# ASENFA Encryption System — Complete Function Reference

Includes documentation and code snippets for all core functions. To run the program use: runASENFA.exe — this will open two terminals: one for the sender and one for the receiver.

## generate_key()

Function: generate_key
----------------------
Generates a cryptographic random key of a specific bit length based on the user's selected option.

Parameters:
length (int): Determines which key size to use (1 → 2048 bits, 2 → 3072 bits, 3 → random choice).

Returns:
int: A randomly generated integer representing the cryptographic key.

```
import secrets

def generate_key(length: int):
    """
    Generates a cryptographically secure random key.
    length: 1 -> 2048 bits, 2 -> 3072 bits, 3 -> random choice between 2048 and 3072
    """
    if length == 1:
        key_size = 2048
    elif length == 2:
        key_size = 3072
    elif length == 3:
        key_size = secrets.choice([2048, 3072])
    else:
        print("Error: 'length' parameter must be between 1 and 3.")
        return None
    key = secrets.randbits(key_size)
    return key
```

## encriptkey_main(len_key: int)

Function: encriptkey_main
-------------------------
Generates and encodes an encryption key by combining a real key with fake padding and mappings.
Returns:
enc_key (str): Encoded key (string).
key (int): Original numeric key.
num_of_gen (int): Number of keys to generate (3 or 4).
drop_key (int): Index of key to drop.

```
import random

def encriptkey_main(len_key: int):
    """
    Generate an encoded key mixing fake data and a real generated key.
    Dependencies: fakesize(), generate_key(), shfit(), option, s
    """
    fake_len = fakesize()
    list_fake_list = list(str(fake_len))
    if len(list_fake_list) == 309:
        list_fake_list.pop()
    elif len(list_fake_list) == 307:
        fake_num = random.randint(0,9)
        list_fake_list.append(str(fake_num))
    sh_f = list_fake_list
    for ind, val in enumerate(sh_f):
        if val == "0":
            num = int(list_fake_list[ind+1])
            break
    if num == 0:
        num = 2
    for ind, val in enumerate(sh_f):
        if val == "1":
            op = int(sh_f[ind+1])
            break
    if op == 0:
        op = 3
    for ind, val in enumerate(reversed(sh_f)):
        if val == "7":
            num_of_gen = int(sh_f[ind+1])
            break
    if 5 <= num_of_gen <= 7 or num_of_gen == 0 or num_of_gen == 1 or num_of_gen == 2:
        num_of_gen = 3
    elif num_of_gen > 7 or num_of_gen == 4:
        num_of_gen = 4
    for ind, val in enumerate(reversed(sh_f)):
        if val == "8":
            drop_key = int(sh_f[ind+1])
            break
    if drop_key == 0:
        drop_key = 1
    elif 2 <= drop_key <= 6:
        drop_key = 2
    elif drop_key > 6:
        if num_of_gen == 4:
            drop_key = 3
        else:
            drop_key = 1
    key = generate_key(len_key)
    list_key = list(str(key))
    l_k = shfit(list_key, num)
    list_map = option[op]
    for index, value in enumerate(l_k):
        for k, v in list_map.items():
            if int(value) == k:
                l_k[index] = v
    for index, value in enumerate(sh_f):
        for k, v in s.items():
            if int(value) == k:
                sh_f[index] = v
    final_list = l_k + sh_f
    enc_key = "".join(final_list)
    return enc_key, key, num_of_gen, drop_key
```

## decript_key(key: str)

Function: decript_key
---------------------
Reverses the encoding produced by encriptkey_main and returns the original key and metadata.

Parameters:
key (str): The encoded key string produced by encriptkey_main.

Returns:
result (int): The original numeric key.
number_of_keys (int): Number of keys generated (3 or 4).
number_drop_key (int): Index of dropped key.

```
def decript_key(key: str):
    """
    Decrypts an encoded key string back to the original integer key.
    Dependencies: option, s, unshfit()
    """
    sh_num, map_num, number_of_keys, number_drop_key = 0, 0, 0, 0
    list_key = list(key)
    fakekey_list = list_key[-308:]
    real_key = list(list_key[:-308])
    for index, value in enumerate(fakekey_list):
        for k, v in s.items():
            if value == v:
                fakekey_list[index] = k
    for ind, val in enumerate(fakekey_list):
        if val == 0:
            sh_num = fakekey_list[ind+1]
            break
    for ind, val in enumerate(fakekey_list):
        if val == 1:
            map_num = fakekey_list[ind+1]
            break
    for ind, val in enumerate(reversed(fakekey_list)):
        if val == 7:
            number_of_keys = fakekey_list[ind+1]
            break
    for ind, val in enumerate(reversed(fakekey_list)):
        if val == 8:
            number_drop_key = fakekey_list[ind+1]
            break
    if sh_num == 0:
        sh_num = 2
    if map_num == 0:
        map_num = 3
    if 5 <= number_of_keys <= 7 or number_of_keys in (0,1,2):
        number_of_keys = 3
    elif number_of_keys > 7 or number_of_keys == 4:
        number_of_keys = 4
    if number_drop_key == 0:
        number_drop_key = 1
    elif 2 <= number_drop_key <= 6:
        number_drop_key = 2
    elif number_drop_key > 6:
        number_drop_key = 3 if number_of_keys == 4 else 1
    ma_p = option[map_num]
    for index, value in enumerate(real_key):
        for k, v in ma_p.items():
            if value == v:
                real_key[index] = k
    list_key = unshfit(real_key, sh_num)
    result = int(''.join(map(str, list_key)))
    return result, number_of_keys, number_drop_key
```

## shfit(l: list, num: int)

Function: shfit
---------------
Performs a circular forward shift on a list of elements. Used to obfuscate key digits.

Parameters:
l (list): The list to shift.
num (int): Number of positions to shift (1..15 supported).

Returns:
list: Shifted list.

```
def shfit(l: list, num: int):
    c = []
    for t in range(len(l)):
        if num == 1:
            c.insert(0, l[-1])
            for i in range(len(l)):
                if i == len(l) - 1:
                    pass
                else:
                    c.insert(i+num, l[i])
            break
        if num == 2:
            c.insert(0, l[-2])
            c.insert(1, l[-1])
            for i in range(len(l)):
                if i == len(l) - 2 or i == len(l) - 1:
                    pass
                else:
                    c.insert(i+num, l[i])
            break
        # ... supports num up to 15, pattern repeated ...
        if num == 15:
            c.insert(0, l[-15])
            c.insert(1, l[-14])
            c.insert(2, l[-13])
            c.insert(3, l[-12])
            c.insert(4, l[-11])
            c.insert(5, l[-10])
            c.insert(6, l[-9])
            c.insert(7, l[-8])
            c.insert(8, l[-7])
            c.insert(9, l[-6])
            c.insert(10, l[-5])
            c.insert(11, l[-4])
            c.insert(12, l[-3])
            c.insert(13, l[-2])
            c.insert(14, l[-1])
            for i in range(len(l)):
                if (i >= len(l) - 15) or (i >= len(l) - 2 and i <= len(l) - 1):
                    pass
                else:
                    c.insert(i+num, l[i])
            break
    return c
```

## unshfit(l: list, num: int)

Function: unshfit
-----------------
Reverses the circular shift applied by shfit.

Parameters:
l (list): The shifted list.

num (int): The number of positions originally shifted.

Returns:
list: Original unshifted list.

```
def unshfit(l: list, num: int):
    c = []
    for t in range(len(l)):
        if num == 1:
            for i in range(len(l)):
                if i == len(l) -1:
                    c.append(l[0])
                else:
                    c.insert(i+num, l[i+num])
        break
    if num == 2:
        for i in range(len(l)):
            if i == len(l) - 1 or i == len(l) - 2:
                c.append(l[0])
                c.append(l[1])
                break
            else:
                c.insert(i+num, l[i+num])
    # ... supports num up to 15, pattern repeated ...
    if num == 15:
        for i in range(len(l)):
            if (i >= len(l) - 15) or (i >= len(l) - 11):
                c.extend(l[0:15])
                break
            else:
                c.insert(i+num, l[i+num])
    return c
```

## cal(num_of_key: int, message, list_keys: list)

Function: cal
-------------
Encrypts a plaintext message into a large integer using provided keys and ASCII mapping.

Parameters:
num_of_key (int): Number of keys (2 or 3).
message (str): Plaintext message.
list_keys (list): List of numeric/string keys.

Returns:
int: A large integer containing the concatenated encrypted chunks.

```
import math

def cal(num_of_key: int, message, list_keys: list):
    if len(str(list_keys[0])) in (307, 308, 309):
        list_msg = list(message)
        for index, val in enumerate(list_msg):
            for k, v in Ascii2.items():
                if val == k:
                    list_msg[index] = v
        if num_of_key == 2 and len(list_keys) == 2:
            f_p = int(str(list_keys[0])[40:43] + str(list_keys[0])[90:93])
            s_p = int(str(list_keys[1])[100:102] + str(list_keys[1])[200:202])
            an = s_p & f_p
            for index, val in enumerate(list_msg):
                cipher = pow(f_p, val) + s_p + an
                c = int(str(len(str(cipher))) + str(cipher))
                list_msg[index] = c
        if num_of_key == 3 and len(list_keys) == 3:
            f_p = int(str(list_keys[0])[40:45] + str(list_keys[0])[90:94])
            s_p = int(str(list_keys[1])[100:105] + str(list_keys[1])[210:212] + str(list_keys[1])[240:24
            t_p = int(str(list_keys[2])[110:115] + str(list_keys[2])[220:223] + str(list_keys[2])[250:25
            nu = int(str(list_keys[1])[640:645] + str(list_keys[1])[120:123] + str(list_keys[2])[460:462
            an = s_p & t_p
            xo = an ^ nu
            for index, val in enumerate(list_msg):
                cipher = pow(f_p, val) + xo
                c = int(str(len(str(cipher))) + str(cipher))
                list_msg[index] = c
    else:
        list_msg = list(message)
        for index, val in enumerate(list_msg):
            for k, v in Ascii.items():
                if val == k:
                    list_msg[index] = v
        if num_of_key == 2 and len(list_keys) == 2:
            for index, val in enumerate(list_msg):
                f_p = int(str(list_keys[0])[40:43] + str(list_keys[0])[90:93])
                s_p = int(str(list_keys[1])[100:105] + str(list_keys[1])[500:505])
                an = s_p & f_p
                cipher = pow(f_p, val) + s_p + an
                c = int(str(len(str(cipher))) + str(cipher))
                list_msg[index] = c
        if num_of_key == 3 and len(list_keys) == 3:
            for index, val in enumerate(list_msg):
                f_p = int(str(list_keys[0])[40:45] + str(list_keys[0])[90:94])
                s_p = int(str(list_keys[1])[100:110] + str(list_keys[1])[510:515] + str(list_keys[1])[71
                t_p = int(str(list_keys[2])[110:120] + str(list_keys[2])[520:525] + str(list_keys[2])[72
                nu = int(str(list_keys[1])[640:650] + str(list_keys[1])[120:130] + str(list_keys[2])[660
                an = s_p & t_p
                xo = an ^ nu
                cipher = pow(f_p, val) + xo
                c = int(str(len(str(cipher))) + str(cipher))
                list_msg[index] = c
    big_number = int(''.join(str(n) for n in list_msg))
    return big_number
```

## revrese_cal(num_of_key: int, message, list_keys: list)

Function: revrese_cal
---------------------
Decrypts the integer produced by cal() and reconstructs the original plaintext message.

Parameters:
num_of_key (int): Number of keys (2 or 3).
message (int or str): The ciphertext as integer or string.
list_keys (list): List of keys used for encryption.

Returns:
str: Decrypted plaintext.

```
import math

def revrese_cal(num_of_key: int, message, list_keys: list):
    message = str(message)
    list_msg = []
    while len(message) >= 3:
        collect = int(message[:3])
        chunk = message[3:collect]
```

```
            list_msg.append(chunk)
            h = collect + 3
            message = message[h:]
        if len(str(list_keys[0])) in (307, 308, 309):
            if num_of_key == 2 and len(list_keys) == 2:
                f_p = int(str(list_keys[0])[40:43] + str(list_keys[0])[90:93])
                s_p = int(str(list_keys[1])[100:102] + str(list_keys[1])[200:202])
                an = s_p & f_p
                for ind, val in enumerate(list_msg):
                    f1 = int(val) - int(s_p) - int(an)
                    f2 = math.log(f1, f_p)
                    list_msg[ind] = math.ceil(f2)
                for index, val in enumerate(list_msg):
                    for k, v in Ascii2.items():
                        if val == v:
                            list_msg[index] = str(k)
            if num_of_key == 3 and len(list_keys) == 3:
                f_p = int(str(list_keys[0])[40:45] + str(list_keys[0])[90:94])
                s_p = int(str(list_keys[1])[100:105] + str(list_keys[1])[210:212] + str(list_keys[1])[240:24
                t_p = int(str(list_keys[2])[110:115] + str(list_keys[2])[220:223] + str(list_keys[2])[250:25
                nu = int(str(list_keys[1])[640:645] + str(list_keys[1])[120:123] + str(list_keys[2])[460:462
                an = s_p & t_p
                xo = an ^ nu
                for ind, val in enumerate(list_msg):
                    f1 = int(val) - int(xo)
                    f2 = math.log(f1, f_p)
                    list_msg[ind] = round(f2)
                for index, val in enumerate(list_msg):
                    for k, v in Ascii2.items():
                        if val == v:
                            list_msg[index] = str(k)
        else:
            if num_of_key == 2 and len(list_keys) == 2:
                f_p = int(str(list_keys[0])[40:43] + str(list_keys[0])[90:93])
                s_p = int(str(list_keys[1])[100:105] + str(list_keys[1])[500:505])
                an = s_p & f_p
                for ind, val in enumerate(list_msg):
                    f1 = int(val) - int(s_p) - int(an)
                    f2 = math.log(f1, f_p)
                    list_msg[ind] = round(f2) + 1
                    for index, val in enumerate(list_msg):
                        for k, v in Ascii.items():
                            if val == v:
                                list_msg[index] = str(k)
            if num_of_key == 3 and len(list_keys) == 3:
                f_p = int(str(list_keys[0])[40:45] + str(list_keys[0])[90:94])
                s_p = int(str(list_keys[1])[100:110] + str(list_keys[1])[510:515] + str(list_keys[1])[710:71
                t_p = int(str(list_keys[2])[110:120] + str(list_keys[2])[520:525] + str(list_keys[2])[720:72
                nu = int(str(list_keys[1])[640:650] + str(list_keys[1])[120:130] + str(list_keys[2])[660:670
                an = s_p & t_p
                xo = an ^ nu
                for ind, val in enumerate(list_msg):
                    f1 = int(val) - int(xo)
                    f2 = math.log(f1, f_p)
                    list_msg[ind] = round(f2)
                    for index, val in enumerate(list_msg):
                        for k, v in Ascii.items():
                            if val == v:
                                list_msg[index] = str(k)
        word = ''.join(list_msg)
        return word
```

## sender()/receiver()

Function: sender / receiver

---------------------------

Sender connects to receiver, validates key length, performs key exchange and transmits encrypted

message.
Receiver listens, validates, exchanges keys, and decrypts the message.

Note: To launch the program you'll run: runASENFA.exe — it will open two terminals:
- one terminal runs the sender program
- the other terminal runs the receiver program

```
import socket
import time

def sender(ip="127.0.0.1", port=5000):
    time.sleep(1)
    draw()
    print("Hello To My Small Script ASENFA ...\n")
    len_key = input("Choose Length Of Key: Press 1 for 2048, 2 for 3072:\n").strip()
    try:
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client.connect((ip, port))
        print(f"Sender connected to {ip}:{port}")
    except Exception as e:
        print("Could not connect to receiver:", e)
        return
    client.sendall(len_key.encode())
    print("Sender sent length of keys:", repr(len_key))
    try:
        ack = client.recv(1024).decode().strip()
    except Exception as e:
        print("Error waiting for receiver response:", e)
        client.close()
        return
    if ack == "INVALID":
        print("Receiver rejected length of key. Closing connection.")
        client.close()
        return
    elif ack == "OK":
        print("Receiver accepted length of key. Continuing key exchange...")
    else:
        print("Unexpected response from receiver:", repr(ack))
        client.close()
        return
    exchanged = []
    enc_key, main_key, number_of_keys, number_drop_key = encriptkey_main(int(len_key))
    time.sleep(1)
    client.sendall(enc_key.encode())
    exchanged.append(main_key)
    print("Sender sent key #1")
    time.sleep(1)
    data = client.recv(3072)
    key1, _, _ = decript_key(data.decode())
    exchanged.append(key1)
    print("Sender got key #2:", key1)
    time.sleep(1)
    enc_key2, main_key2, _, _ = encriptkey_main(int(len_key))
    client.sendall(enc_key2.encode())
    exchanged.append(main_key2)
    print("Sender sent key #3")
    if number_of_keys == 4:
        data = client.recv(3072)
        key2, _, _ = decript_key(data.decode())
        exchanged.append(key2)
        print("Sender got key #4:", key2)
    print("\nSender exchanged keys:", exchanged)
    print(f"drop Key {number_drop_key}")
    exchanged[number_drop_key] = 0
    exchanged.remove(0)
    print("Final keys:", exchanged)
    plain_text = input("Enter message to send: ")
    encrypted_text = dicit_math.cal(len(exchanged), plain_text, exchanged)
    client.sendall(str(encrypted_text).encode())
    print("Sender sent encrypted message:", encrypted_text)
    client.close()

def receiver(ip="127.0.0.1", port=5000):
    draw()
    print("Hello To My Small Script ASENFA ...\n")
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((ip, port))
    server.listen(1)
    print(f"Receiver listening on {ip}:{port}")
    try:
        conn, addr = server.accept()
    except Exception as e:
        print("Accept failed:", e)
        server.close()
        return
    print(f"Connected by {addr}")
    exchanged = []
    try:
        data = conn.recv(3072)
    except Exception as e:
        print("Receive failed:", e)
        conn.close()
        server.close()
        return
    if not data:
        print("No data received. Closing.")
        conn.close()
        server.close()
        return
    len_keys = data.decode().strip()
    if len_keys in ("1", "2"):
        print("Received valid length of keys:", len_keys)
        conn.sendall("OK".encode())
    else:
        print("Unsucessful Process For Length Of Key:", repr(len_keys))
        try:
            conn.sendall("INVALID".encode())
        except Exception:
            pass
        conn.close()
        server.close()
        print("Receiver closed connection and stopped listening due to invalid length.")
        return
    try:
        data = conn.recv(3072)
        result, number_of_keys, number_drop_key = decript_key(data.decode())
    except Exception as e:
        print("Error during key exchange:", e)
        conn.close()
        server.close()
        return
    exchanged.append(result)
    print("Receiver got key #1:", result)
    key_one, key1, _, _ = encriptkey_main(int(len_keys))
    conn.sendall(key_one.encode())
    exchanged.append(key1)
    print("Receiver sent key #2")
    data = conn.recv(3072)
    result2, _, _ = decript_key(data.decode())
    exchanged.append(result2)
    print("Receiver got key #3:", result2)
    if number_of_keys == 4:
        key_two, key2, _, _ = encriptkey_main(int(len_keys))
        conn.sendall(key_two.encode())
        exchanged.append(key2)
        print("Receiver sent key #4")
    print("\nReceiver exchanged keys:", exchanged)
    print(f"drop Key {number_drop_key}")
    exchanged[number_drop_key] = 0
```

```
exchanged.remove(0)
print("Final keys:", exchanged)
data = conn.recv(1000000).decode()
encrypted_int = data
decrypted_text = dicit_math.revrese_cal(len(exchanged), encrypted_int, exchanged)
print("Receiver decrypted message:", decrypted_text)
conn.close()
server.close()
```