1. **Caesar Cipher**

def encrypt\_func(txt,s):

    result=""

    for i in range(len(txt)):

        char=txt[i]

        if(char.isupper()):

            result+=chr((ord(char)+s-65)%26+65)

        else:

            result+=chr((ord(char)+s-97)%26+97)

    return result

txt="PLAYSTATION"

s=4

print("Plain text is: "+txt)

print("Shift pattern: "+str(s))

print("Cipher text is: "+encrypt\_func(txt,s))

1. **RSA Encryption Algorithm**

try:

    input=raw\_input

except NameError:

    pass

try:

    chr=unichr

except NameError:

    pass

p=int(input("Enter prime p: "))

q=int(input("Enter prime q: "))

print("Choose primes:\np="+str(p)+",q="+str(q)+"\n")

n=p\*q

print("n=p\*q= "+str(n)+"\n")

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

print("Euler's function (totient) {phi(n)}:"+str(phi)+"\n")

def gcd(a,b):

    while b!=0:

        c=a%b

        a=b

        b=c

    return a

def modinv(a,m):

    for x in range(1,m):

        if(a\*x)% m==1:

            return x

    return None

def coprimes(a):

    y=[]

    for x in range(2,a):

        if gcd(a,x)==1 and modinv(x,phi)!=None:

            y.append(x)

    for x in y:

        if x==modinv(x,phi):

            y.remove(x)

    return y

print("Choose an e from a below coprimes array:\n")

print(str(coprimes(phi))+"\n")

e=int(input())

d=modinv(e,phi)

print("\n Your public key is a pair of numbers(e="+str(e)+",n="+str(n)+").\n")

print("\n Your private key is a pair of numbers(d="+str(d)+",n="+str(n)+").\n")

def encrypt\_block(m):

    c=modinv(m\*\*e,n)

    if c == None:

        print("No Modular Multiplicative inverse for block "+str(m)+".")

    return c

def decrpyt\_block(c):

    m=modinv(c\*\*d,n)

    if m==None:

        print("No Modular Multiplicative inverse for block "+str(c)+".")

    return m

def encrypt\_string(s):

    return ''.join([chr(encrypt\_block(ord(x))) for x in list(s)])

def decrypt\_string(s):

    return ''.join([chr(decrpyt\_block(ord(x))) for x in list(s)])

s=input("Enter a message to encrypt: ")

print("\n Plain message: "+s+"\n")

enc=encrypt\_string(s)

print("Encrypted message: "+enc+"\n")

dec=decrypt\_string(enc)

print("Decrypted message: "+dec+"\n")

for output :

7

11

37

9

1. **SHA-256**

import hashlib

string="Playstation2020"

encoded=string.encode()

result=hashlib.sha256(encoded)

print("String :",end="")

print(string)

print("Hash Value: ",end="")

print(result)

print("Hexadecimal equivalent: ",result.hexdigest())

print("Digest Size :",end="")

print(result.digest\_size)

print("Block Size :",end="")

print(result.block\_size)

1. **Merkel Tree**

from hashlib import sha256

class MerkelNode:

    """Stores the hash and the parent."""

    def \_\_init\_\_(self,hash):

        self.hash=hash

        self.parent=None

class MerkelTree:

    """Stores the leaves and root hash of the tree."""

    def \_\_init\_\_(self,data\_chunks):

        leaves=[]

        for chunk in data\_chunks:

            node=MerkelNode(self.compute\_hash(chunk))

            leaves.append(node)

        self.root=self.build\_merkel\_tree(leaves)

    def build\_merkel\_tree(self,leaves):

        """builds the Merkel tree from a list of leaves. In case of an odd number of leaves, the last leaf is duplicated."""

        num\_leaves=len(leaves)

        if num\_leaves == 1:

            return leaves[0]

        parents=[]

        i=0

        while i < num\_leaves:

            left\_child=leaves[i]

            right\_child=leaves[i+1] if i+1 < num\_leaves else left\_child

            parents.append(self.create\_parent(left\_child,right\_child))

            i+=2

        return self.build\_merkel\_tree(parents)

    def create\_parent(self,left\_child,right\_child):

        """Creates the parent node from the children, and updates their parent field."""

        parent=MerkelNode(self.compute\_hash(left\_child.hash+right\_child.hash))

        left\_child.parent,right\_child.parent=parent,parent

        print("Left child: {}, Right child: {}, Parent: {}".format(left\_child.hash,right\_child.hash,parent.hash))

        return parent

    @staticmethod

    def compute\_hash(data):

        data=data.encode('utf-8')

        return sha256(data).hexdigest()

**for output:**

**file="hello"**

**chunks = list(file)**

**chunks**

**merkle\_tree = MerkelTree(chunks)**

1. **Implement the creation of Bitcoin Bloc/Blockchain(Genesis Block)**

blockchain=[]

def get\_last\_value():

    return (blockchain[-1])

def add\_value(sender,recipient,amount=1.0):

    transaction={'sender':sender,'recipient':recipient,'amount':amount}

    blockchain.append(transaction)

def get\_transaction\_value():

    tx\_sender=input('Enter the sender: ')

    tx\_recipient=input('Enter the recipient of the transaction: ')

    tx\_amount=float(input('Enter your transaction: '))

    return tx\_sender,tx\_recipient,tx\_amount

def print\_block():

    for block in blockchain:

        print("Here is you block")

        print(block)

again=True

while again==True:

    tx=get\_transaction\_value()

    s,r,a=tx

    add\_value(s,r,a)

    print(blockchain)

    more=input("add more block (Y/N)? ")

    if more.lower()=='y':

        again=True

    else:

        again=False

**for Output :**

**Ace**

**Beta**

**1000**

**y**

**Ace**

**Delta**

**3000**

1. **Implement the creation of a Blockchain(Adding the blocks to the chain and validating)**

#implent the creation of blockchain (Adding the blocks to the chain and validating)

import hashlib as hasher

import datetime as date

class Block:

    def \_\_init\_\_(self, index, timestamp, data, previous\_hash):

        self.index = index

        self.timestamp = timestamp

        self.data = data

        self.previous\_hash = previous\_hash

        self.hash = self.hash\_block()

    def \_\_repr\_\_(self):

        return "index %04d: \n Time %s, \n Data %s : \n Previous hash %s" % (self.index, str(self.timestamp), str(self.data), str(self.previous\_hash))

    def hash\_block(self):

        sha = hasher.sha256()

        sha.update(repr(self).encode('ascii'))

        return sha.hexdigest()

def create\_genesis\_block():

    return Block(0, date.datetime.now(), "Genesis Block", "0")

blockchain = [create\_genesis\_block()]

previous\_block = blockchain[0]

blockchain

def next\_block(last\_block):

    this\_index = last\_block.index + 1

    this\_timestamp = date.datetime.now()

    this\_data = "Hey! I,m block " + str(this\_index)

    this\_hash = last\_block.hash

    return Block(this\_index, this\_timestamp, this\_data, this\_hash)

num\_of\_block\_to\_add = 5

for i in range (0, num\_of\_block\_to\_add):

    block\_to\_add = next\_block(previous\_block)

    blockchain.append(block\_to\_add)

    previous\_block = block\_to\_add

    print(repr(block\_to\_add))

    print(" \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_")

def validate\_blockchain(in\_blockchain):

    for current\_position in range(1, len(in\_blockchain)):

        previous\_position = current\_position - 1

        if in\_blockchain [previous\_position].hash\_block() == in\_blockchain[current\_position].previous\_hash:

            print('Block %d is valid' % current\_position)

        else:

            warn('Block %d is invalid! (%s)' % (current\_position, repr(in\_blockchain[current\_position])))

            break

validate\_blockchain(blockchain)

1. **PRIVATE BLOCKCHAIN USING GETH (Implement the creation of a public/private Blockchain)**

1)  Download and Install NodeJs

2)  Installation Geth 64 bit window 1.7.2 stable

3)  Create folder "Private-chain"

4)  Inside "Private-chain" create new folder "chaindata"

5)  Create "genesis.json" in "Private-chain"

6)  Paste inside "genesis.json"

{

    "config": {

    "chainId": 4777,

    "homesteadBlock": 0,

    "eip150Block": 0,

    "eip155Block": 0,

    "eip158Block": 0

    },

    "alloc" : {},

    "difficulty" : "0x400",

    "extraData" : "",

    "gasLimit" : "0x7A1200",

    "parentHash" :

    "0x0000000000000000000000000000000000000000000000000000000000000000",

    "timestamp" : "0x00"

}

7)

    geth --datadir chaindata init genesis.json  -> It will Intialize geth into chaindata

    geth --datadir=./chaindata/         -> Now we can start Geth and connect to our own private chain

8) new terminal

    geth attach ipc:\\.\pipe\geth.ipc       -> IPC to interact with Geth

    personal.newAccount()               -> Create Account

    eth.accounts                    -> show all account

    eth.coinbase

    eth.getBalance(eth.accounts[0])         -> get balance

    miner.start()                   -> start min

        "Wait for successfully sealed."

    miner.stop()                    -> stop min

    eth.blockNumber                 -> block number

    personal.unlockAccount(eth.accounts[0])     -> go first account

    eth.sendTransaction({from: eth.coinbase, to: eth.accounts[1], value: web3.toWei(10, "ether")})    -> create transaction

    start min -> "Wait for successfully sealed." -> stop min

    check balance