Caesar Cipher: Intro to Cyber Security with Encryption & Decryption

Historical Context

- The Caesar Cipher, also known as the Caesar Shift, is an encryption technique used by the famous Roman general & statesman, Julius Caesar to protect messages that were commonly of military and political importance.
- The encryption technique was effective enough that his nephew and successor Octavianus (later renamed to Augustus) used it himself though with a slight variation.
- The Cipher Later evolved into the Vigenère cipher where variations of shifts were done on each letter instead of the same shift on all letters.

The Encryption Algorithm

- The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places, equivalent to a right shift of 23.

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

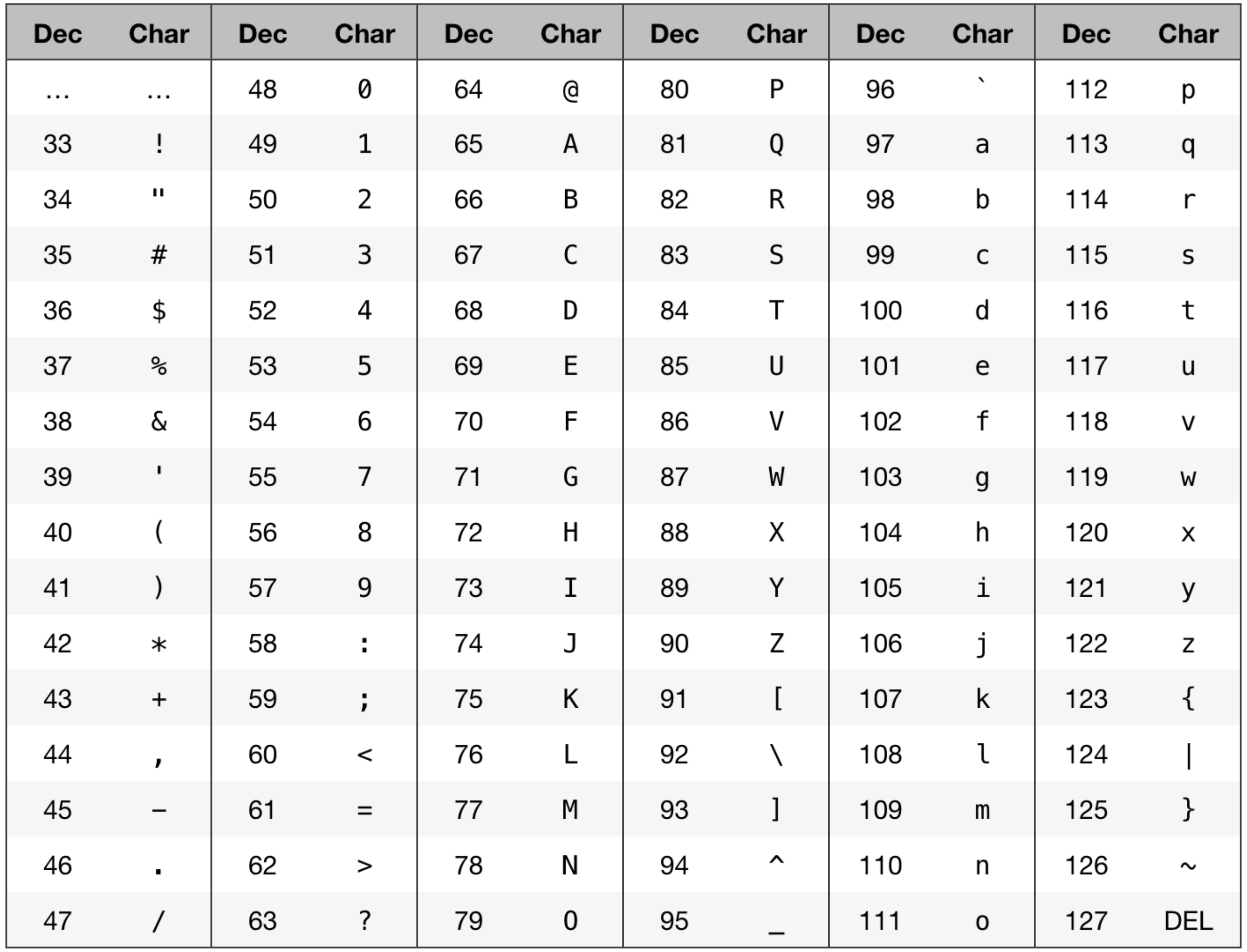
- As a math function, the Caesar Cipher encryption only requires a numerical value representing the letter that is desired to be encrypted (represented as '***x***') and added with the number of rotations desired (represented as '***n***'). The sum result is then modulo'd by the number of letters in the writing system (in the case of the Latin alphabet, there 26 letters so we modulo the sum result by 26).
- $E_n(x) = (x + n)\ mod\ 26$
- **<u>Problem we need to solve – How do we get '*x*'?</u>**

Getting ‘x’: Mapping a letter to a number

- The first step is that we need to decide on how to map our writing system to a numerical value.
- Basically, we need to do this:

  {A, B, C, …, Z} → {0, 1, 2, …, 25}
- What do we know?
  - **The numbers are an incremental sequence.**
  - **All alphabets are sequential.**
- Efficient & Inefficient ways to do this.
  - Inefficient: in this context, using any data structure to map characters to numbers.
  - Decently Efficient: look-up table aka an array of premade values OR a function that returns a number based on the character.
  - Most Efficient: **Subtraction**.
- Subtraction?
  - We refer to the ASCII table which defines the decimal values for each character for computers to print out those characters.
  - Given the property of incremental “sequentiality”, we can take advantage of this through simple arithmetic.
- $x(c) = c - r$
- Where ‘***c***’ is any letter in the alphabet and ‘***r***’ is the first letter in the alphabet of the writing system. ‘A’ represents 65.
- $x(A) = 65 - 65 = 0; \quad x(Z) = 90 - 65 = 25$

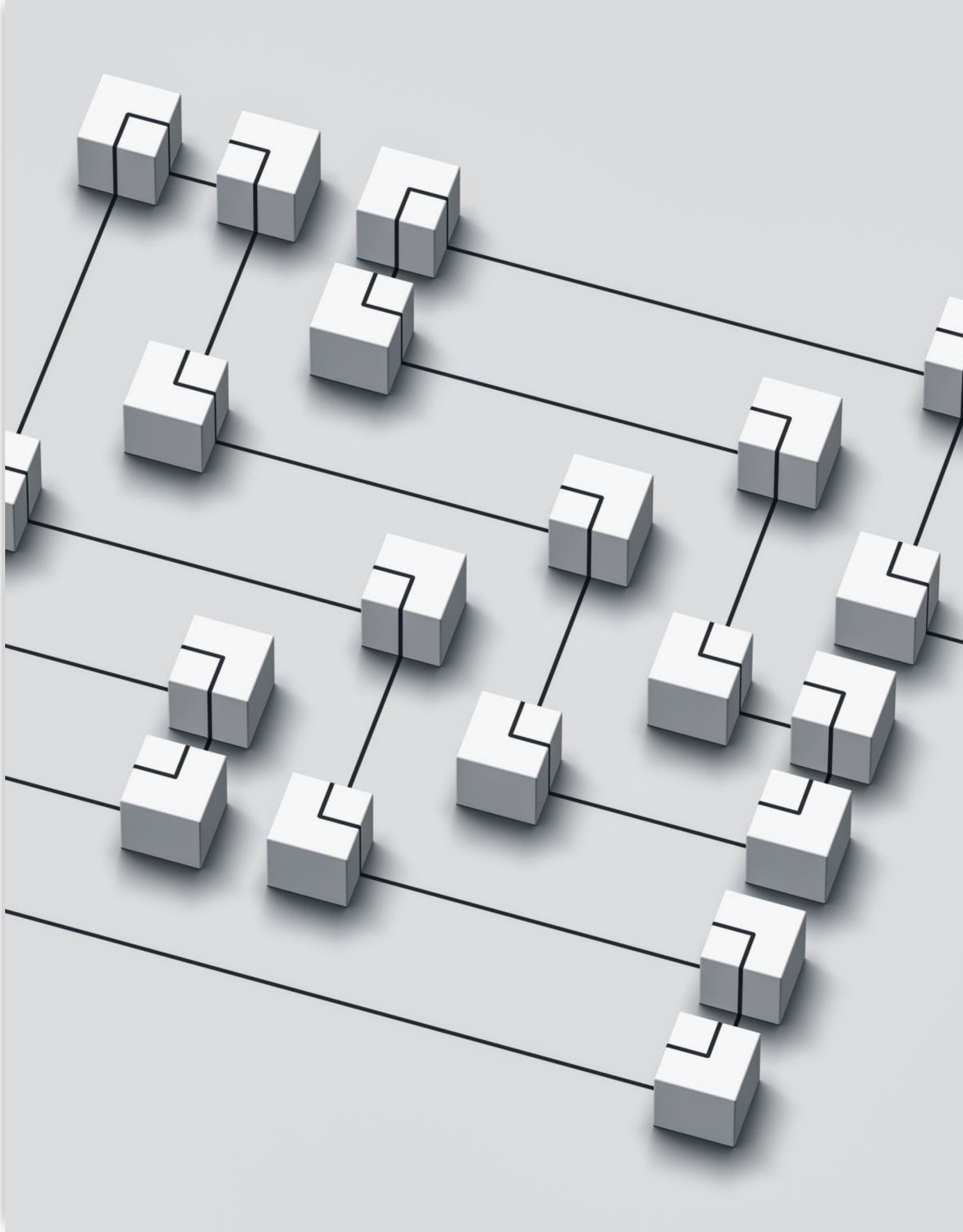
The ASCII Table



Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char
...	...	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Applying the Encryption Algorithm to code

- Now that we have a complete encryption algorithm, it’s time we apply it in code.
- Recall that the C family of programming languages usually have a special type for representing characters (`char`, `rune`, etc.)
- $\forall c \in text \rightarrow E_n(c) = [x(c) + n] \; mod \; 26$
- The math above means: *For-each character ‘c’, subtract it from the first letter in the alphabet that’s used, add it with a rotation, and modulo it by 26.*
- This means we need a for-loop looping over our given text.
- Remember to ignore/skip non-letters!
    - If using C: include ctypes.h and use $isalpha(c)$ which returns bool as an int.
    - If using Java: use $Character.isAlphabetic(c)$ which returns a boolean
- It’s important to remember that the Encryption algorithm does not give you back a letter but the numeric representation of that letter as it’s mapped in the sequence.
- We have to now convert the sequence number to a character representation.
    - Easiest way is by getting the character of the letters we’re using for the cipher by indexing.
    - Most efficient way is by adding the number with the first character in the alphabet.
- The overall inputs/parameters that we need to have is:
    - The text to encrypt.
    - The letters of that alphabet.
    - The number of shifts/rotations desired.



Pseudocode of the Encryption Algorithm and Testing

```
• $CaesarCipherEncrypt(text: string, alphabet: string, shifts: int):$
    • $foreach\ letter\ in\ text:$
        • $seq_{shifted} \leftarrow [(letter - alphabet[0]) + shifts]$
        • $index \leftarrow seq_{shifted} \bmod len(alphabet)$
        • $cipher \leftarrow cipher + alphabet[index]$
    • $return\ cipher$
```

- Test Values:
  - $text = $ "the quick brown fox jumps over the lazy dog“
  - $alphabet = "abcdefghijklmnopqrstuvwxyz“$
  - $shifts = 23$
- Why *that* for the testing text? Because *that* sentence uses every letter in the Latin alphabet.
- **Correct output should be:**
- **<u>“qeb nrfzh yoltk clu grjmp lsbo qeb ixwv ald”</u>**

```
irror_mod = modifier_ob
 mirror object to mirror
irror_mod.mirror_object
operation == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
 operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
 operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

selection at the end -add
_ob.select= 1
er_ob.select=1
ntext.scene.objects.active
("Selected" + str(modifier
irror_ob.select = 0
 bpy.context.selected_obj
ata.objects[one.name].sel

int("please select exactly

- OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
rror X"

context):
ext.active_object is not
```

Problems and Unintended/False operations

- What happens if the ‘shifts’ parameter is a negative number instead of 23?
- We will get the wrong sequence and thus wrong letter to make our cipher!
- $A - A = 0$
- $0 + -3 = -3$
- $-3 \; mod \; 26 = -3 \rightarrow \text{index}$
- Bad, negative numbers do not modulo back to positive number and you can’t index arrays with negative numbers.
- Our sequence doesn’t take negative numbers into account.
- How do we fix this?
    - We need to clamp the value of our resulting sequence number to fit within the bounds of our valid sequence.
    - Quick-fix: Add & Subtract the number of letters in the alphabet to the sequence number result.
    - Better, permanent fix: **Do Summations & Deductions!** ***<u>Requires Looping</u>***
- $-3 + 26 = 23 \rightarrow X$
- Shifting by -3 is the same as shifting by 23! Try it!

Caesar Cipher Decryption Algorithm

- Decryption is considered the opposite of Encrypting.
- But is it the same operation backwards? Usually: **NO**.
- In the case of the Caesar Cipher: **YES**.
- Recall: $E_n(x) = (x + n)\ mod\ 26$
- $D_n(x) = (x - n)\ mod\ 26$
- Like the Encryption algorithm, the Decryption algorithm also relies on the sequence number of the letters.

Caesar Cipher Decryption Algorithm in Pseudocode

```
$CaesarCipherDecrypt(text: string, alphabet: string, shifts: int)$:
    $foreach\ letter\ in\ text$:
        $seq_{shifted} \leftarrow [(letter - alphabet[0]) - shifts]$
        $index \leftarrow seq_{shifted}\ mod\ len(alphabet)$
        $cipher \leftarrow cipher + alphabet[index]$
    $return\ cipher$
```

Notice anything familiar with this pseudocode?

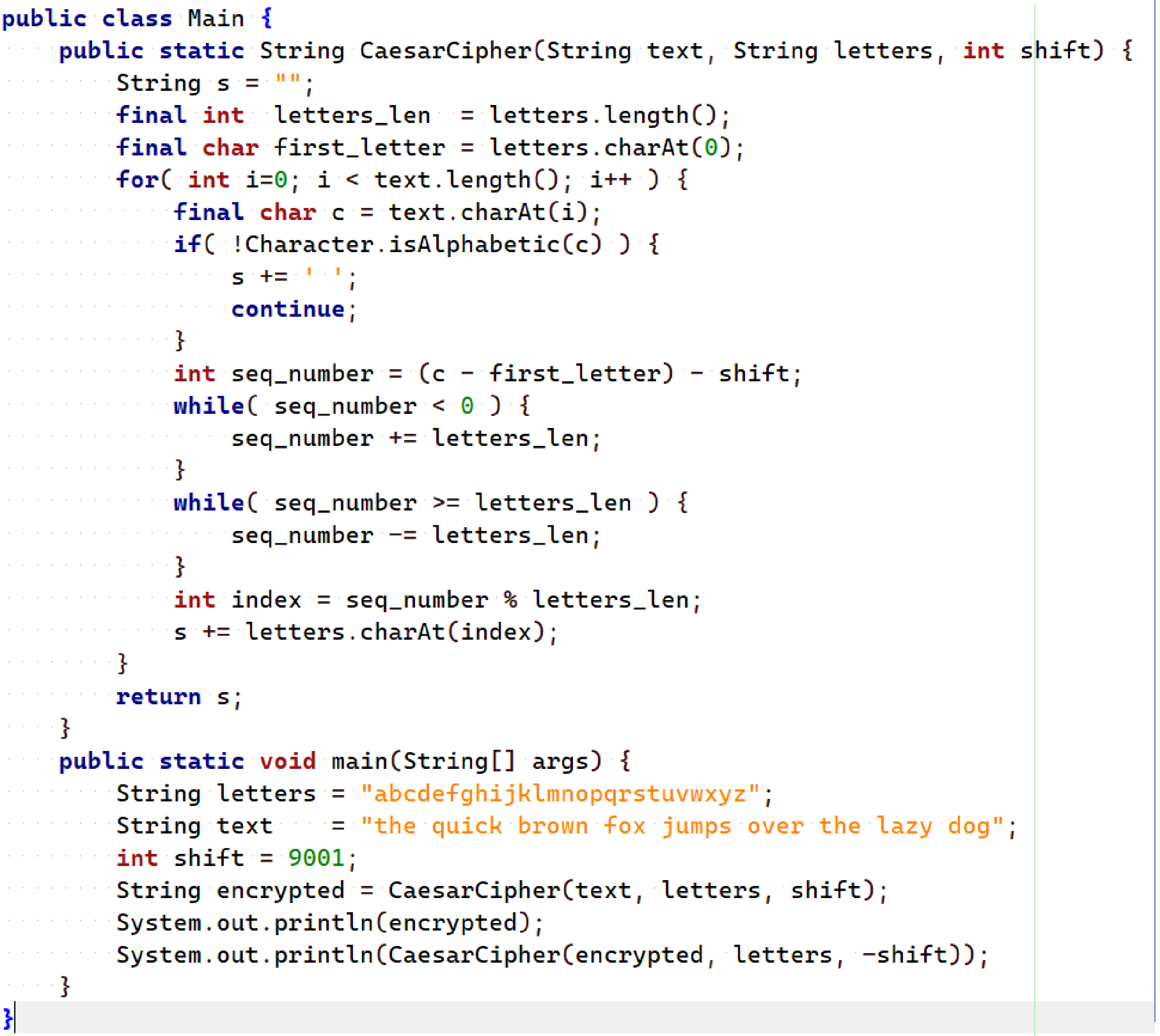
- $CaesarCipherEncrypt(text, alphabet, shifts) = CaesarCipherDecrypt(text, alphabet, -shifts)$
- $CaesarCipherEncrypt(text, alphabet, -shifts) = CaesarCipherDecrypt(text, alphabet, shifts)$

Decryption is **not** always the inverse of Encryption

- For certain encryption algorithms, there are **no** ways to reverse the encrypted data.
- These are called cryptographic hashing functions.
- Some are special "one-way" functions which makes encryption easy to compute but decryption is too expensive to compute.
- Great but also requires saving the encrypted data to compare with later.
- Requires salting and hashing to add further security.
- Professional Scale decryption is much more complicated than this workshop can explain.
- **Keys**: critical component for encrypting & decrypting operations.
  - Values/strings used for encrypting.
  - Without it, decryption is nearly impossible.
  - Example just using math:
    - $\sqrt{4} * key = 2 * 2 \text{ or } -2 * -2?$
    - Key is the missing information to tell us whether our 2's were negative or not.
  - Like a real life key, you need the key to easily unlock an encryption.
  - In the case of the Caesar Cipher, the shift is the key.

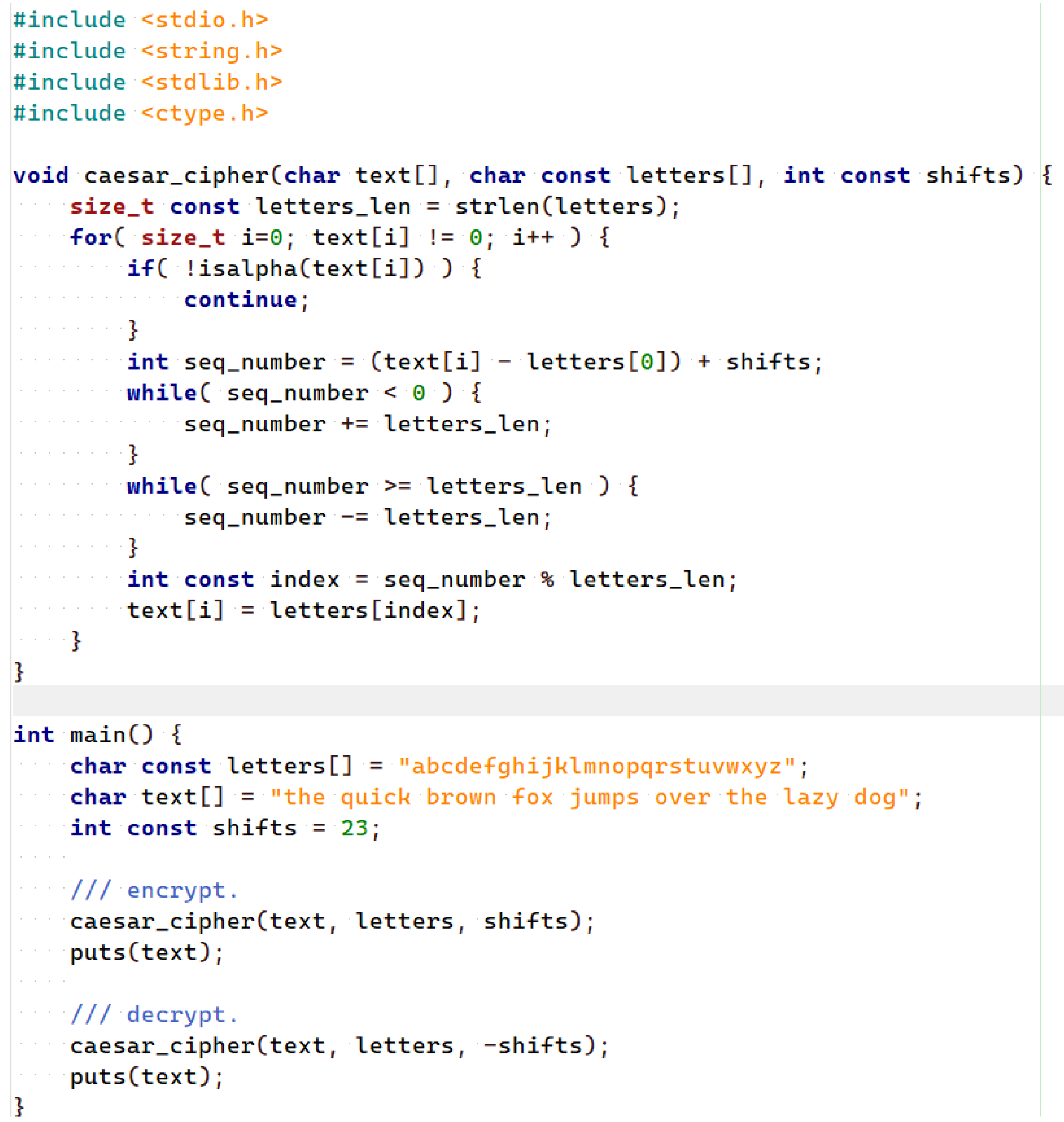
Code Check

Java Implementation



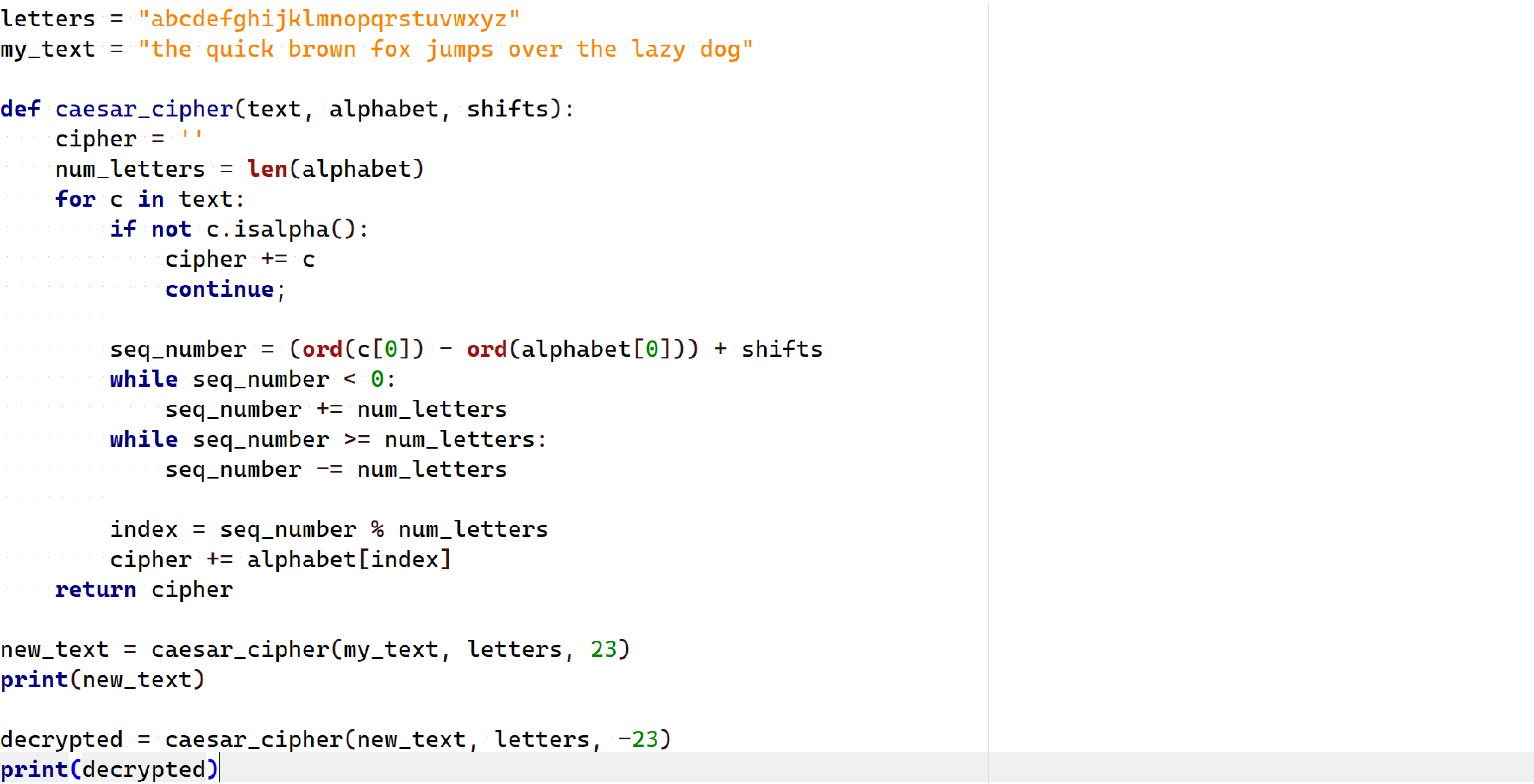
```java
public class Main {
    public static String CaesarCipher(String text, String letters, int shift) {
        String s = "";
        final int  letters_len  = letters.length();
        final char first_letter = letters.charAt(0);
        for( int i=0; i < text.length(); i++ ) {
            final char c = text.charAt(i);
            if( !Character.isAlphabetic(c) ) {
                s += ' ';
                continue;
            }
            int seq_number = (c - first_letter) - shift;
            while( seq_number < 0 ) {
                seq_number += letters_len;
            }
            while( seq_number >= letters_len ) {
                seq_number -= letters_len;
            }
            int index = seq_number % letters_len;
            s += letters.charAt(index);
        }
        return s;
    }
    public static void main(String[] args) {
        String letters = "abcdefghijklmnopqrstuvwxyz";
        String text    = "the quick brown fox jumps over the lazy dog";
        int shift = 9001;
        String encrypted = CaesarCipher(text, letters, shift);
        System.out.println(encrypted);
        System.out.println(CaesarCipher(encrypted, letters, -shift));
    }
}
```

C Implementation



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

void caesar_cipher(char text[], char const letters[], int const shifts) {
    size_t const letters_len = strlen(letters);
    for( size_t i=0; text[i] != 0; i++ ) {
        if( !isalpha(text[i]) ) {
            continue;
        }
        int seq_number = (text[i] - letters[0]) + shifts;
        while( seq_number < 0 ) {
            seq_number += letters_len;
        }
        while( seq_number >= letters_len ) {
            seq_number -= letters_len;
        }
        int const index = seq_number % letters_len;
        text[i] = letters[index];
    }
}

int main() {
    char const letters[] = "abcdefghijklmnopqrstuvwxyz";
    char text[] = "the quick brown fox jumps over the lazy dog";
    int const shifts = 23;

    /// encrypt.
    caesar_cipher(text, letters, shifts);
    puts(text);

    /// decrypt.
    caesar_cipher(text, letters, -shifts);
    puts(text);
}
```

Bonus Code Check: Python



```python
letters = "abcdefghijklmnopqrstuvwxyz"
my_text = "the quick brown fox jumps over the lazy dog"

def caesar_cipher(text, alphabet, shifts):
    cipher = ''
    num_letters = len(alphabet)
    for c in text:
        if not c.isalpha():
            cipher += c
            continue;

        seq_number = (ord(c[0]) - ord(alphabet[0])) + shifts
        while seq_number < 0:
            seq_number += num_letters
        while seq_number >= num_letters:
            seq_number -= num_letters

        index = seq_number % num_letters
        cipher += alphabet[index]
    return cipher

new_text = caesar_cipher(my_text, letters, 23)
print(new_text)

decrypted = caesar_cipher(new_text, letters, -23)
print(decrypted)
```

Conclusion: End of the Caesar Cipher Workshop

What have we learned?

How to use character constants as numbers using the ASCII Table.

How to use those character constants to map & create sequences.

Basic encryption & decryption using arithmetic operations.

Using mathematical & numeric properties to create efficient algorithms.

Extra practice in using loops & their use-cases.

Next week's Engineering Club Coding Workshop (Nov 24th, 2023)

How to build a HashTable/HashMap aka Dictionary from scratch!