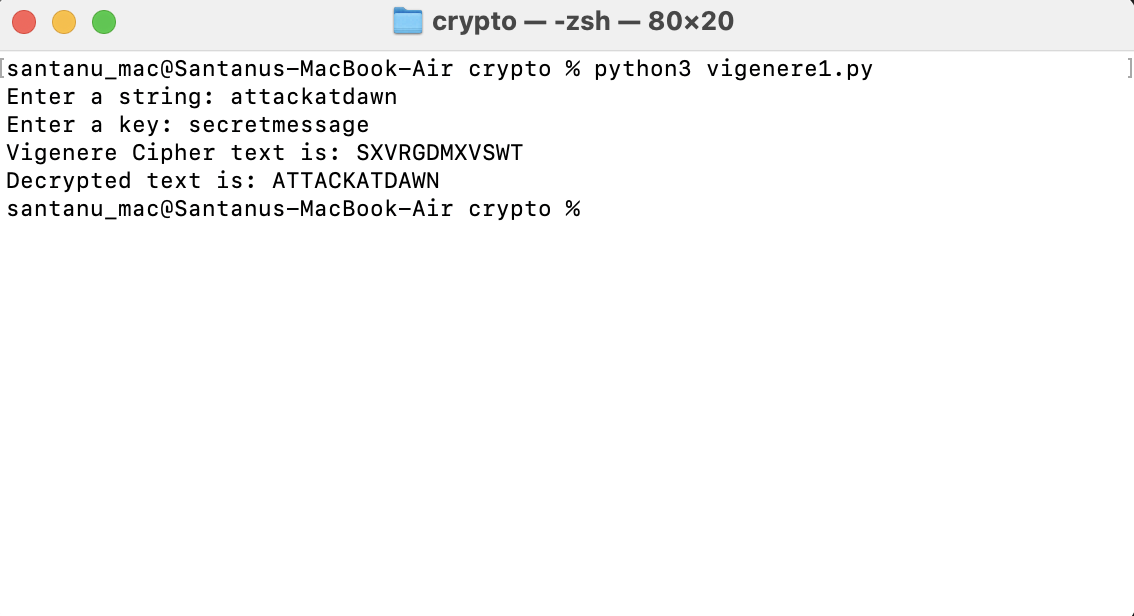
key = input("Enter a key: ")

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

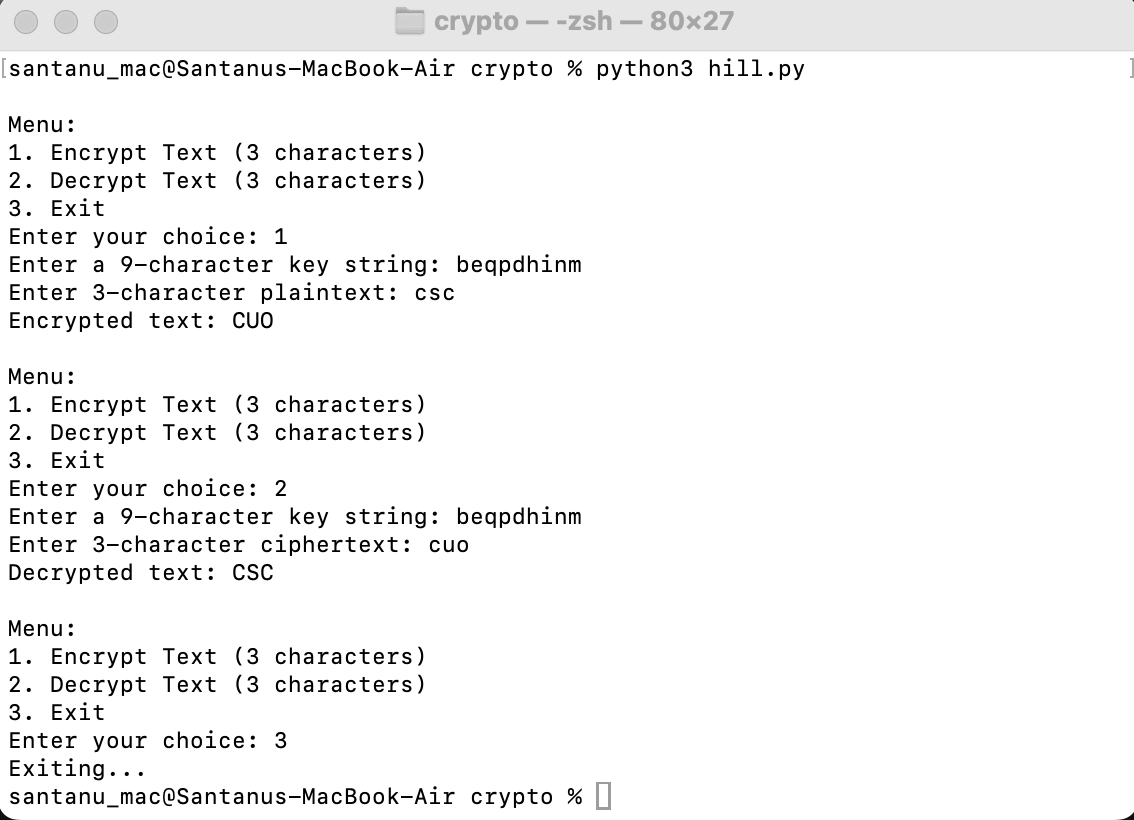
        else:

            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)

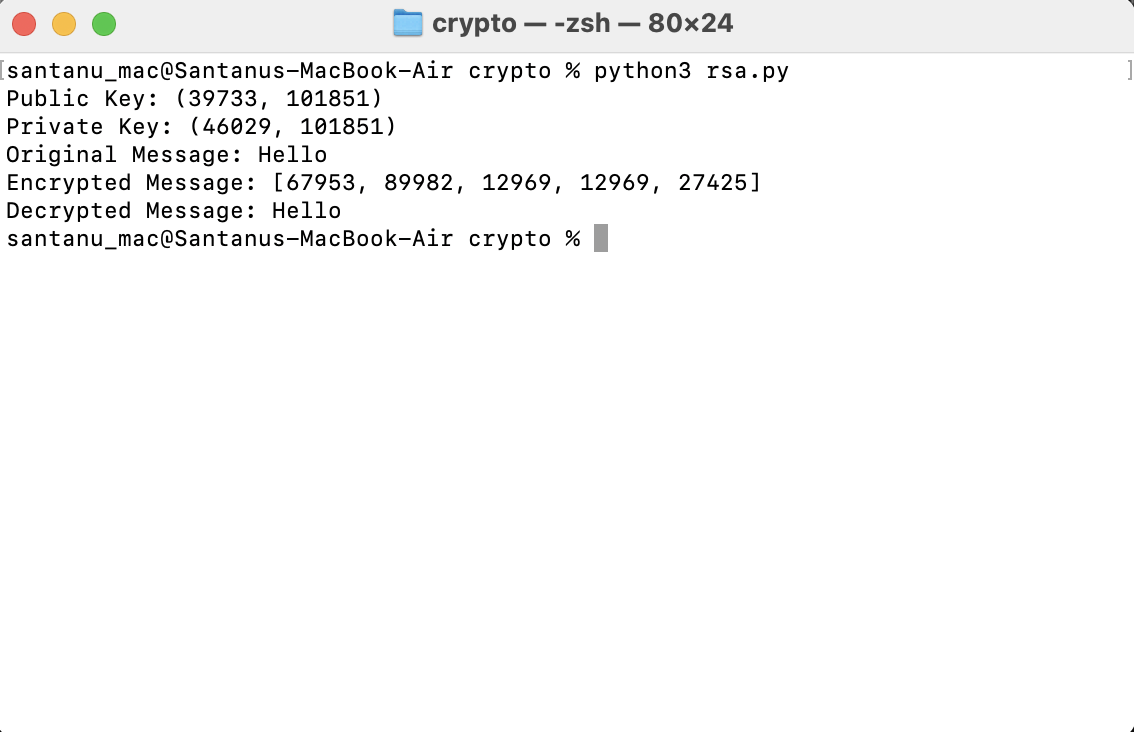
encrypted\_msg = encrypt(message, public\_key)

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)

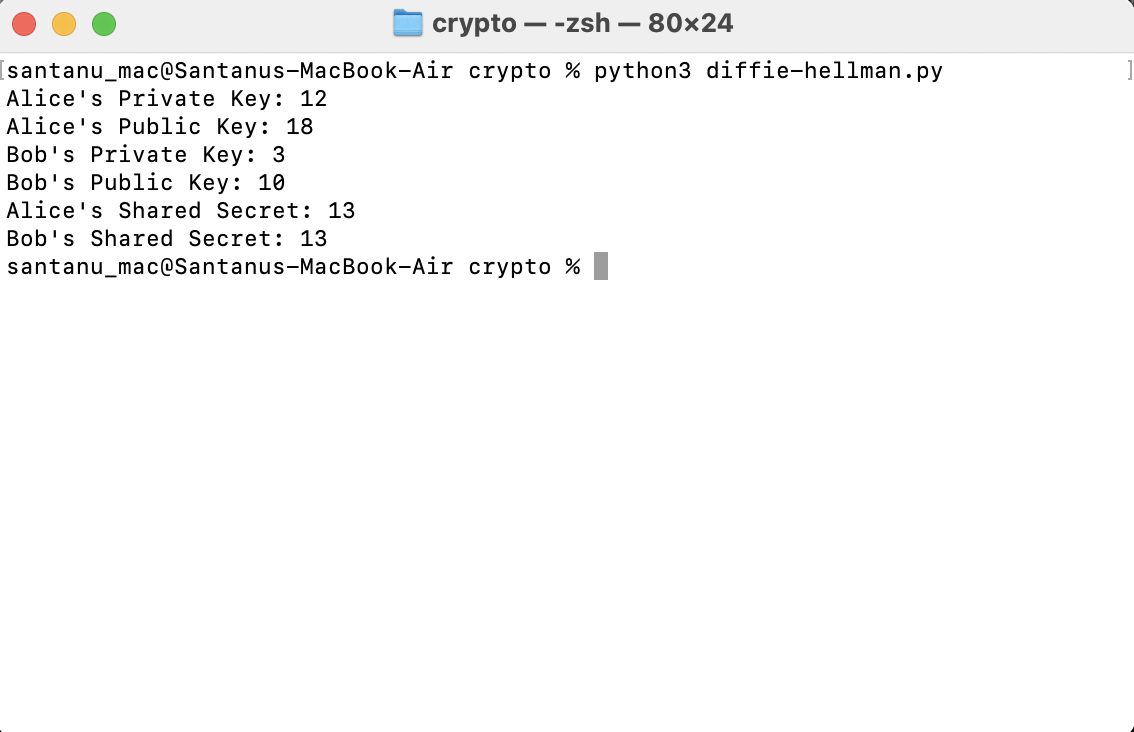
print("Bob's Private Key:", bob\_private\_key)

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: ")

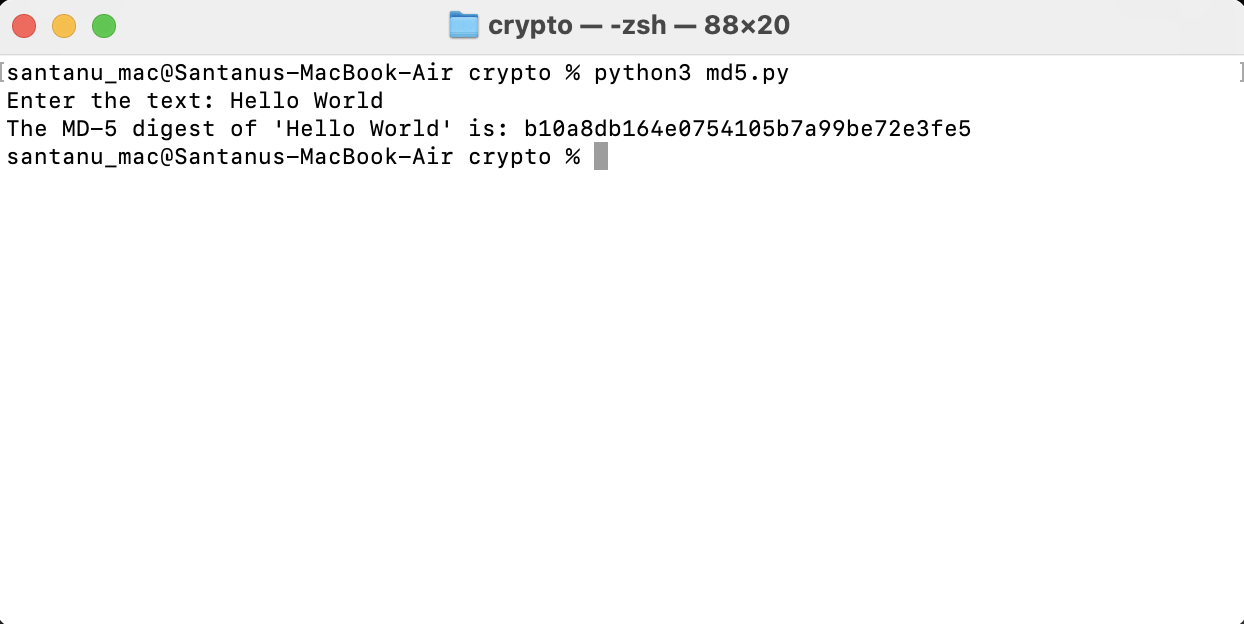
md5 = hashlib.md5()

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: ")

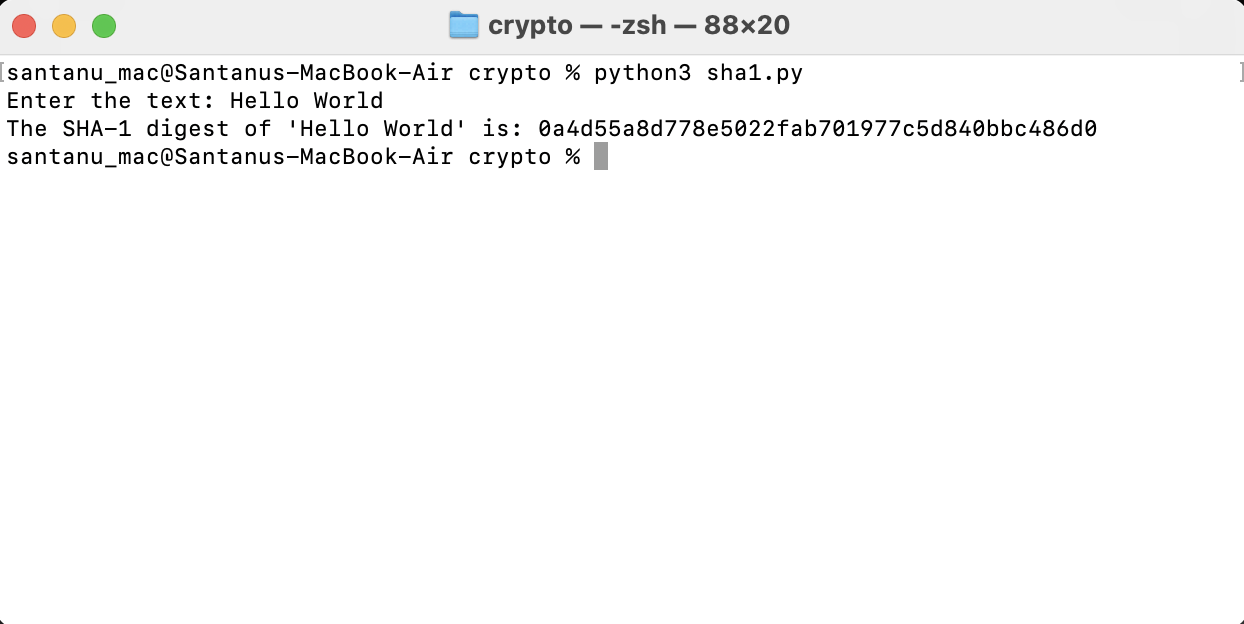
sha1 = hashlib.sha1()

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: ")

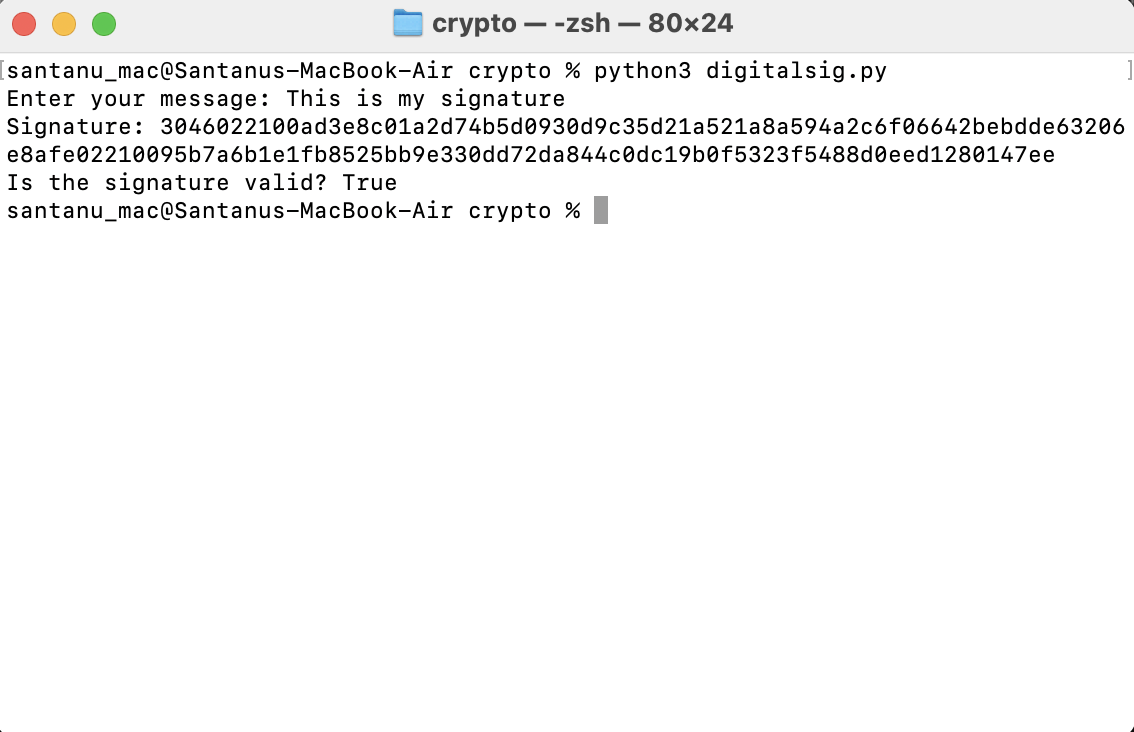
    signature = sign\_message(private\_key, message)

    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:**

****