Source Code:

```
def caesar_encrypt(plaintext, shift):
    encrypted_text = ""
    for char in plaintext:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr((ord(char) + shift - ord('a')) % 26 +
ord('a'))
            else:
                encrypted_text += chr((ord(char) + shift - ord('A')) % 26 +
ord('A'))
        else:
            encrypted_text += char
    return encrypted_text


def caesar_decrypt(ciphertext, shift):
    decrypted_text = ""
    for char in ciphertext:
        if char.isalpha():
            if char.islower():
                decrypted_text += chr((ord(char) - shift - ord('a')) % 26 +
ord('a'))
            else:
                decrypted_text += chr((ord(char) - shift - ord('A')) % 26 +
ord('A'))
        else:
            decrypted_text += char
    return decrypted_text


plaintext = "hello"
shift = 3
ciphertext = caesar_encrypt(plaintext, shift)
print(f"Caesar Cipher Encryption: {ciphertext}")
plaintext = caesar_decrypt(ciphertext, 3)
print(f"Caesar Cipher Decryption: {plaintext}")
```

Output:

Caesar Cipher Encryption: khoor

Caesar Cipher Decryption: hello

Source Code:

```
class MonoalphabeticCipher:
    def __init__(self):
        self.normal_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
                            'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
                            's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

        self.coded_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',
                           'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',
                           'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']

    def string_encryption(self, s):
        encrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.normal_char[i]:
                    encrypted_string += self.coded_char[i]
                    break
                elif char < 'a' or char > 'z':
                    encrypted_string += char
                    break
        return encrypted_string

    def string_decryption(self, s):
        decrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.coded_char[i]:
                    decrypted_string += self.normal_char[i]
                    break
                elif char < 'A' or char > 'Z':
                    decrypted_string += char
                    break
        return decrypted_string


def main():
    cipher = MonoalphabeticCipher()
    plain_text = "I am ICEIAN"

    print("Plain text:", plain_text)

    # Changing the whole string to lowercase
    encrypted_message = cipher.string_encryption(plain_text.lower())
    print("Encrypted message:", encrypted_message)

    decrypted_message = cipher.string_decryption(encrypted_message)
    print("Decrypted message:", decrypted_message)
```

```
main()
```

Output:

Plain text: I am ICEIAN

Encrypted message: O QD OETOQF

Decrypted message: i am iceian

Source Code:

```
def playfair_cipher(plaintext, key, mode):
    # Define the alphabet, excluding 'j'
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    # Remove whitespace and 'j' from the key and convert to lowercase
    key = key.lower().replace(' ', '').replace('j', 'i')
    # Construct the key square
    key_square = ''
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter
    # Split the plaintext into digraphs, padding with 'x' if necessary
    plaintext = plaintext.lower().replace(' ', '').replace('j', 'i')
    replaceplaintext = ''
    if mode == 'encrypt':
        it = 0
        while it < len(plaintext) - 1:
            if plaintext[it] == plaintext[it + 1]:
                replaceplaintext += plaintext[it]
                replaceplaintext += 'x'
                it += 1
            else:
                replaceplaintext += plaintext[it]
                replaceplaintext += plaintext[it + 1]
                it += 2
        replaceplaintext += plaintext[-1] if it < len(plaintext) else ''
        plaintext = replaceplaintext

    if len(plaintext) % 2 == 1:
        plaintext += 'x'
    digraphs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

    # Define the encryption/decryption functions
    def encrypt(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a + 1) % 5
            col_b = (col_b + 1) % 5
        elif col_a == col_b:
            row_a = (row_a + 1) % 5
            row_b = (row_b + 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    def decrypt(digraph):
        a, b = digraph
```

```
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a - 1) % 5
            col_b = (col_b - 1) % 5
        elif col_a == col_b:
            row_a = (row_a - 1) % 5
            row_b = (row_b - 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    # Encrypt or decrypt the plaintext
    result = ''
    for digraph in digraphs:
        if mode == 'encrypt':
            result += encrypt(digraph)
        elif mode == 'decrypt':
            result += decrypt(digraph)

    # Return the result
    return result


# Example usage
plaintext = 'caee'
key = 'monkey'
ciphertext = playfair_cipher(plaintext, key, 'encrypt')
print('Cipher Text:', ciphertext)
decrypted_text = playfair_cipher(ciphertext, key, 'decrypt')
print('Decrypted Text:', decrypted_text)  # (Note: 'x' is added as padding)
```

Output:

Cipher Text: dbkzkz

Decrypted Text: caexex

Source Code:

```
import string
import numpy as np

alphabet = string.ascii_lowercase
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))


def egcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = egcd(b % a, a)
        return gcd, y - (b // a) * x, x


def mod_inv(det, modulus):
    gcd, x, y = egcd(det, modulus)
    if gcd != 1:
        raise Exception("Matrix is not invertible.")
    return (x % modulus + modulus) % modulus


def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = mod_inv(det, modulus)
    matrix_modulus_inv = (
            det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    )
    return matrix_modulus_inv.astype(int)


def encrypt_decrypt(message, K):
    msg = ""
    message_in_numbers = [letter_to_index[letter] for letter in message]
    split_P = [
        message_in_numbers[i: i + len(K)]
        for i in range(0, len(message_in_numbers), len(K))
    ]
    for P in split_P:
        P = np.transpose(np.asarray(P))[:, np.newaxis]
        while P.shape[0] != len(K):
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]
        numbers = np.dot(K, P) % len(alphabet)
        n = numbers.shape[0]
        for idx in range(n):
            number = int(numbers[idx, 0])
            msg += index_to_letter[number]
```

```
    return msg


message = "help"
K = np.matrix([[3, 3], [2, 5]])
Kinv = matrix_mod_inv(K, len(alphabet))

encrypted_message = encrypt_decrypt(message, K)
decrypted_message = encrypt_decrypt(encrypted_message, Kinv)

print("Original message: " + message.upper())
print("Encrypted message: " + encrypted_message.upper())
print("Decrypted message: " + decrypted_message.upper())
```

Output:

Original message: HELP

Encrypted message: HIAT

Decrypted message: HELP

Source Code:

```
# Poly_alphabetic cipher

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generateKey(plainText, keyword):
    key = ''
    for i in range(len(plainText)):
        key += keyword[i % len(keyword)]
    return key


def cipherText(plainText, key):
    cipher_text = ""
    for i in range(len(plainText)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[plainText[i].upper()] + shift) % 26]
        cipher_text += newChar
    return cipher_text


def decrypt(cipher_text, key):
    plainText = ''
    for i in range(len(cipher_text)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[cipher_text[i].upper()] - shift + 26) % 26]
        plainText += newChar
    return plainText


plainText = "wearediscoveredsaveyourself"
keyword = "deceptive"
key = generateKey(plainText, keyword)
cipher_text = cipherText(plainText, key)
print("Ciphertext :", cipher_text)
print("Decrypted Text :", decrypt(cipher_text, key))
```

Output:

Ciphertext : ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Decrypted Text : WEAREDISCOVEREDSAVEYOURSELF

Source Code:

```
import random

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generate_key(length):
    key = ""
    for i in range(length):
        key += chr(random.randint(65, 90))  # ASCII codes for A-Z
    return key


def encrypt(plaintext, key):
    ciphertext = ""
    cipherCode = []
    for i in range(len(plaintext)):
        xor = mp[plaintext[i]] ^ mp[key[i]]
        cipherCode.append(xor)
        ciphertext += mp2[(mp['A'] + xor) % 26]
    return ciphertext, cipherCode


def decrypt(cipherCode, key):
    plaintext = ""
    for i in range(len(cipherCode)):
        xor = cipherCode[i] ^ mp[key[i]]
        plaintext += mp2[xor % 26]
    return plaintext


plaintext = "OAK"
plaintext = plaintext.upper()
key = generate_key(len(plaintext))
ciphertext, cipherCode = encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
decryptedtext = decrypt(cipherCode, key)
print("Decrypted text:", decryptedtext)
```

Output:

Ciphertext: NBE

Decrypted text: OAK

Source Code:

```
def brute_force_decrypt(ciphertext):
    for shift in range(26):
        decrypted_text = caesar_decrypt(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_text}")

def brute_force_encrypt(plainText):
    for shift in range(26):
        encrypted_text = caesar_encrypt(plainText, shift)
        print(f"Shift {shift}: {encrypted_text}")


def caesar_encrypt(plainText, shift):
    encrypted_text = ""
    for char in plainText:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr((ord(char) + shift - ord('a')) % 26 +
ord('a'))
            else:
                encrypted_text += chr((ord(char) + shift - ord('A')) % 26 +
ord('A'))
        else:
            encrypted_text += char
    return encrypted_text


def caesar_decrypt(ciphertext, shift):
    decrypted_text = ""
    for char in ciphertext:
        if char.isalpha():
            if char.islower():
                decrypted_text += chr((ord(char) - shift - ord('a')) % 26 +
ord('a'))
            else:
                decrypted_text += chr((ord(char) - shift - ord('A')) % 26 +
ord('A'))
        else:
            decrypted_text += char
    return decrypted_text

plantext='hello'
print('Brute Force Encryption for Caesar Cipher:')
brute_force_encrypt(plantext)

ciphertext = "ifmmp"
print("\nBrute Force Decryption for Caesar Cipher:")
brute_force_decrypt(ciphertext)
```

Output:

Brute Force Decryption for Caesar Cipher:

Shift 0: ifmmp

Shift 1: hello

Shift 2: gdkkn

Shift 3: fcjjm

Shift 4: ebiil

Shift 5: dahhk

Shift 6: czggj

Shift 7: byffi

Shift 8: axeeh

Shift 9: zwddg

Shift 10: yvccf

Shift 11: xubbe

Shift 12: wtaad

Shift 13: vszzc

Shift 14: uryyb

Shift 15: tqxxa

Shift 16: spwwz

Shift 17: rovvy

Shift 18: qnuux

Shift 19: pmttw

Shift 20: olssv

Shift 21: nkrru

Shift 22: mjqqt

Shift 23: lipps

Shift 24: khoor

Shift 25: jgnnq