

ASCII Encoding

In ASCII encoding, each letter is converted to one byte. Look at the following examples:

A = 65 or 0b01000001

B = 66 or 0b01000010

C = 67 or 0b01000011

ABC = 0b01000001 0b01000010 0b01000011

Base64 Encoding Text

Base64, also known as privacy enhanced electronic mail (PEM), is the encoding that converts binary data into a textual format; it can be passed through communication channels where text can be handled in a safe environment. PEM is primarily used in the email encryption process. To use the functions included in the Base64 module, you will need to import the library in your code. Base64 offers a decode and encode module that both accepts input and provides output.

To break ASCII encoding into Base64-encoded text, each sequence of six bits encodes to a single character. The characters used can be seen in the following examples:

A-Z: 0-25

a-z: 26-51

0, 1, 2, ..., 9: 52-61

+, /: 62 and 63

Examine the 24 bits from the previous section:

0b01000001 0b01000010 0b01000011

Break the line into 6-bit groups:

0b010000 010100 001001 000011

When you convert the four groups to decimal, you will see that they are equal to the following:

16 20 9 3

You now convert the numbers to Base64:

Q U J D

Therefore, when you encode “ABC” to Base64, you should end up with QUJD, as shown here:

```
>>> import base64
>>> value = 'ABC'.encode()
>>> print(base64.b64encode(value))
b'QUJD'
```

In the previous example, we used Python to encode three bytes at a time. When performing Base64 encoding, the text is broken down into groups of three. In the event that the text cannot be broken down into groups of three, you will see the padding character, which is shown using the equal sign (=). If the example had four bytes, then the output would look like the following:

```
>>> value = 'ABCD'.encode()
>>> print(base64.b64encode(value))
b'QUJDRA=='

>>> value = 'ABCDE'.encode()
>>> print(base64.b64encode(value))
b'QUJDREU='

>>> value = 'ABCDEF'.encode()
>>> print(base64.b64encode(value))
b'QUJDREVG'
```

The preceding padding uses null bytes, which equals A. The capital A is the first character you have in Base64; it stands for six bits of zero (000000). You can prove that with the following:

```
>>> value1 = 'ABCD'.encode()
>>> value2 = 'ABCD\x00'.encode()
>>> value3 = 'ABCD\x00\x00'.encode()
>>> print(base64.b64encode(value1))
b'QUJDRA=='
>>> print(base64.b64encode(value2))
b'QUJDRAA='
>>> print(base64.b64encode(value3))
b'QUJDRAAA'
```

Once you start evaluating Base64, it may become confusing to tell Base64 and ASCII apart. One of the major differences between the two is the encoding process. When you encode text in ASCII, you first start with a text string, and it is converted into a sequence of bytes. When you encode Base64, you are starting with a sequence of bytes and converting the bytes to text.

Binary Data

We will now examine binary data. The file types on your computer are composed of binary data. ASCII data always begins with a first bit of zero. When you open a file using Python, you can convert binary data to Base64:

```
With open('file.exe','rb') as f:
    data = f.read()
    data.encode()
```

Decoding

In this section, we will take what you have learned using the `encode()` methods and get the inverse using the `decode()` method. Examine the following syntax:

```
>>> # encode ABCD
>>> value1 = 'ABCD'.encode()
>>> value1
b'ABCD'

>>> #decode ABCD
>>> value1.decode()
'ABCD'
```

Once you have the binary values, you can use the `base64` library to encode the values using `b64encode`. The inverse is the `b64decode`:

```
>>> myb64 = base64.b64encode(value1)
>>> myb64
b'QUJDRA=='

>>> my = base64.b64decode(myb64)
>> my
b'ABCD'

>>> print (my.decode())
ABCD
>>>
```

Historical Ciphers

Since the invention of writing, there has been the need for message secrecy. While most of the historical ciphers you will learn about in this chapter have been broken, it is important to understand their encryption scheme so that you have a better understanding of how to break them.

In fact, all historical codes used prior to 1980 have been broken except for the one the Native American code talkers used in World Wars I and II. During World War I, the Choctaw Indian language was used, and during World War II, the Navajo language was used. The US Marine Corps recruited Navajo men to serve as Marine Corps radio operators. Both languages served perfectly as they were unwritten and undecipherable due to their complexities.

Scytale of Sparta

One of the oldest cryptographic tools was the Spartan scytale, which was used to perform a transposition cipher. We will examine transposition ciphers in greater detail both later in this chapter and throughout the book. The scytale consisted of a cylinder with a strip of parchment; the parchment would wrap around the cylinder and then the message would be written lengthwise on the parchment. The key in this case would be the radius of the cylinder itself. If the parchment were wrapped around a cylinder of a different radius, the letters would not align in the same way, making the message unreadable.

Substitution Ciphers

The substitution cipher simply substitutes one letter in the alphabet for another based upon a cryptovariable. The substitution involves shifting positions in the alphabet. This includes the Caesar cipher and ROT-13, which will be covered shortly. Examine the following example:

Plaintext: WE HOLD THESE TRUTHS TO BE SELF-EVIDENT, THAT ALL
MEN ARE CREATED EQUAL.

Ciphertext: ZH KROG WKHVH WUXWKV WR EH VHOI-HYLGHQW, WKDW
DOO PHQ DUH FUHDWHG HTXDO.

The Python syntax to both encrypt and decrypt a substitution cipher is presented next. This example shows the use of ROT-13:

```
key = 'abcdefghijklmnopqrstuvwxyz'

def enc_substitution(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()
```

```
def dec_substitution(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
        except ValueError:
            result += l

    return result

origtext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_substitution(13, origtext)
plaintext = dec_substitution(13, ciphertext)

print(origtext)
print(ciphertext)
print(plaintext)
```

Caesar Cipher

The Caesar cipher is one of the oldest recorded ciphers. De Vita Caesarum, Divus Iulius ("The Lives of the Caesars, the Deified Julius), commonly known as The Twelve Caesars, was written in approximately 121 CE. In The Twelve Caesars, it states that if someone has a message that they want to keep private, they can do so by changing the order of the letters so that the original word cannot be determined. When the recipient of the message receives it, the reader must substitute the letters so that they shift by four positions.

Simply put, the cipher shifted letters of the alphabet three places forward so that the letter A was replaced with the letter D, the letter B was replaced with E, and so on. Once the end of the alphabet was reached, the letters would start over:

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
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

The Caesar cipher is an example of a mono-alphabet substitution. This type of substitution substitutes one character of the ciphertext from a character in plaintext. Other examples that include this type of substitution are Atbash, Affine, and the ROT-13 cipher. There are many flaws with this type of cipher, the most obvious of which is that the encryption and decryption methods are fixed and require no shared key. This would allow anyone who knew this method to read Caesar's encrypted messages with ease. Over the years, there have been several variations that include ROT-13, which shifts the letters 13

places instead of 3. We will explore how to encrypt and decrypt Caesar cipher and ROT-13 codes using Python.

For example, given that x is the current letter of the alphabet, the Caesar cipher function adds three for encryption and subtracts three for decryption. While this could be a variable shift, let's start with the original shift of 3:

```
Enc(x) = (x + 3) % 26
Dec(x) = (x - 3) % 26
```

These functions are the first use of modular arithmetic; there are other ways to get the same result, but this is the cleanest and fastest method. The encryption formula adds 3 to the numeric value of the number. If the value exceeds 26, which is the final position of Z, then the modular arithmetic wraps the value back to the beginning of the alphabet. While it is possible to get the ordinal (ord) of a number and convert it back to ASCII, the use of the key simplifies the alphabet indexing. You will learn how to use the `ord()` function when we explore the Vigenère cipher in the next section. In the following Python recipe, the `enc_caesar` function will access a variable index to encrypt the plaintext that is passed in.

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_caesar(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()

plaintext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_caesar(3, plaintext)
print (ciphertext)
```

The output of this should result in the following:

```
zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh fuhdwhg httxdo.
```

The reverse in this case is straightforward. Instead of adding, we subtract. The decryption would look like the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def dec_caesar(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
```

```
i = (key.index(l) - n) % 26
result += key[i]
except ValueError:
    result += l
return result

ciphertext = 'zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh
fuhdwhg htndo.'
plaintext = dec_caesar(3, ciphertext)
print (plaintext)
```

ROT-13

Now that you understand the Caesar cipher, take a look at the ROT-13 cipher. The unique construction of the ROT-13 cipher allows you to encrypt and decrypt using the same method. The reason for this is that since ROT-13 moves the letter of the alphabet exactly halfway, when you run the process again, the letter goes back to its original value.

To see the code behind the cipher, take a look at the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_dec_ROT13(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()

plaintext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_dec_ROT13(13, plaintext)
print (ciphertext)

plaintext = enc_dec_ROT13(13, ciphertext)
print (plaintext)

jr ubyq gurfr gehguf gb or frys-rivrag, gung nyy zra ner pernrgq rdhny.
we hold these truths to be self-evident, that all men are created equal.
```

Whether we use a Caesar cipher or the ROT-13 variation, brute-forcing an attack would take at most 25 tries, and we could easily decipher the plaintext results when we see a language we understand. This will get more complex as we explore the other historical ciphers; the cryptanalysis requires frequency analysis and language detectors. We will focus on these concepts in upcoming chapters.

Atbash Cipher

The Atbash cipher is one of many substitution ciphers you will explore. Similar to ROT-13, the Atbash cipher is also its own inverse, which means you can encode and decode using the same key; this also means we need to have only one function to perform both the encryption and decryption processes. The original cipher was used to encode the Hebrew alphabets but, in reality, it can be modified to encode or decode any alphabet. The Atbash cipher is often thought to be a special case of the Alphine cipher that we will be exploring next.

The following is the Atbash key as it maps to the English alphabet:

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
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

The Python code that implements the Atbash cipher is as follows:

```
def toAtBash(text):
    characters = list(text.upper())
    result = ""
    for character in characters:
        if character in code_dictionary:
            result += code_dictionary.get(character)
        else:
            result += character # preserve non-alpha chars found
    return result

alphabet = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
reverse_alphabet = list(reversed(alphabet))
code_dictionary = dict(zip(alphabet, reverse_alphabet))
plainText= "we hold these truths to be self-evident"
print(plainText)
cipherText = toAtBash(plainText)
print(cipherText)
cipherText = toAtBash(cipherText)
print(cipherText)

we hold these truths to be self-evident
DV SLOW GSVHV GIFGSH GL YV HVOU-VERWVMG
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT
```

Vigenère Cipher

The Vigenère cipher consists of using several Caesar ciphers in sequence with different shift values. To encipher, a table of alphabets can be used, termed a tabula recta, Vigenère square, or Vigenère table. It consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left

compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

Here's an example:

Keyword: **DECLARATION**

D	E	C	L	A	R	A	T	I	O	N
3	4	2	11	0	17	0	19	8	14	13

Plaintext: We hold these truths to be self-evident, that all men are created equal.

Ciphertext: zi jzlu tamgr wvwehj th js fhph-pvzdxvh, gkev llc mxv oeh gtpakew mehdp.

To create a numeric key such as the one shown, use the following syntax. You should see the output [3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:

```
def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i]) - 65
        keyArray.append(keyElement)
    return keyArray

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
print(key)
```

Once you have created the key, you can use it to create ciphertext. You should see the output dpnemvnmifrwgtpakewbsdxe:

```
def shiftEnc(c, n):
    return chr(((ord(c) - ord('A') + n) % 26) + ord('a'))

def enc_vigenere(plaintext, key):
    secret = "".join([shiftEnc(plaintext[i], key[i % len(key)]) for i
in range(len(plaintext))])
    return secret

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
plaintext = 'ALL MEN ARE CREATED EQUAL'
ciphertext = enc_vigenere(plaintext, key)
print(ciphertext)
```

When you know the key, such as in this case, you can decrypt the Vigenère cipher with the following:

```
def shiftDec(c, n):
    c = c.upper()
    return chr((ord(c) - ord('A') - n) % 26 + ord('a'))

def dec_vigenere(ciphertext, key):
    plain = "".join([shiftDec(ciphertext[i], key[i % len(key)]) for i in
range(len(ciphertext))])
    return plain

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
decoded = dec_vigenere(ciphertext, key)
```

We will examine how to brute-force the Vigenère cipher in a later chapter. We will do this by creating a random key that will use the same encryption function, and then we will use frequency analysis to help find the appropriate key. For now, it is more important to understand how the Python code works with this cryptography scheme.

Playfair

The Playfair cipher was used by the Allied forces in World War II; it is the most common digraphic system, and was named after Lord Playfair of England. With this scheme, the sender and the receiver use a shared keyword. The keyword is then used to construct a table that consists of five rows and five columns; the shared word is then populated into the table followed by the rest of the alphabet. As the table is being built out, letters that already appear in the key are skipped. In addition, the letters I and J use the same letter. For this example, we will use the word DECLARATION, as shown here:

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

The Playfair table is read by looking at where the two letters of the blocks intersect. For example, if the first block, TH, were made into a rectangle, the letters at the other two corners of the rectangle would be OF. To see a graphical interpretation of this, examine the next table:

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

When you have two letters that fall in the same column such as WE, you will need to incorporate the next lower letter, and wrap to the top of the column if necessary. This would form a block around the letters ET. The block WE would be encrypted as ET. The same rule applies if letters fall in the same row.

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

Using the preceding table, examine the following plaintext and ciphertext:

Plaintext: He has obstructed the Administration of Justice by refusing his Assent to Laws for establishing Judiciary Powers

Ciphertext: flklyhhmitqafterfldeqrropondionrthazpogiawhvvntpmuorkxgouylulpriwnnyadypwkntwapodkcoysorkxazcrigdnzystem

To create a function in Python that encrypts plaintext using Playfair, type the following:

```
def Playfair_box_shift(i1, i2):
    r1 = i1/5
    r2 = i2/5
    c1 = i1 % 5
    c2 = i2 % 5
    out_r1 = r1
    out_c1 = c2
    out_r2 = r2
    out_c2 = c1
    if r1 == r2:
        out_c1 = (c1 + 1) % 5
        out_c2 = (c2 + 1) % 5
    elif c1 == c2:
        out_r1 = (r1 + 1) % 5
        out_r2 = (r2 + 1) % 5
    return out_r1*5 + out_c1, out_r2*5 + out_c2
```

```

def Playfair_enc(plain):
    random.shuffle(words)
    seed = "".join(words[:10]).replace('j','i')
    alpha = 'abcdefghijklmnopqrstuvwxyz'
    suffix = "".join( sorted( list( set(alpha) - set(seed) ) ) )
    seed_set = set()
    prefix = ""
    for letter in seed:
        if not letter in seed_set:
            seed_set.add(letter)
            prefix += letter
    key = prefix + suffix
    secret = ""
    for i in range(0,len(plain),2):
        chr1 = plain[i]
        chr2 = plain[i+1]
        if chr1 == chr2:
            chr2 = 'X'
        i1 = key.find(chr1.lower())
        i2 = key.find(chr2.lower())
        ci1, ci2 = Playfair_box_shift(i1, i2)
        secret += key[ci1] + key[ci2]
    return secret, key

```

As with the other historical ciphers presented in this chapter, the Playfair cipher can be cracked given enough text. Playfair has a weakness; it will decrypt to the same letter pattern in the plaintext for digraphs that are reciprocals of each other. Examine the next table. Notice how the letters DT and ER form a block. The letters ER (and their reverse RE) form a common digraph in the English language. Digraphs are often used for phonemes that cannot be represented using a single character, like the English *sh* in *ship* and *fish*. When using the English language, there are many words that contain these digraphs, such as *receiver*; notice how receiver contains both an RE and the reciprocal ER; these two letter combinations would encrypt to letter combinations that are easy to identify such as TD and DT. This weakness gives you additional foresight into the cryptographic scheme.

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

Other digraphs in the English language include *sc*, *ng*, *ch*, *ck*, *gh*, *py*, *rh*, *sh*, *ti*, *th*, *wh*, *zh*, *ci*, *wr*, *qu*. Identifying nearby reversed digraphs in the ciphertext and matching the pattern to a list of known plaintext words containing the pattern

is an easy way to generate possible plaintext strings with which to begin constructing the key.

Another way to break the Playfair cipher is with a method called *shotgun hill climbing*. This starts with a random square of letters. Then minor changes are introduced (i.e., switching letters, or rows, or reflecting the entire square) to see if the candidate plaintext is more like standard plaintext than before the change. The minor changes are examined through frequency analysis and language detectors. For now, here is the Python that decrypts the ciphertext using Playfair with a known shared key:

```
def Playfair_box_shift_dec(i1, i2):
    r1 = i1/5
    r2 = i2/5
    c1 = i1 % 5
    c2 = i2 % 5
    out_r1 = r1
    out_c1 = c2
    out_r2 = r2
    out_c2 = c1
    if r1 == r2:
        out_c1 = (c1 - 1) % 5
        out_c2 = (c2 - 1) % 5
    elif c1 == c2:
        out_r1 = (r1 - 1) % 5
        out_r2 = (r2 - 1) % 5
    return out_r1*5 + out_c1, out_r2*5 + out_c2

def Playfair_dec(ciphertext, sharedkey):
    seed = "".join(sharedkey).replace('j','i')
    alpha = 'abcdefghijklmnopqrstuvwxyz'
    suffix = "".join( sorted( list( set(alpha) - set(seed) ) ) )
    seed_set = set()
    prefix = ""
    for letter in seed:
        if not letter in seed_set:
            seed_set.add(letter)
            prefix += letter
    key = prefix + suffix
    plaintext = ""
    for i in range(0,len(ciphertext),2):
        chr1 = ciphertext[i]
        chr2 = ciphertext[i+1]
        print chr1, chr2
        if chr1 == chr2:
            chr2 = 'X'
        i1 = key.find(chr1.lower())
        i2 = key.find(chr2.lower())
        ci1, ci2 = Playfair_box_shift_dec(i1, i2)
        plaintext += key[ci1] + key[ci2]
    return plaintext
```

Hill 2x2

The Hill 2x2 cipher is a polygraphic substitution cipher based on linear algebra. The inventor, Lester S. Hill, created the cipher in 1929. The cipher uses matrices and matrix multiplication to mix the plaintext to produce the ciphertext. To fully understand the Hill cipher, it helps to be familiar with a branch of mathematics known as number theory. The Hill cipher is often covered in depth in many textbooks on the number theory topic. The key used here is HILL, which corresponds to the numbers 7, 8, 11, and 11:

Key: 7, 8, 11, 11

Plaintext: SECRETMESSAG

Ciphertext: CIUBYTMUKGWO

To create a Python function that will create the Hill 2x2 encryption, type the following:

```
import sys
import numpy as np

def cipher_encryption(plain, key):

    # if message length is an odd number, place a zero at the end.
    len_chk = 0
    if len(plain) % 2 != 0:
        plain += "0"
        len_chk = 1

    # msg to matrices
    row = 2
    col = int(len(plain)/2)
    msg2d = np.zeros((row, col), dtype=int)

    itr1 = 0
    itr2 = 0
    for i in range(len(plain)):
        if i%2 == 0:
            msg2d[0][itr1] = int(ord(plain[i]) - 65)
            itr1 += 1
        else:
            msg2d[1][itr2] = int(ord(plain[i]) - 65)
            itr2 += 1

    # key to 2x2
    key2d = np.zeros((2,2), dtype=int)
    itr3 = 0
    for i in range(2):
```

```
        for j in range(2):
            key2d[i][j] = ord(key[itr3]) - 65
            itr3 += 1

    print (key2d)

    # checking validity of the key
    # finding determinant
    deter = key2d[0][0] * key2d[1][1] - key2d[0][1] * key2d[1][0]
    deter = deter % 26

    # finding multiplicative inverse
    for i in range(26):
        temp_inv = deter * i
        if temp_inv % 26 == 1:
            mul_inv = i
            break
        else:
            continue

    if mul_inv == -1:
        print("Invalid key")
        sys.exit()

    encryp_text = ""
    itr_count = int(len(plain)/2)
    if len_chk == 0:
        for i in range(itr_count):
            temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
            encryp_text += chr((temp1 % 26) + 65)
            temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
            encryp_text += chr((temp2 % 26) + 65)
        else:
            for i in range(itr_count-1):
                temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
                encryp_text += chr((temp1 % 26) + 65)
                temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
                encryp_text += chr((temp2 % 26) + 65)

    print("Encrypted text: {}".format(encryp_text))
    return encryp_text

def cipher_decryption(cipher, key):
```

```

# if message length is an odd number, place a zero at the end.
len_chk = 0
if len(cipher) % 2 != 0:
    cipher += "0"
    len_chk = 1

# msg to matrices
row = 2
col = int(len(cipher)/2)
msg2d = np.zeros((row, col), dtype=int)

itr1 = 0
itr2 = 0
for i in range(len(cipher)):
    if i%2 == 0:
        msg2d[0][itr1] = int(ord(cipher[i]) - 65)
        itr1 += 1
    else:
        msg2d[1][itr2] = int(ord(cipher[i]) - 65)
        itr2 += 1

# key to 2x2
key2d = np.zeros((2,2), dtype=int)
itr3 = 0
for i in range(2):
    for j in range(2):
        key2d[i][j] = ord(key[itr3]) - 65
        itr3 += 1

# finding determinant
deter = key2d[0][0] * key2d[1][1] - key2d[0][1] * key2d[1][0]
deter = deter % 26

# finding multiplicative inverse
for i in range(26):
    temp_inv = deter * i
    if temp_inv % 26 == 1:
        mul_inv = i
        break
    else:
        continue

# adjugate matrix
# swapping
key2d[0][0], key2d[1][1] = key2d[1][1], key2d[0][0]

#changing signs
key2d[0][1] *= -1
key2d[1][0] *= -1

```



```
key2d[0][1] = key2d[0][1] % 26
key2d[1][0] = key2d[1][0] % 26

# multiplying multiplicative inverse with adjugate matrix
for i in range(2):
    for j in range(2):
        key2d[i][j] *= mul_inv

# modulo
for i in range(2):
    for j in range(2):
        key2d[i][j] = key2d[i][j] % 26

# cipher to plaintext
decryp_text = ""
itr_count = int(len(cipher)/2)
if len_chk == 0:
    for i in range(itr_count):
        temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
        decryp_text += chr((temp1 % 26) + 65)
        temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
        decryp_text += chr((temp2 % 26) + 65)
    # for
else:
    for i in range(itr_count-1):
        temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
        decryp_text += chr((temp1 % 26) + 65)
        temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
        decryp_text += chr((temp2 % 26) + 65)
    # for
# if else

print("Decrypted text: {}".format(decryp_text))

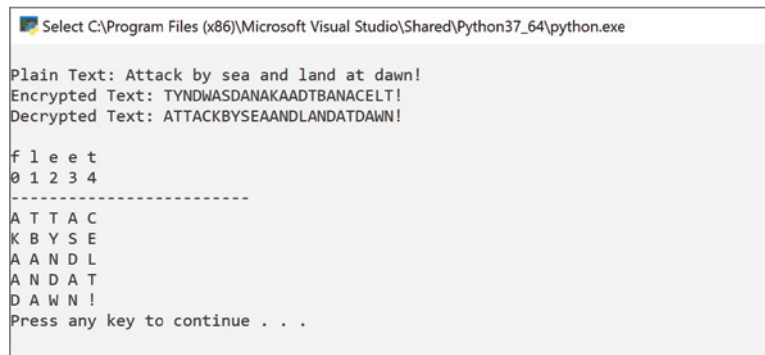
plaintext = "Secret Message"
plaintext = plaintext.upper().replace(" ", "")
key = "hill"
key = key.upper().replace(" ", "")
ciphertext = cipher_encryption(plaintext, key)
cipher_decryption(ciphertext, key)

[[ 7  8]
 [11 11]]
Encrypted text: CIUBYTMUKGWO
Decrypted text: SECRETMESSAG
```

To examine the cryptanalysis of a Hill 2×2 cipher, you attack it by measuring the frequencies of all the digraphs that occur in the ciphertext. In standard English, the most common digraph is *th*, followed by *he*. If you know that the Hill cipher has been employed and the most common digraph is *kx*, followed by *vz*, you would guess that *kx* and *vz* correspond to *th* and *he*, respectively. Once you have a better understanding of frequency analysis, you will revisit this cipher and break it without a key.

Column Transposition

The column transposition cipher is based on a geometric arrangement. While relatively insecure as a standalone cipher, columnar transposition can be a powerful enhancement to other systems. A keyword is chosen as a permutation of *N* columns. The message is then written in a grid with *N* columns. Finally, in the order of the permutation, the columns are taken as the ciphertext. The keyword in this case is 11 characters long. The columns are read off in the letters' numerical order. The order is represented by the numbers under the keyword to give you a visual representation. See Figure 3.1.



```
Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe

Plain Text: Attack by sea and land at dawn!
Encrypted Text: TYNDWASDANAKAADTBANACELT!
Decrypted Text: ATTACKBYSEAANDLANDATDAWN!

f l e e t
0 1 2 3 4
-----
A T T A C
K B Y S E
A A N D L
A N D A T
D A W N !
Press any key to continue . . .
```

Figure 3.1: Column transposition table

The following code reworks the example to be a shorter (it is really long) and removes some debug and fairly self-evident comments in the code. Note that this code works only on Python 3:

```
def cipher_encryption(plain_text, key):

    keyword_num_list = keyword_num_assign(key)
    num_of_rows = int(len(plain_text) / len(key))

    # break message into grid for key
    arr = [[0] * len(key) for i in range(num_of_rows)]
    z = 0
```

```
for i in range(num_of_rows):
    for j in range(len(key)):
        arr[i][j] = plain_text[z]
        z += 1

num_loc = get_number_location(key, keyword_num_list)

cipher_text = ""
k = 0
for i in range(num_of_rows):
    if k == len(key):
        break
    else:
        d = int(num_loc[k])

        for j in range(num_of_rows):
            cipher_text += arr[j][d]
        k += 1
return cipher_text

def get_number_location(key, keyword_num_list):
    num_loc = ""
    for i in range(len(key) + 1):
        for j in range(len(key)):
            if keyword_num_list[j] == i:
                num_loc += str(j)
    return num_loc

def keyword_num_assign(key):
    alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    keyword_num_list = list(range(len(key)))
    init = 0
    for i in range(len(alpha)):
        for j in range(len(key)):
            if alpha[i] == key[j]:
                init += 1
                keyword_num_list[j] = init
    return keyword_num_list

def print_grid(plain_text, key):

    keyword_num_list = keyword_num_assign(key)

    for i in range(len(key)):
        print(key[i], end = " ", flush=True)

    print()
    for i in range(len(key)):
        print(str(keyword_num_list[i]), end=" ", flush=True)
    print()
```

```

print("-----")

# in case characters don't fit the entire grid perfectly.
extra_letters = len(plain_text) % len(key)

dummy_characters = len(key) - extra_letters

if extra_letters != 0:
    for i in range(dummy_characters):
        plain_text += "."

num_of_rows = int(len(plain_text) / len(key))

# Converting message into a grid
arr = [[0] * len(key) for i in range(num_of_rows)]
z = 0

for i in range(num_of_rows):
    for j in range(len(key)):
        arr[i][j] = plain_text[z]
        z += 1

for i in range(num_of_rows):
    for j in range(len(key)):
        print(arr[i][j], end=" ", flush=True)
    print()

def cipher_decryption(encrypted, key):

    keyword_num_list = keyword_num_assign(key)
    num_of_rows = int(len(encrypted) / len(key))

    num_loc = get_number_location(key, keyword_num_list)

    # Converting message into a grid
    arr = [[0] * len(key) for i in range(num_of_rows)]

    # decipher
    plain_text = ""
    k = 0
    itr = 0

    for i in range(len(encrypted)):
        d = 0
        if k == len(key):
            k = 0
        else:
            d: int = int(num_loc[k])
        for j in range(num_of_rows):
            arr[j][d] = encrypted[itr]

```

```

        itr += 1
    if itr == len(encrypted):
        break
    k += 1

print()

for i in range(num_of_rows):
    for j in range(len(key)):
        plain_text += str(arr[i][j])
    return plain_text

plain_text = "Attack by sea and land at dawn!"
key = "fleet"

msg = plain_text.replace(" ", "").upper()
msgkey = key.upper()

encrypted = cipher_encryption(msg, msgkey)
decrypted = cipher_decryption(encrypted, msgkey)

print ("Plain Text: " + plain_text)
print ("Encrypted Text: " + encrypted)
print ("Decrypted Text: " + decrypted)
print ()
print_grid(msg, key)

```

Affine Cipher

Next on our list is the Affine cipher. The Affine cipher is a mono-alphabetic substitution cipher. The difference with the Affine cipher is that each letter is mapped to a numeric equivalent, encrypted using a mathematical function, and then converted back to a letter. While using the mathematical function may sound difficult, the process is fundamentally a standard substitution cipher with a set of rules that govern which letters map to other letters. The mathematical function makes use of the modulo m , where m is the length of the alphabet. Each letter is mapped to a number as shown in the following grid where A = 0, B = 1, . . . , Y = 24, Z = 25:

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
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

The encryption process of the Affine cipher uses modular arithmetic to transform letters into their corresponding integers and then converts the integer into

another number, which in turn is converted back into a letter. The encryption function for a single letter would look like the following:

$$\text{Encrypt}(x) = (ax + b) \bmod m$$

where x is the integer represented by the letter and m is the number of letters in the alphabet.

To decrypt the Affine cipher, you must find the inverse function. The first step is to convert the encrypted letter into an integer, use a mathematical function to convert the number to another number, and then convert the number back into a letter. The decryption process would look like the following:

$$\text{Decrypt}(x) = a^{-1}(x - b) \bmod m$$

where x is the integer represented by the letter, m is the number of letters in the alphabet, and a^{-1} is the modular multiplicative inverse of a modulo m . We will explore the multiplicative inverse of a modulo in more detail in the next chapter. If you find yourself having a hard time understanding the code, feel free to review the next chapter and come back to this cipher.

For now, you will see a working example of the Affine cipher in practice. Let's re-examine the formula for the encryption process: $E(x) = (ax + b)$. You can use the previous table to find the letter in the top row and the corresponding integer in the second row. You are left needing to know what the value is for both a and b . This is the key for the cipher. In this case, we will set the value of $a = 17$ and $b = 20$:

Plaintext	C	O	D	E	B	O	O	K
Value of x	02	14	03	04	01	14	14	10
$ax + b \% 26$	02	24	19	10	11	24	24	8
Encrypted	C	Y	T	K	L	Y	Y	I

Examine the letter E in the table. The letter maps to integer 04. You can use Python to validate the math using the following:

```
>>> print ((17 * 4 + 20) % 26)
10
```

The multiplicative inverse for the Affine cipher is $D(x) = 23 * (x - b) \% 26$. You will learn how it is derived in the next chapter. You should find that you are able to decode the cipher using the following table.

Encrypted	C	Y	T	K	L	Y	Y	I
Encrypted x	02	24	19	10	11	24	24	8
$23 * (x - b) \% 26$	02	14	03	04	01	14	14	10
Plaintext	C	O	D	E	B	O	O	K

To prove the previous encryption and decryption scheme, examine the decryption of the letter K. The letter maps to integer 04. You can use Python to validate the math using the following:

```
>>> print ((23 * (10 - 20) % 26))
4
```

This explanation should give you a bit of insight as you examine the following Python script for implementing the Affine cipher:

```
# Extended Euclidean Algorithm for finding modular inverse
# eg: modinv(7, 26) = 15
def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y

def modinv(a, m):
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None # modular inverse does not exist
    else:
        return x % m

# affine cipher encryption function
# returns the cipher text
def encrypt(text, key):
    '''
    E = (a * x + b) % 26
    '''
    return ''.join([ chr((( key[0]*(ord(t) - ord('A')) + key[1] ) % 26)
                      + ord('A')) for t in text.upper().replace(' ', '') ])

# affine cipher decryption function
# returns original text
def decrypt(cipher, key):
    '''
    D = (a^-1 * (x - b)) % 26
    '''
```

```

        return ''.join([ chr((( modinv(key[0], 26)*(ord(c) - ord('A') -
key[1]))
                        % 26) + ord('A')) for c in cipher ])

# Test the encrypt and decrypt functions
def main():
    # declaring text and key
    text = 'CODEBOOK'
    key = [17, 20]

    # calling encryption function
    encrypted_text = encrypt(text, key)

    print('Encrypted Text: {}'.format(encrypted_text ))

    # calling decryption function
    print('Decrypted Text: {}'.format
(decrypt(encrypted_text, key) ))

if __name__ == '__main__':
    main()

```

Summary

Maintaining password best practices will help mitigate against brute-force attacks on your data. If you are responsible for creating your own authentication system, it is highly recommended that you hash, salt, and/or stretch passwords. As you have learned, just encrypting the data is not enough. In fact, in the next chapter, you will learn how to use Python to crack historical ciphers. We will be building on the mathematical concepts that will enable you to determine the language and encryption scheme of several historical ciphers and determine methods toward their cryptanalysis.