Python

Caesar Cipher in Python (Text encryption tutorial)

Mokhtar Ebrahim Published: April 28, 2020 Last updated: August 14, 2021

Cryptography deals with encrypting or encoding a piece of information (in a plain text) into a form that looks gibberish and makes little sense in ordinary language.
This encoded message(also called **ciphertext**) can then be decoded back into a plain text by the intended recipient using a decoding technique (often along with a private key) communicated to the end-user.

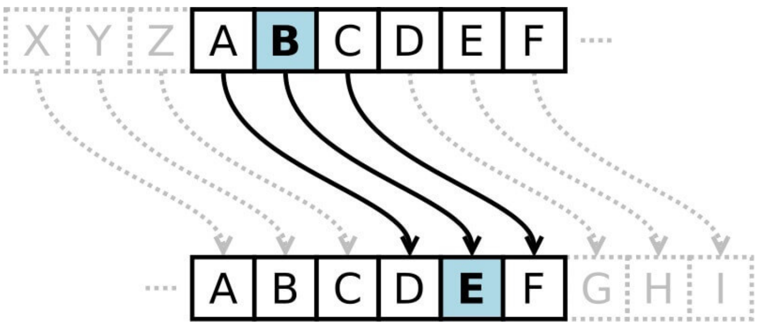
Caesar Cipher is one of the oldest encryption technique that we will focus on in this tutorial, and will implement the same in Python.
Although Caesar Cipher is a **very weak encryption technique** and is rarely used today, we are doing this tutorial to introduce our readers, especially the newcomers, to encryption.
Consider this as the 'Hello World' of Cryptography.

Table of Contents

What is Caesar Cipher?

Caesar Cipher is a type of substitution cipher, in which each letter in the plain text is replaced by another letter at some fixed positions from the current letter in the alphabet.

For example, if we shift each letter by three positions to the right, each of the letters in our plain text will be replaced by a letter at three positions to the right of the letter in the plain text.
Let us see this in action – let's encrypt the text "HELLO WORLD" using a right shift of 3.



So the letter H will be replaced by K, E will be replaced by H, and so on. The final encrypted message for **HELLO WORLD** will be **KHOOR ZRUOG.** That gibberish doesn't make sense, does it?

Note that the letters on edge i.e., X, Y, Z wrap around and are replaced by A, B, C respectively, in case of the right shift. Similarly, the letters in the beginning – A, B, C, etc. will be wrapped around in case of left shifts.

The **Caesar Cipher encryption rule** can be expressed mathematically as:

```
c = (x + n) % 26
```

Where c is the encoded character, x is the actual character, and n is the number of positions we want to shift the character x by. We're taking mod with 26 because there are 26 letters in the English alphabet.

Caesar Cipher in Python

Before we dive into defining the functions for the encryption and decryption process of Caesar Cipher in Python, we'll first look at two important functions that we'll use extensively during the process – **chr()** and **ord()**.
It is important to realize that the alphabet as we know them, is stored differently in a computer's memory. The computer doesn't understand any of our English language's alphabet or other characters by itself.

Each of these characters is represented in computer memory using a number called ASCII code (or its extension – the Unicode) of the character, which is an 8-bit number and encodes almost all the English language's characters, digits, and punctuations.
For instance, the uppercase 'A' is represented by the number 65, 'B' by 66, and so on. Similarly, lowercase characters' representation begins with the number 97.

As the need to incorporate more symbols and characters of other languages arose, the 8 bit was not sufficient, so a new standard – **Unicode** – was adopted, which represents all the characters used in the world using 16 bits.
ASCII is a subset of Unicode, so the ASCII encoding of characters remains the same in Unicode. That means 'A' will still be represented using the number 65 in Unicode.
Note that the special characters like space " ", tabs "\t", newlines "\n", etc. are also represented in memory by their Unicode.

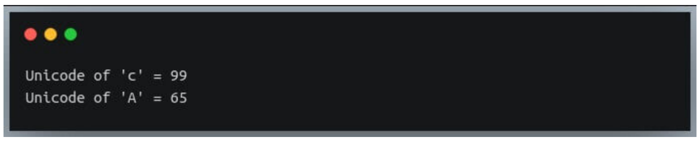
We'll look at two built-in functions in Python that are used to find the Unicode representation of a character and vice-versa.

The ord() function

You can use the ord() method to convert a character to its numeric representation in Unicode. It accepts a single character and returns the number representing its Unicode. Let's look at an example.

```
c_unicode = ord("c")

A_unicode = ord("A")

print("Unicode of 'c' =", c_unicode)

print("Unicode of 'A' =", A_unicode)
```

Output:



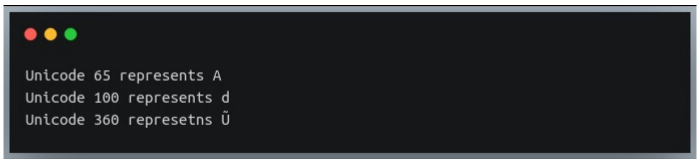
```
Unicode of 'c' = 99
Unicode of 'A' = 65
```

The chr() function

Just like how we could convert a character into its numeric Unicode using ord() method, we do the inverse i.e., find the character represented by a number using chr() method. The chr() method accepts a number representing the Unicode of a character and returns the actual character corresponding to the numeric code.
Let's first look at a few examples:

```
character_65 = chr(65)

character_100 = chr(100)

print("Unicode 65 represents", character_65)

print("Unicode 100 represents", character_100)

character_360 = chr(360)

print("Unicode 360 represents", character_360)
```

Output:



```
Unicode 65 represents A
Unicode 100 represents d
Unicode 360 represetns Ũ
```

Notice how the German letter **Ü** (U umlaut) is also represented in Unicode by the number 360.

We can also apply a chained operation(ord followed by chr) to get the original character back.

```
c = chr(ord("Ũ"))

print(c)
```

Output: Ũ

Encryption for Capital Letters

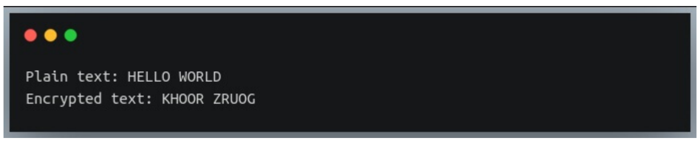
Now that we understand the two fundamental methods we'll use, let's implement the encryption technique for capital letters in Python. We shall encrypt only the uppercase characters in the text and will leave the remaining ones unchanged.
Let's first look at the step-by-step process of encrypting the capital letters:

1. Define the shift value i.e., the number of positions we want to shift from each character.
2. Iterate over each character of the plain text:
    1. If the character is upper-case:
        1. Calculate the position/index of the character in the 0-25 range.
        2. Perform the **positive shift** using the modulo operation.
        3. Find the character at the new position.
        4. Replace the current capital letter by this new character.
    2. Else, If the character is not upper-case, keep it with no change.

Let us now look at the code:

```
shift = 3 # defining the shift count

text = "HELLO WORLD"

encryption = ""

for c in text:

    # check if character is an uppercase letter
    if c.isupper():

        # find the position in 0-25
        c_unicode = ord(c)

        c_index = ord(c) - ord("A")

        # perform the shift
        new_index = (c_index + shift) % 26

        # convert to new character
        new_unicode = new_index + ord("A")

        new_character = chr(new_unicode)

        # append to encrypted string
        encryption = encryption + new_character

    else:

        # since character is not uppercase, leave it as it is
        encryption += c

print("Plain text:",text)

print("Encrypted text:",encryption)
```

Output:



```
Plain text: HELLO WORLD
Encrypted text: KHOOR ZRUOG
```

As we can see, the encrypted text for “HELLO WORLD” is “KHOOR ZRUOG”, and it matches the one we arrived at manually in the Introduction section.
Also, this method doesn’t encrypt the space character, and it continues to be a space in the encrypted version.

Decryption for Capital Letters

Now that we’ve figured out the encryption for plain text capital letters using Ceaser Cipher let’s look at how we will decrypt the ciphertext into plain text.

Earlier, we looked at the mathematic formulation of the encryption process. Let's now check out the same for the decryption process.

```
x = (c - n) % 26
```

The meaning of the notations remains the same as in the previous formula.
If any value becomes negative after subtraction, the modulo operator will take care of that, and it will wrap it around.

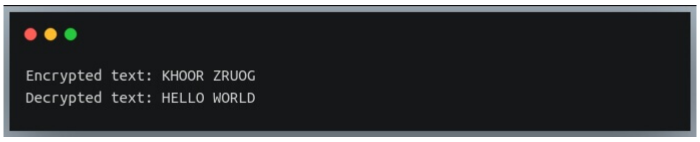
Let us look at the step-by-step implementation of the decryption process, which will be more or less the reverse of the encryption:

- Define the number of shifts
- Iterate over each character in the encrypted text:
  - If the character is an uppercase letter:
    1. Calculate the position/index of the character in the 0-25 range.
    2. Perform the **negative shift** using the modulo operation.
    3. Find the character at the new position.
    4. Replace the current encrypted letter by this new character (which will also be an uppercase letter).
    5. Else, if the character is not capital, keep it unchanged.

Let's write the code for the above procedure:

```
shift = 3 # defining the shift count

encrypted_text = "KHOOR ZRUOG"

plain_text = ""

for c in encrypted_text:

    # check if character is an uppercase letter
    if c.isupper():

        # find the position in 0-25
        c_unicode = ord(c)

        c_index = ord(c) - ord("A")

        # perform the negative shift
        new_index = (c_index - shift) % 26

        # convert to new character
        new_unicode = new_index + ord("A")

        new_character = chr(new_unicode)

        # append to plain string
        plain_text = plain_text + new_character

    else:

        # since character is not uppercase, leave it as it is
        plain_text += c

print("Encrypted text:",encrypted_text)

print("Decrypted text:",plain_text)
```

Output:



```
Encrypted text: KHOOR ZRUOG
Decrypted text: HELLO WORLD
```

Notice how we have successfully recovered the original text "HELLO WORLD" from its encrypted form.

Encrypting numbers and punctuation

Now that we've seen how we can encode and decode capital letters of the English alphabet using Caesar Cipher, it begs an important question – What about the other characters?
What about the numbers? What about the special characters and the punctuation?

Well, the original Caesar Cipher algorithm was not supposed to deal with anything other than the 26 letters of the alphabet – either in uppercase or lowercase.
So a typical Caesar Cipher would not encrypt punctuation or numbers and would convert all the letters to either lowercase or uppercase and encode only those characters.

But we can always extend an existing good solution and tweak them to suit our needs – that's true for any kind of challenge in software engineering.
So we'll try to encode uppercase and lowercase characters the way we did in the previous section, we'll ignore the punctuations for now, and then we'll also encode the numbers in the text.

For numbers, we can do the encryption in one of the two ways:

1. Shift the digit value by the same amount as you shift the letters of the alphabet, i.e., for a shift of 3 – digit 5 becomes 8, 2 becomes 5, 9 becomes 2, and so on.
2. Make the numbers part of the alphabet, i.e., z or Z will be followed by 0,1,2. up to 9, and this time our divider for modulo operation will be 36 instead of 26.

We'll implement our solution using the first strategy. Also, this time, we'll implement our solution as a function that accepts the shift value (which serves as the key in Caesar Cipher) as a parameter.
We'll implement 2 functions – **cipher_encrypt()** and **cipher_decrypt()**
Let's get our hands dirty!

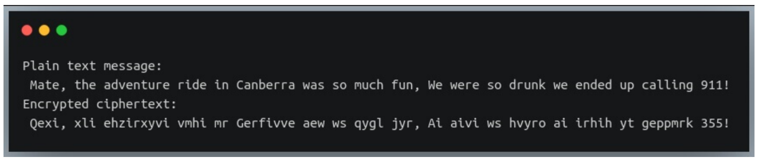
The solution

```
# The Encryption Function
def cipher_encrypt(plain_text, key):

    encrypted = ""

    for c in plain_text:

        if c.isupper(): #check if it's an uppercase character

            c_index = ord(c) - ord('A')

            # shift the current character by key positions
            c_shifted = (c_index + key) % 26 + ord('A')

            c_new = chr(c_shifted)

            encrypted += c_new

        elif c.islower(): #check if its a lowecase character

            # subtract the unicode of 'a' to get index in [0-25) range
            c_index = ord(c) - ord('a')

            c_shifted = (c_index + key) % 26 + ord('a')

            c_new = chr(c_shifted)

            encrypted += c_new
        elif c.isdigit():

            # if it's a number,shift its actual value
            c_new = (int(c) + key) % 10

            encrypted += str(c_new)

        else:

            # if its neither alphabetical nor a number, just leave it like that
            encrypted += c

    return encrypted

# The Decryption Function
def cipher_decrypt(ciphertext, key):

    decrypted = ""

    for c in ciphertext:

        if c.isupper():

            c_index = ord(c) - ord('A')

            # shift the current character to left by key positions to get its original position
            c_og_pos = (c_index - key) % 26 + ord('A')

            c_og = chr(c_og_pos)

            decrypted += c_og

        elif c.islower():

            c_index = ord(c) - ord('a')

            c_og_pos = (c_index - key) % 26 + ord('a')

            c_og = chr(c_og_pos)

            decrypted += c_og

        elif c.isdigit():

            # if it's a number,shift its actual value
            c_og = (int(c) - key) % 10

            decrypted += str(c_og)

        else:

            # if its neither alphabetical nor a number, just leave it like that
            decrypted += c

    return decrypted
```

Now that we've defined our two functions let's first use the encryption function to encrypt a secret message a friend is sharing via text message to his buddy.

```
plain_text = "Mate, the adventure ride in Canberra was so much fun, We were so drunk we ended up calling 911!"

ciphertext = cipher_encrypt(plain_text, 4)

print("Plain text message:\n", plain_text)

print("Encrypted ciphertext:\n", ciphertext)
```

Output:



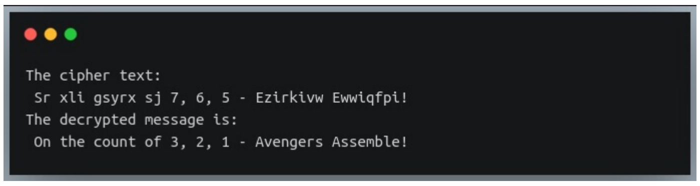
```
Plain text message:
 Mate, the adventure ride in Canberra was so much fun, We were so drunk we ended up calling 911!
Encrypted ciphertext:
 Qexi, xli ehzirxyvi vmhi mr Gerfivve aew ws qygl jyr, Ai aivi ws hvyro ai irhih yt geppmrk 355!
```

Notice how everything except punctuation and spaces has been encrypted.

Now let us look at a ciphertext that Colonel Nick Fury was sending on his pager: **Sr xli gsyrx sj 7, 6, 5 – Ezirkivw Ewwiqfpi!**'
It turns out it's Caesar's ciphertext and fortunately, we got our hands on the key to this ciphertext!
Let's see if we can unearth the hidden message.

```
ciphertext = "Sr xli gsyrx sj 7, 6, 5 - Ezirkivw Ewwiqfpi!"

decrypted_msg = cipher_decrypt(ciphertext, 4)

print("The cipher text:\n", ciphertext)

print("The decrypted message is:\n",decrypted_msg)
```

Output:



```
The cipher text:
 Sr xli gsyrx sj 7, 6, 5 - Ezirkivw Ewwiqfpi!
The decrypted message is:
 On the count of 3, 2, 1 - Avengers Assemble!
```

Way to go, Avengers!

Using a lookup table

At this stage, we have understood the encryption and decryption process of the Caesar Cipher, and have implemented the same in Python.

Now we will look at how it can be made more efficient and more flexible.
Specifically, we'll focus on how we can avoid the repeated computations of the shifted positions for each letter in the text during the encryption and decryption process, by building a **lookup table** ahead of time.

We'll also look at how we can accommodate any set of user-defined symbols and not just the letters of the alphabet in our encryption process.
We'll also merge the encryption and decryption process into one function and will accept as a parameter which of the two processes the user wants to execute.

What is a lookup table?

A lookup table is simply a mapping of the original characters and the characters they should translate to in an encrypted form.
So far, we have been iterating over each of the letters in the string and computing their shifted positions.
This is inefficient because our character set is limited, and most of them occur more than once in the string.
So computing their encrypted equivalence each time they occur is not efficient, and it becomes costly if we are encrypting a very long text with hundreds of thousands of characters in it.

We can avoid this by computing the shifted positions of each of the characters in our character set only once before starting the encryption process.
So if there are 26 uppercase and 26 lowercase letters, we'd need only 52 computations once and some space in memory to store this mapping.
Then during the encryption and decryption process, all we'd have to do is perform a 'lookup' in this table – an operation that is faster than performing a modulo operation each time.

Creating a lookup table

Python's ***string*** module provides an easy way not just to create a lookup table, but also to translate any new string based on this table.

Let's take an example where we want to create a table of the first five lowercase letters and their indices in the alphabet.
We'd then use this table to translate a string where each of the occurrences of 'a', 'b', 'c', 'd' and 'e' are replaced by '0', '1', '2', '3' and '4' respectively; and the remaining characters are untouched.

We will use the **maketrans()** function of the *str* module to create the table.
This method accepts as its first parameter, a string of characters for which translation is needed, and another string parameter of the same length that contains the mapped characters for each character in the first string.

Let's create a table for a simple example.

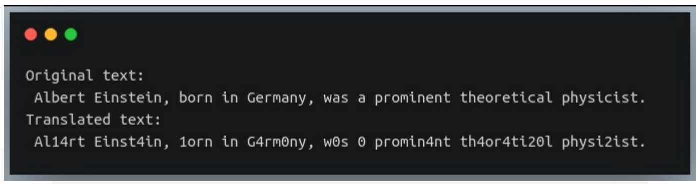
```
table = str.maketrans("abcde", "01234")
```

The table is a Python dictionary that has the characters' Unicode values as keys, and their corresponding mappings as values.
Now that we have our table ready, we can translate strings of any length using this table.
Fortunately, the translation is also handled by another function in the str module, called **translate.**

Let's use this method to convert our text using our table.

```
text = "Albert Einstein, born in Germany, was a prominent theoretical physicist."

translated = text.translate(table)

print("Original text:/n", text)

print("Translated text:/n", translated)
```

Output:



```
Original text:
 Albert Einstein, born in Germany, was a prominent theoretical physicist.
Translated text:
 Al14rt Einst4in, 1orn in G4rm0ny, w0s 0 promin4nt th4or4ti20l physi2ist.
```

As you can see, each instance of the first five lowercase letters have been replaced by their relative indices.

We'll now use the same technique to create a lookup table for Caesar Cipher, based on the key provided.

Implementing the encryption

Let's create a function ***caesar_cipher()*** that accepts a string to be encrypted/decrypted, the 'character set' showing which characters in the string should be encrypted (this will default to lowercase letters),
the key, and a boolean value showing if decryption has performed or otherwise(encryption).

```
import string

def cipher_cipher_using_lookup(text,  key, characters = string.ascii_lowercase, decrypt=False):

    if key < 0:

        print("key cannot be negative")

        return None

    n = len(characters)

    if decrypt==True:

        key = n - key

    table = str.maketrans(characters, characters[key:]+characters[:key])

    translated_text = text.translate(table)

    return translated_text
```

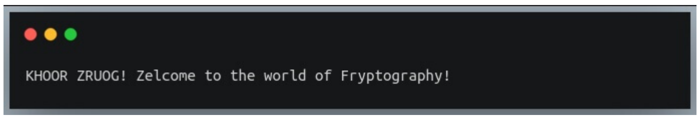
Now that's one powerful function out there!

The whole shifting operation has been reduced to a slicing operation.
Also, we are using *string.ascii_lowercase* attribute – it is a string of characters from 'a' to 'z'.
Another important feature we've achieved here is that the same function achieves both encryption and decryption; this can be done by changing the value of the 'key' parameter.
The slicing operation along with this new key ensures the character set has been left-shifted – something we do in the decryption of a right shift Caesar ciphertext.

Let's validate if this works by using an earlier example.
We'll encrypt only capital letters of the text and will supply the same to the 'characters' parameter.
We'll encrypt the text: "HELLO WORLD! Welcome to the world of Cryptography!"

```
text = "HELLO WORLD! Welcome to the world of Cryptography!"

encrypted = cipher_cipher_using_lookup(text, 3, string.ascii_uppercase, decrypt=False)

print(encrypted)
```

Output:



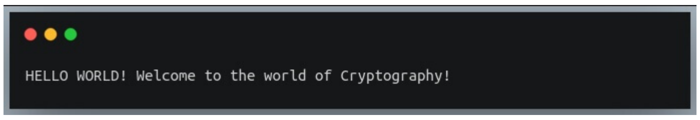
```
KHOOR ZRUOG! Zelcome to the world of Fryptography!
```

Check how the "KHOOR ZRUOG" part matches to encryption of "HELLO WORLD" with key 3 in our first example.
Also, note that we are specifying the character set to be uppercase letters using **string.ascii_uppercase**

We can check if decryption works properly by using the same encrypted text we got in our previous result.
If we can recover our original text back, that means our function works perfectly.

```
text = "KHOOR ZRUOG! Zelcome to the world of Fryptography!"

decrypted = cipher_cipher_using_lookup(text, 3, string.ascii_uppercase, decrypt=True)

print(decrypted)
```

Output:



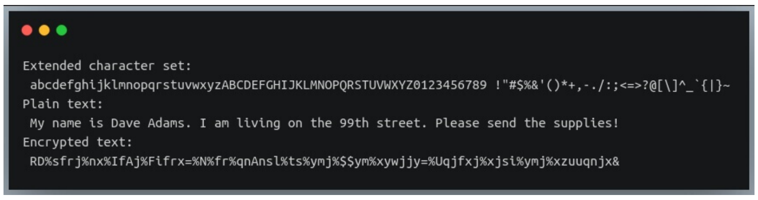
```
HELLO WORLD! Welcome to the world of Cryptography!
```

Notice how we have set the '**decrypt'** parameter in our function to True.
Since we have recovered our original text back, it's a sign our encryption-decryption algorithm using a lookup table is works well!

Let's now see if we can **extend the character set** to include not just lowercase/uppercase characters but also digits and punctuations.

```
character_set = string.ascii_lowercase + string.ascii_uppercase + string.digits + " "+ string.punctuat
ion

print("Extended character set:\n", character_set)

plain_text = "My name is Dave Adams. I am living on the 99th street. Please send the supplies!"

encrypted = cipher_cipher_using_lookup(plain_text, 5, character_set, decrypt=False)

print("Plain text:\n", plain_text)

print("Encrypted text:\n", encrypted)
```

Output:



```
Extended character set:
 abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
Plain text:
 My name is Dave Adams. I am living on the 99th street. Please send the supplies!
Encrypted text:
 RD%sfrj%nx%IfAj%Fifrx=%N%fr%qnAnsl%ts%ymj%$$ym%xywjjy=%Uqjfxj%xjsi%ymj%xzuuqnjx&
```

Here we included all the characters we discussed so far (including space character) in the character set to be encoded.
As a result, everything (even the spaces) in our plain text has been replaced by another symbol!
The only difference here is that the wrap-around doesn't happen individually for lowercase or uppercase characters, but it happens as a whole for the entire character set.
That means 'Y' with a shift of 3 will not become 'B', but will be encoded to '1'.

Negative shift

So far we've been doing 'positive' shifts or 'right shifts' of the characters in the encryption process. And the decryption process for the same involved doing a 'negative' shift or 'left shift' of the characters.
But what if we want to perform the encryption process with a negative shift? Would our encryption-decryption algorithm change?
Yes, it will, but only slightly. The only change we need for a left shift is to make the sign of the key negative, the rest of the process shall remain the same and will achieve the result of a left shift in encryption and a right shift in the decryption process.

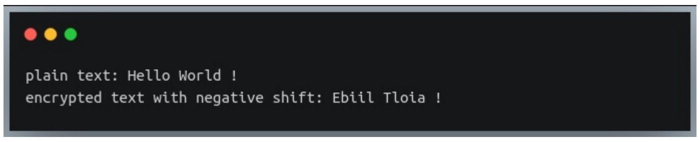
Let us try this by modifying our previous function by adding one more parameter – **'shift_type'** to our function **cipher_cipher_using_lookup()**.

```
import string

def cipher_cipher_using_lookup(text, key, characters = string.ascii_lowercase, decrypt=False, shift
_type="right"):

    if key < 0:

        print("key cannot be negative")

        return None

    n = len(characters)

    if decrypt==True:

        key = n - key

    if shift_type=="left":

        # if left shift is desired, we simply inverse they sign of the key
        key = -key

    table = str.maketrans(characters, characters[key:]+characters[:key])

    translated_text = text.translate(table)

    return translated_text
```

Let us test this modified method on a simple text.

```
text = "Hello World !"

encrypted = cipher_cipher_using_lookup(text, 3, characters = (string.ascii_lowercase + string.ascii
_uppercase), decrypt = False, shift_type="left")

print("plain text:", text)

print("encrypted text with negative shift:",encrypted)
```

Output:

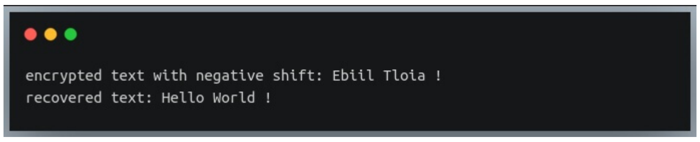


```
plain text: Hello World !
encrypted text with negative shift: Ebiil Tloia !
```

Notice how each of the characters in our plain text has been shifted to the left by three positions.
Let’s now check the decryption process using the same string.

```
text = "Ebiil Tloia !"

decrypted = cipher_cipher_using_lookup(text, 3, characters = (string.ascii_lowercase + string.ascii
_uppercase), decrypt = True, shift_type="left")

print("encrypted text with negative shift:", text)

print("recovered text:",decrypted)
```

Output:



```
encrypted text with negative shift: Ebiil Tloia !
recovered text: Hello World !
```

So we could encrypt and decrypt a text using a lookup table and a negative key.

File encryption

In this section, we’ll look at using Caesar Cipher to encrypt a file.
Note that we can only encrypt plain text files, and not binary files because we know the character set for plain text files.
So, you can encrypt a file using one of the following two approaches:

1. Read the whole file into a string, encrypt the string and dump it into another file.
2. Iteratively read the file one line at a time, encrypt the line, and write it to another text file.

We’ll go with the second approach because the first one is feasible only for small files whose content can fit into memory easily.
So let’s define a function that accepts a file and encrypts it using Caesar Cipher with a right shift of 3. We’ll use the default character set of lower case letters.

```
def fileCipher(fileName, outputFileName, key = 3, shift_type = "right", decrypt=False):

    with open(fileName, "r") as f_in:

        with open(outputFileName, "w") as f_out:

            # iterate over each line in input file
            for line in f_in:

                #encrypt/decrypt the line
                lineNew = cipher_cipher_using_lookup(line, key, decrypt=decrypt, shift_type=shift_type)

                #write the new line to output file
                f_out.write(lineNew)

    print("The file {} has been translated successfully and saved to {}".format(fileName, outputFileName))
```

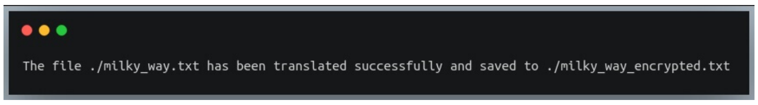
The function accepts the input file name, output file name, and the encryption/decryption parameters we saw in the last section.

Let’s encrypt a file ‘**milky_way.txt**‘ (has the introductory paragraph of the ‘Milky Way’ page on Wikipedia).
We will output the encrypted file to ‘**milky_way_encrypted.txt**‘.

Let’s encrypt it using the function we defined above:

```
inputFile = "./milky_way.txt"

outputFile = "./milky_way_encrypted.txt"

fileCipher(inputFile, outputFile, key=3, shift_type="right", decrypt = False)
```

Output:



```
The file ./milky_way.txt has been translated successfully and saved to ./milky_way_encrypted.txt
```

Let’s check how our encrypted file ‘**milky_way_encrypted.txt**‘ looks like now:

```
Tkh Mlonb Wdb lv wkh jdodab wkdw frqwdlqv rxu Srodu Sbvwhp, zlwk wkh qdph ghvfulelqj wkh jdo
dab'v dsshdudqfh iurp Eduwk: d kdcb edqg ri
oljkw vhhq lq wkh qljkw vnb iruphg iurp vwduv wkdw fdqqrw eh lqglylgxdoob glvwlqjxlvkhg eb wkh
qdnhg hbh.
Tkh whup Mlonb Wdb lv d...
...
...
```

So our function correctly encrypts the file.
As an exercise, you can try the decryption functionality by passing the encrypted file path as an input and setting the ‘decrypt’ parameter to True.
See if you’re able to recover the original text.

Make sure you don’t pass the same file path as both input and output, which would lead to undesired results as the program would do read and write operation on the same file simultaneously.

Multiple shifts (Vigenère Cipher)

So far, we’ve used a single shift value (key) to shift all the characters of the strings by the same no. of positions.
We can also try a variant of this, where we will not use one key, but a sequence of keys to perform different shifts at different positions in the text.

For instance, let us say we use a sequence of 4 keys: [1,5,2,3] With this method, our 1st character in the text will be shifted by a one position, the second character will be shifted by five positions,
the 3rd character by two positions, the 4th character by three positions, and then again the 5th character will be shifted by one position, and so on.
This is an improved version of Caesar Cipher and is called the **Vigenère Cipher.**

Let us implement the Vigenère Cipher.

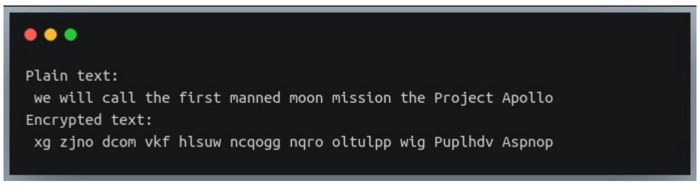
```
def vigenere_cipher(text, keys, decrypt=False):

    # vigenere cipher for lowercase letters
    n = len(keys)

    translatedText =""

    i = 0 #used to record the count of lowercase characters processed so far

    # iterate over each character in the text
    for c in text:

        #translate only if c is lowercase
        if c.islower():

            shift = keys[i%n] #decide which key is to be used

            if decrypt == True:

                # if decryption is to be performed, make the key negative
                shift = -shift

            # Perform the shift operation
            shifted_c = chr((ord(c) - ord('a') + shift)%26 + ord('a'))

            translatedText += shifted_c

            i += 1

        else:

            translatedText += c

    return translatedText
```

The function performs both encryption and decryption, depending on the value of the boolean parameter ‘decrypt’.
We are keeping the count of the total lowercase letters encoded/decoded using the variable i, we use this with modulo operator to determine which key from the list to be used next.
Notice that we have made the shift operation very compact; this is equivalent to the multi-step process of converting between Unicode and character values and computation of the shift we had seen earlier.

Let us test this function using another plain text:

```
text = "we will call the first manned moon mission the Project Apollo"

encrypted_text = vigenere_cipher(text, [1,2,3])

print("Plain text:\n", text)

print("Encrypted text:\n", encrypted_text)
```

Output:



```
Plain text:
 we will call the first manned moon mission the Project Apollo
Encrypted text:
 xg zjno dcom vkf hlsuw ncqogg nqro oltulpp wig Puplhdv Aspnop
```

Here we are performing the encryption using the keys**[1,2,3]** and as expected, the first character ‘w’ has been shifted by one position to ‘x’,
the second character ‘e’ has been shifted by two positions to ‘g’; the third character ‘w’ is shifted by three positions to ‘z’.
This process repeats with subsequent characters.
Go ahead and perform the decryption process with the same keys and see if you can recover the original statement back.

Why Caesar Cipher is weak?

As simple as it is to understand and implement the Caesar Cipher, it makes it easier for anybody to figure out the decryption without a lot of effort.
Caesar Cipher is a substitution cipher technique where we replace each character in the text by some fixed character.

If someone identifies the regularity and pattern in the occurrence of certain characters in a ciphertext, they would quickly identify that Caesar Cipher has been used to encrypt the text.
Once you're convinced that Caesar Cipher technique has been used to encrypt a text, then recovering the original text without the possession of the key is a cakewalk.
A simple BruteForce algorithm figures out the original text in a limited amount of time.

BruteForce Attack

Breaking a ciphertext encoded using Caesar Cipher is only about trying out all the possible keys.
This is feasible because there can only be a limited number of keys that can generate a unique ciphertext.

For instance, if the ciphertext has all the lowercase text encoded, all we have to do is run the decryption step with key values 0 to 25.
Even if the user had supplied a key greater than 25, it would produce a ciphertext that is the same as one of those generated using keys between 0 to 25.

Let's check out a ciphertext that has all its lowercase characters encoded, and see if we can extract a sensible text from it using a BruteForce attack.
The text at our hand is:

```
"ks gvozz ohhoqy hvsa tfca hvs tfcbh oh bccb cb Tisgrom"
```

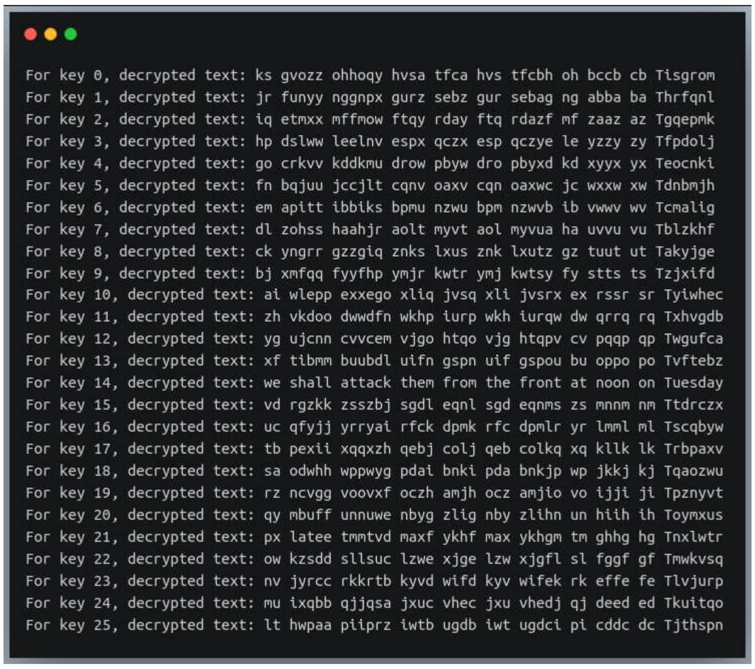
Let's first define the decrypt function that accepts a ciphertext and a key, and decrypts all its lowercase letters.

```
def cipher_decrypt_lower(ciphertext, key):

    decrypted = ""

    for c in ciphertext:

        if c.islower():

            c_index = ord(c) - ord('a')

            c_og_pos = (c_index - key) % 26 + ord('a')

            c_og = chr(c_og_pos)

            decrypted += c_og

        else:

            decrypted += c

    return decrypted
```

Now we have our text, but we don't know the key i.e., the shift value. Let's write a Brute force attack, that tries all the keys from 0 to 25 and displays each of the decrypted strings:

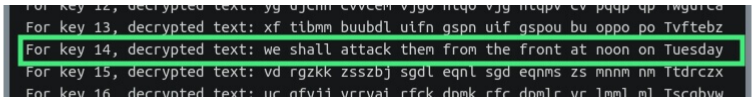
```
cryptic_text = "ks gvozz ohhoqy hvsa tfca hvs tfcbh oh bccb cb Tisgrom"

for i in range(0,26):

    plain_text = cipher_decrypt_lower(cryptic_text, i)

    print("For key {}, decrypted text: {}".format(i, plain_text))
```

Output:



```
For key 0, decrypted text: ks gvozz ohhoqy hvsa tfca hvs tfcbh oh bccb cb Tisgrom
For key 1, decrypted text: jr funyy nggnpx gurz sebz gur sebag ng abba ba Thrfqnl
For key 2, decrypted text: iq etmxx mffmow ftqy rday ftq rdazf mf zaaz az Tgqepmk
For key 3, decrypted text: hp dslww leelnv espx qczx esp qczye le yzzy zy Tfpdolj
For key 4, decrypted text: go crkvv kddkmu drow pbyw dro pbyxd kd xyyx yx Teocnki
For key 5, decrypted text: fn bqjuu jccjlt cqnv oaxv cqn oaxwc jc wxxw xw Tdnbmjh
For key 6, decrypted text: em apitt ibbiks bpmu nzwu bpm nzwvb ib vwwv wv Tcmalig
For key 7, decrypted text: dl zohss haahjr aolt myvt aol myvua ha uvvu vu Tblzkhf
For key 8, decrypted text: ck yngrr gzzgiq znks lxus znk lxutz gz tuut ut Takyjge
For key 9, decrypted text: bj xmfqq fyyfhp ymjr kwtr ymj kwtsy fy stts ts Tzjxifd
For key 10, decrypted text: ai wlepp exxego xliq jvsq xli jvsrx ex rssr sr Tyiwhec
For key 11, decrypted text: zh vkdoo dwwdfn wkhp iurp wkh iurqw dw qrrq rq Txhvgdb
For key 12, decrypted text: yg ujcnn cvvcem vjgo htqo vjg htqpv cv pqqp qp Twgufca
For key 13, decrypted text: xf tibmm buubdl uifn gspn uif gspou bu oppo po Tvftebz
For key 14, decrypted text: we shall attack them from the front at noon on Tuesday
For key 15, decrypted text: vd rgzkk zsszbj sgdl eqnl sgd eqnms zs mnnm nm Ttdrczx
For key 16, decrypted text: uc qfyjj yrryai rfck dpmk rfc dpmlr yr lmml ml Tscqbyw
For key 17, decrypted text: tb pexii xqqxzh qebj colj qeb colkq xq kllk lk Trbpaxv
For key 18, decrypted text: sa odwhh wppwyg pdai bnki pda bnkjp wp jkkj kj Tqaozwu
For key 19, decrypted text: rz ncvgg voovxf oczh amjh ocz amjio vo ijji ji Tpznyvt
For key 20, decrypted text: qy mbuff unnuwe nbyg zlig nby zlihn un hiih ih Toymxus
For key 21, decrypted text: px latee tmmtvd maxf ykhf max ykhgm tm ghhg hg Tnxlwtr
For key 22, decrypted text: ow kzsdd sllsuc lzwe xjge lzw xjgfl sl fggf gf Tmwkvsq
For key 23, decrypted text: nv jyrcc rkkrtb kyvd wifd kyv wifek rk effe fe Tlvjurp
For key 24, decrypted text: mu ixqbb qjjqsa jxuc vhec jxu vhedj qj deed ed Tkuitqo
For key 25, decrypted text: lt hwpaa piiprz iwtb ugdb iwt ugdci pi cddc dc Tjthspn
```

The output lists all the strings you can generate from decryption.
If you look at it closely, the string with key 14 is a valid English statement and hence is the correct choice.



```
For key 13, decrypted text: xf tibmm buubdl uifn gspn uif gspou bu oppo po Tvftebz
For key 14, decrypted text: we shall attack them from the front at noon on Tuesday
For key 15, decrypted text: vd rgzkk zsszbj sgdl eqnl sgd eqnms zs mnnm nm Ttdrczx
```

Now you know how to break a Caesar Cipher encrypted text.
We could use other, stronger variants of Caesar Cipher, like using multiple shifts (Vigenère cipher), but even in those cases, determined attackers can figure out the correct decryption easily.
So the Caesar Cipher algorithm is relatively much weaker than the modern encryption algorithms.

Conclusion

In this tutorial, we learned what Caesar Cipher is, how it is easy to implement it in Python, and how its implementation can be further optimized using what we call 'lookup tables'.
We wrote a Python function to implement a generic Caesar Cipher encryption/decryption algorithm that takes various user inputs as the parameter without assuming much.

We then looked at how we can encrypt a file using Caesar Cipher, and then how Caesar Cipher can be strengthened using multiple shifts.
Finally, we looked at how vulnerable Caesar Cipher to BruteForce attacks.

Share on Facebook Tweet on Twitter

Mokhtar Ebrahim

Mokhtar is the founder of LikeGeeks.com. He works as a Linux system administrator since 2010. He is responsible for maintaining, securing, and troubleshooting Linux servers for multiple clients around the world. He loves writing shell and Python scripts to automate his work.

4 thoughts on "Caesar Cipher in Python (Text encryption tutorial)"

Richard says:
2021-02-20 at 11:53 pm

Very Interesting.

at the python prompt
>>> import this

then have a look at this.py

Reply

Mokhtar Ebrahim says:
2021-02-22 at 8:27 am

Thanks a lot!

Reply

yo its liilz says:
2022-04-26 at 12:46 pm

very good very nice

Reply

Mokhtar Ebrahim says:
2022-04-27 at 8:49 pm

Thanks!

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

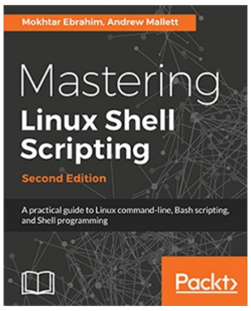
Comment *

Name *

Email

Subscribe

My Published Book


Mokhtar Ebrahim, Andrew Mallett
Mastering Linux Shell Scripting
Second Edition
A practical guide to Linux command-line, Bash scripting, and Shell programming
Packt

Latest Posts

LRU cache in Python (Simple Examples)

Python deque tutorial (Simple Examples)

How to create a Python terminal progress bar using tqdm?

Python Discord bot tutorial

Python print() function tutorial

Quicksort algorithm in Python (Step By Step)

Profiling in Python (Detect CPU & memory bottlenecks)

Python PDF processing
tutorial

Advertisements

Advertisements

NumPy loadtxt tutorial (Load data from files)

20+ examples for NumPy matrix multiplication

Disclaimer Privacy Policy About Contact