PRACTICAL 10

DATE: 17-04-2025

AIM: Implement RSA Encryption-Decryption in Python.

Code:

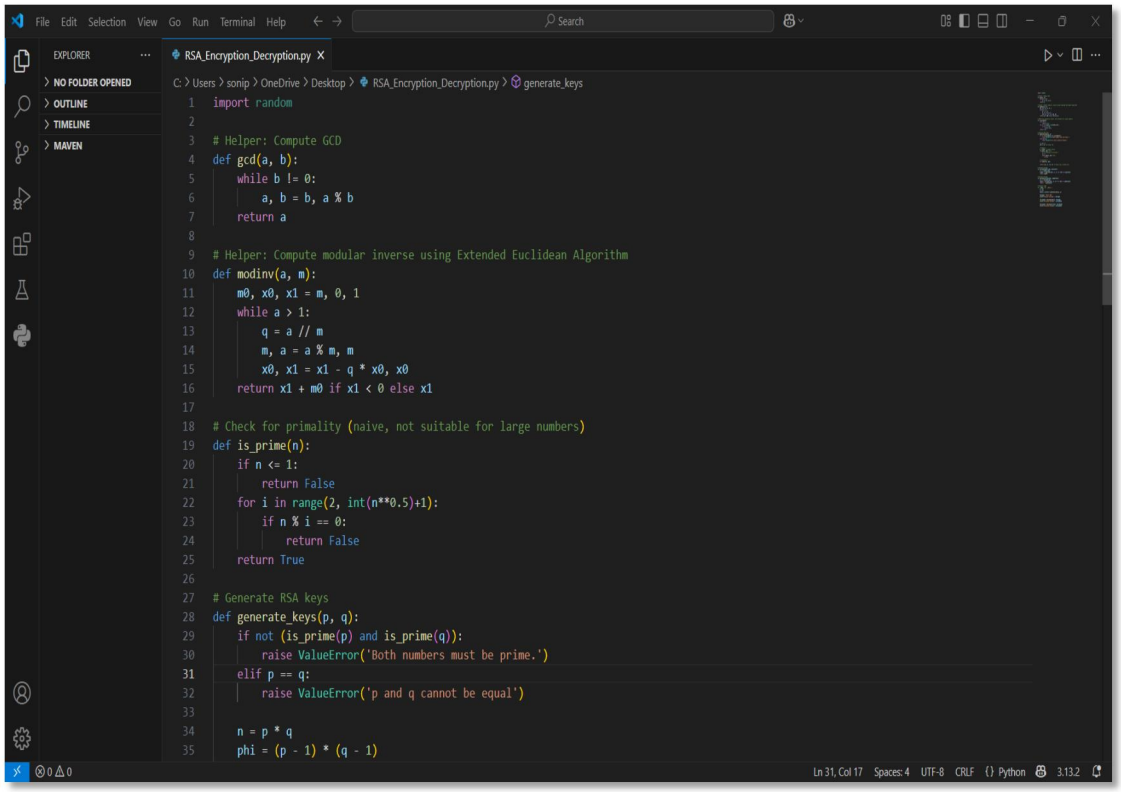
```
import random

# Helper: Compute GCD
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Helper: Compute modular inverse using Extended Euclidean Algorithm
def modinv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Check for primality (naive, not suitable for large numbers)
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

# Generate RSA keys
def generate_keys(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')

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

    # Choose e
    e = 65537  # Common choice
    if gcd(e, phi) != 1:
        # Try finding an alternate e
        e = 3
```

```
        while gcd(e, phi) != 1:
            e += 2

    # Calculate d
    d = modinv(e, phi)

    return ((e, n), (d, n))  # public key, private key

# Encrypt message
def encrypt(public_key, plaintext):
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in plaintext]
    return cipher

# Decrypt message
def decrypt(private_key, ciphertext):
    d, n = private_key
    plain = [chr(pow(char, d, n)) for char in ciphertext]
    return ''.join(plain)

# Example usage
if __name__ == '__main__':
    p = 61
    q = 53
    public, private = generate_keys(p, q)

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

    encrypted = encrypt(public, message)
    print("Encrypted message:", encrypted)

    decrypted = decrypt(private, encrypted)
    print("Decrypted message:", decrypted)
```

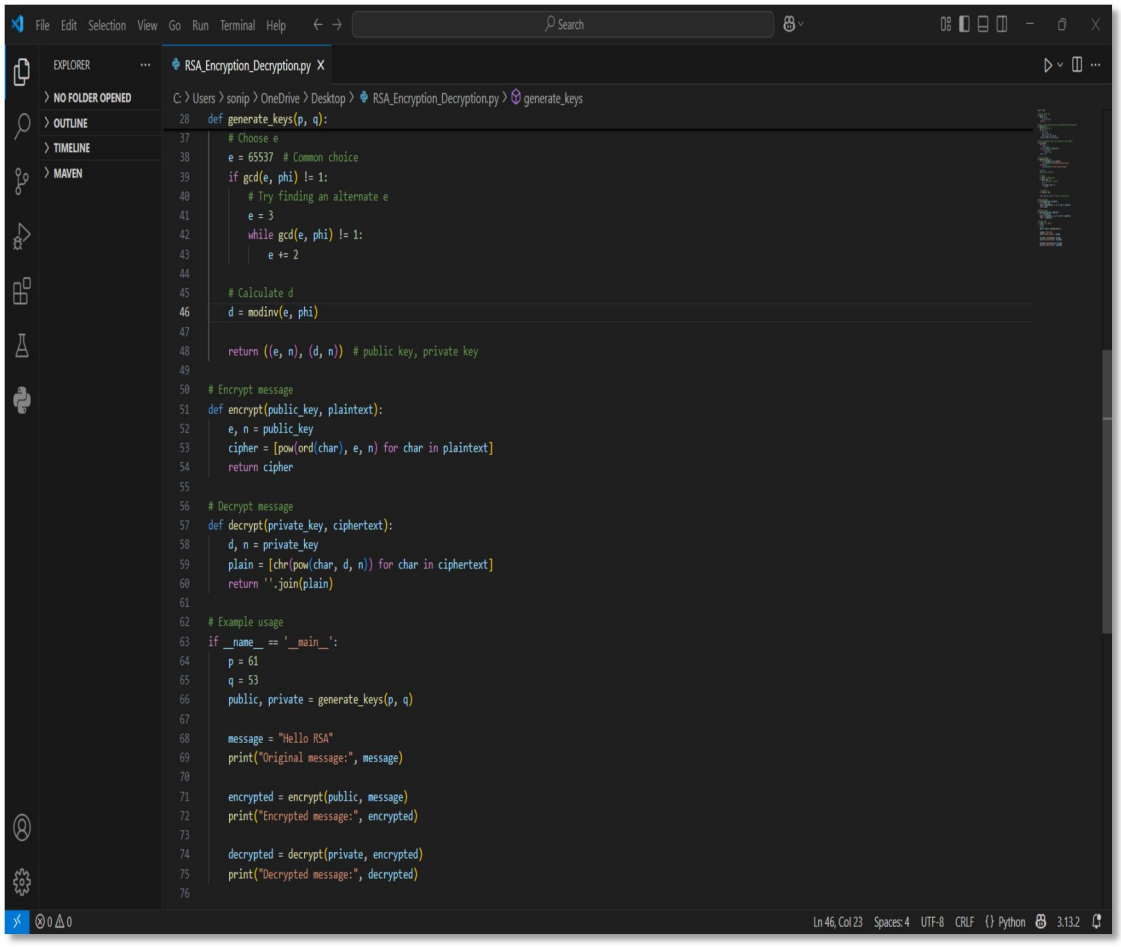
Output:

Original message: Hello RSA
Encrypted message: [3000, 1313, 745, 745, 2185, 1992, 1859, 2680, 2790]
Decrypted message: Hello RSA

Code:

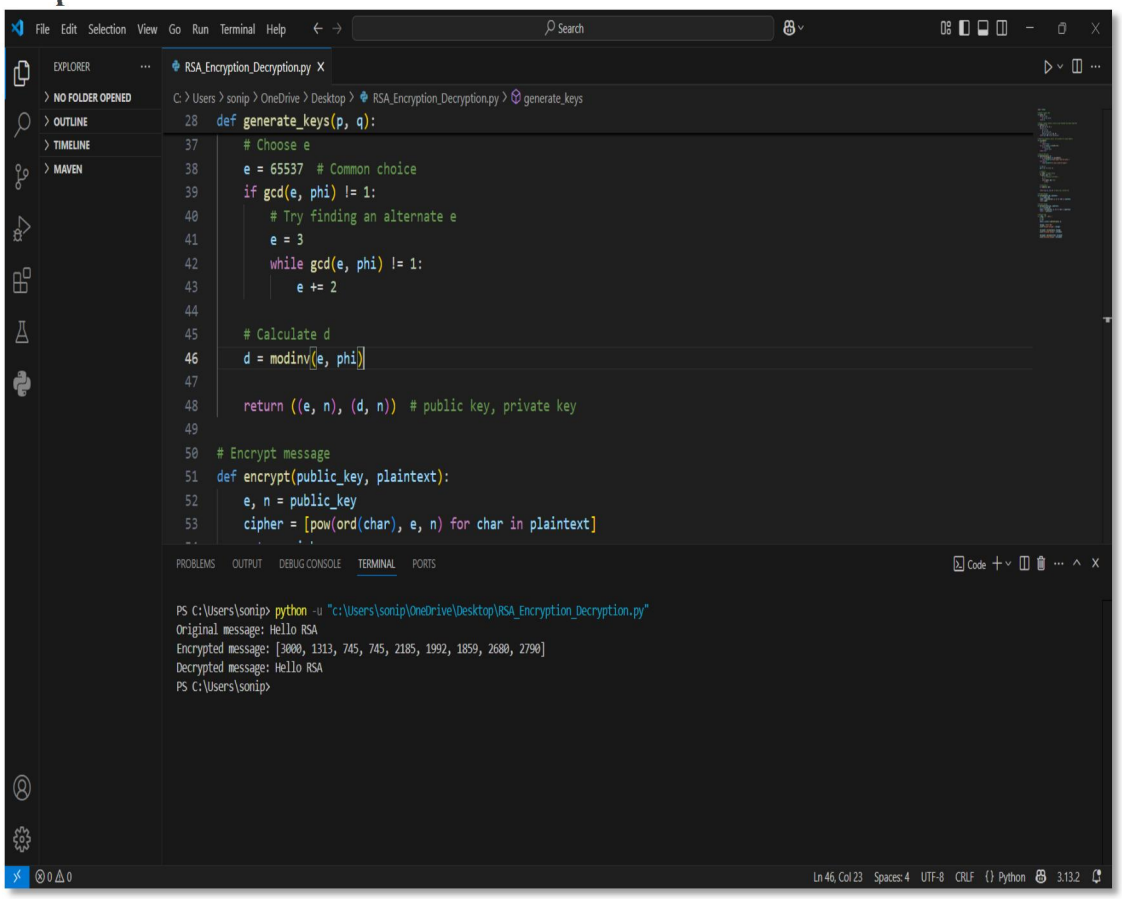


```python
import random

# Helper: Compute GCD
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Helper: Compute modular inverse using Extended Euclidean Algorithm
def modinv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Check for primality (naive, not suitable for large numbers)
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

# Generate RSA keys
def generate_keys(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')

    n = p * q
    phi = (p - 1) * (q - 1)
```



```python
def generate_keys(p, q):
    # Choose e
    e = 65537  # Common choice
    if gcd(e, phi) != 1:
        # Try finding an alternate e
        e = 3
        while gcd(e, phi) != 1:
            e += 2

    # Calculate d
    d = modinv(e, phi)

    return ((e, n), (d, n))  # public key, private key

# Encrypt message
def encrypt(public_key, plaintext):
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in plaintext]
    return cipher

# Decrypt message
def decrypt(private_key, ciphertext):
    d, n = private_key
    plain = [chr(pow(char, d, n)) for char in ciphertext]
    return ''.join(plain)

# Example usage
if __name__ == '__main__':
    p = 61
    q = 53
    public, private = generate_keys(p, q)

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

    encrypted = encrypt(public, message)
    print("Encrypted message:", encrypted)

    decrypted = decrypt(private, encrypted)
    print("Decrypted message:", decrypted)
```

Output:



Conclusion:

Implementing RSA encryption and decryption in Python is a valuable exercise for understanding the fundamentals of public-key cryptography. It involves generating a pair of keys using prime numbers, encrypting data with the public key, and decrypting it with the private key. This process helps illustrate important mathematical concepts such as modular arithmetic, Euler's totient function, and modular inverses. Through this implementation, one gains insight into how secure communication can be established without the need for sharing a secret key in advance.