Caesar Cipher

```
def encrypt_text(plaintext,n):
ans = ""
for i in range(len(plaintext)):
ch = plaintext[i]
if ch==" ":
ans+=" "
elif (ch.isupper()):
ans += chr((ord(ch) + n-65) % 26 + 65)
else:
ans += chr((ord(ch) + n-97) % 26 + 97)
return ans
plaintext = "HELLO EVERYONE"
n = 1
print("Plain Text is : " + plaintext)
print("Shift pattern is : " + str(n))
print("Cipher Text is : " + encrypt_text(plaintext,n))
```

Monoalphabetic

```
def generate_cipher_key(shift):
alphabet = 'abcdefghijklmnopqrstuvwxyz'
shifted_alphabet = alphabet[shift:] + alphabet[:shift]
key = dict(zip(alphabet, shifted_alphabet))
return key
def encrypt(message, key):
encrypted_message = ''
for char in message:
if char.isalpha():
if char.islower():
encrypted_message += key[char]
else:
encrypted_message += key[char.lower()].upper()
else:
encrypted_message += char
return encrypted_message
def decrypt(ciphertext, key):
reverse_key = {v: k for k, v in key.items()}
decrypted_message = ''
for char in ciphertext:
if char.isalpha():
if char.islower():
decrypted_message += reverse_key[char]
else:
decrypted_message +=
reverse_key[char.lower()].upper()
else:
decrypted_message += char
return decrypted_message
def main():
shift = int(input("Enter the shift value for the cipher:
"))
key = generate_cipher_key(shift)
choice = input("Encrypt or decrypt? (e/d): ").lower()
if choice == 'e':
plaintext = input("Enter the message to encrypt: ")
encrypted = encrypt(plaintext, key)
print("Encrypted message:", encrypted)
elif choice == 'd':
ciphertext = input("Enter the message to decrypt: ")
decrypted = decrypt(ciphertext, key)
print("Decrypted message:", decrypted)
else:
print("Invalid choice. Please enter 'e' for encrypt or
'd' for decrypt.")
if __name__ == "__main__":
main()
```

Message Authentication(SHA)

```
import hashlib
str = "GeeksforGeeks"
result = hashlib.sha256(str.encode())
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())
result = hashlib.sha384(str.encode())
print("The hexadecimal equivalent of SHA384 is : ")
print(result.hexdigest())
result = hashlib.sha224(str.encode())
print("The hexadecimal equivalent of SHA224 is : ")
print(result.hexdigest())
result = hashlib.sha512(str.encode())
# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA512 is : ")
print(result.hexdigest())
result = hashlib.sha1(str.encode())
print("The hexadecimal equivalent of SHA1 is:")
print(result.hexdigest())
```

DES

```
def hex2bin(s):
  mp = {hex(i)[2:].upper(): format(i, '04b') for i in
range(16)}
  return ''.join(mp[ch] for ch in s)
def bin2hex(s):
  mp = {format(i, '04b'): hex(i)[2:].upper() for i in
range(16)}
  return ''.join(mp[s[i:i+4]] for i in range(0, len(s), 4))
def bin2dec(binary):
  return int(binary, 2)
def dec2bin(num):
  return format(num, '04b')
def permute(k, arr):
  return ''.join(k[i-1] for i in arr)
def shift_left(k, shifts):
  return k[shifts:] + k[:shifts]
def xor(a, b):
  return ''.join('0' if i == j else '1' for i, j in zip(a, b))
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17, 9, 1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7]
def encrypt(pt, rkb, rk):
  pt = permute(hex2bin(pt), initial_perm)
  left, right = pt[:32], pt[32:]
  for i in range(16):
    right_expanded = permute(right, exp_d)
    xor_x = xor(right_expanded, rkb[i])
    sbox_str = "".join(
      dec2bin(sbox[j][bin2dec(xor_x[j*6] +
xor_x[j*6+5])]
            [bin2dec(xor_x[j*6+1:j*6+5])]])
      for j in range(8)
    )
    result = xor(left, permute(sbox_str, per))
    left, right = right, result if i != 15 else result, right
  return bin2hex(permute(left + right, final_perm))
pt = "123456ABCD132536"
key = "AABB09182736CCDD"
key_bin = permute(hex2bin(key), keyp)
print("Cipher Text:", encrypt(pt, round_keys_bin,
round_keys_hex))
```

AES

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
def encrypt(data, key):
cipher = AES.new(key, AES.MODE_CBC)
padded_data = pad(data.encode(), AES.block_size)
ciphertext = cipher.encrypt(padded_data)
return cipher.iv, ciphertext
def decrypt(iv, ciphertext, key):
cipher = AES.new(key, AES.MODE_CBC, iv=iv)
```

```
decrypted_data = cipher.decrypt(ciphertext)

return unpad(decrypted_data,
AES.block_size).decode()

if __name__ == "__main__":

key = get_random_bytes(16)

data = "This is a secret message."

iv, ciphertext = encrypt(data, key)

print(f"Ciphertext: {ciphertext.hex()}")

decrypted_data = decrypt(iv, ciphertext, key)

print(f"Decrypted data: {decrypted_data}")
```

DES short code

```
from pyDes import des, CBC, PAD_PKCS5

import binascii

def encrypt_decrypt(data, key):

   cipher = des(key, CBC, key, padmode=PAD_PKCS5)

   encrypted = cipher.encrypt(data)

   decrypted = cipher.decrypt(encrypted)

   return binascii.hexlify(encrypted).decode(),
decrypted.decode()

key = b"8bytekey”

data = "Hello123"

ciphertext, decrypted_text = encrypt_decrypt(data,
key)

print(f"Ciphertext: {ciphertext}")

print(f"Decrypted Text: {decrypted_text}")
```

RSA

```
from math import gcd

def RSA(p: int, q: int, message: int):

   n = p * q

   t = (p - 1) * (q - 1)

   for i in range(2, t):

      if gcd(i, t) == 1:

         e = i

         break

   d = 0

   for j in range(1, t):

      if (j * e) % t == 1:

         d = j

         break

   ct = (message ** e) % n

   print(f"Encrypted message is {ct}")

   mes = (ct ** d) % n

   print(f"Decrypted message is {mes}")

RSA(p=53, q=59, message=89)

RSA(p=3, q=7, message=12)
```

Diffie-Hellman(Secure key exchange)

```
import random

p = int(input('Enter a prime number: '))

g = int(input('Enter a number: '))

class Participant:

   def __init__(self):

      self.n = random.randint(1, p)

   def publish(self):

      return pow(g, self.n, p)

   def compute_secret(self, received):

      return pow(received, self.n, p)

alice, bob = Participant(), Participant()

eve = [random.randint(1, p) for _ in range(2)]

print(f'Alice selected (a): {alice.n}')

print(f'Bob selected (b): {bob.n}')

print(f'Eve selected private numbers (c, d): {eve}')

ga, gb = alice.publish(), bob.publish()

gea, geb = pow(g, eve[0], p), pow(g, eve[1], p)

print(f'Alice published (ga): {ga}')

print(f'Bob published (gb): {gb}')

print(f'Eve published values (gc, gd): {gea}, {geb}')

sa, sea = alice.compute_secret(gea), pow(ga, eve[0],
p)

sb, seb = bob.compute_secret(geb), pow(gb, eve[1],
p)

print(f'Alice computed (S1): {sa}')

print(f'Eve computed key for Alice (S1): {sea}')

print(f'Bob computed (S2): {sb}')

print(f'Eve computed key for Bob (S2): {seb}')
```

Digital Sign Gen

```
from Crypto.PublicKey import RSA

from Crypto.Signature import pkcs1_15

from Crypto.Hash import SHA256

from Crypto.Random import get_random_bytes

key = RSA.generate(2048)

private_key = key.export_key()

public_key = key.publickey().export_key()

with open('private.pem', 'wb') as f:

   f.write(private_key)

with open('public.pem', 'wb') as f:

   f.write(public_key)

def sign_message(message, private_key):

   key = RSA.import_key(private_key)

   h = SHA256.new(message.encode())

   signature = pkcs1_15.new(key).sign(h)

   return signature

def verify_signature(message, signature,
public_key):

   key = RSA.import_key(public_key)

   h = SHA256.new(message.encode())

   try:

      pkcs1_15.new(key).verify(h, signature)

      return True

   except (ValueError, TypeError):

      return False

if __name__ == "__main__":

   message = "This is a secret message."

   signature = sign_message(message, private_key)

   print(f"Signature: {signature.hex()}")

   is_valid = verify_signature(message, signature,
public_key)

   print(f"Signature valid: {is_valid}")
```