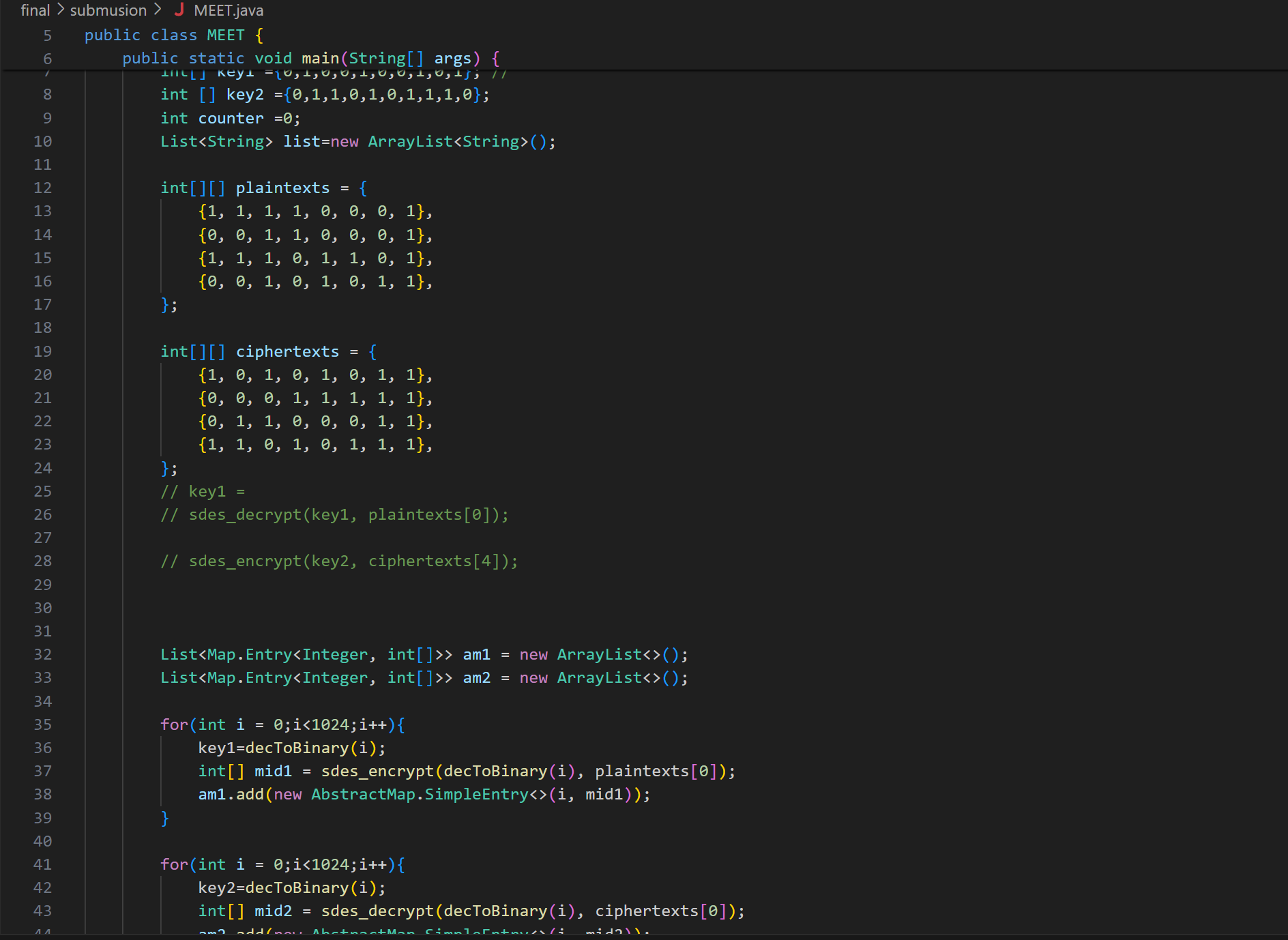


Final report

Part1:

Code:

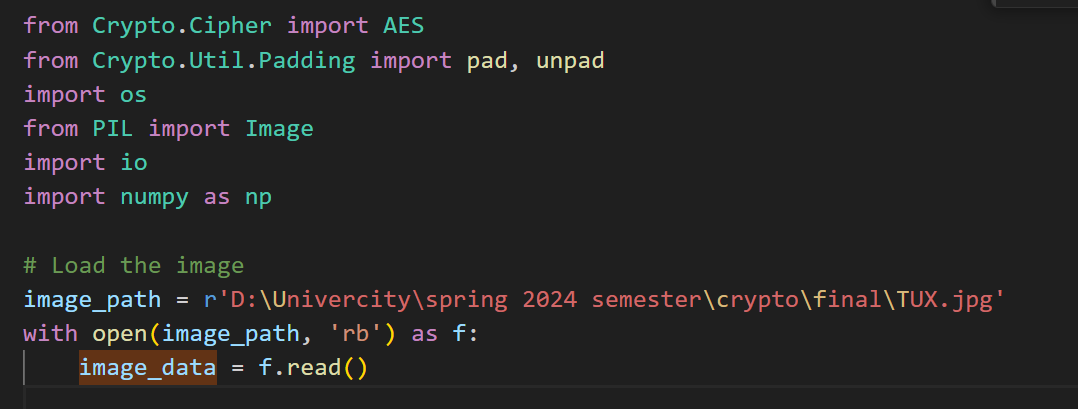


C:\Users\User\Downloads\MicrosoftTeams-image.png

Part2:

1\_ code explanation:

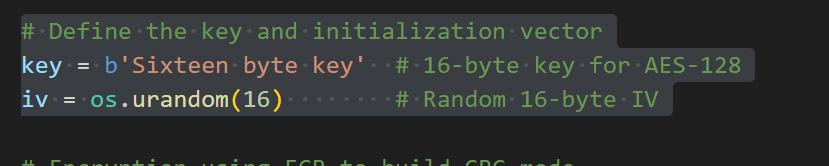
**CBC mode**



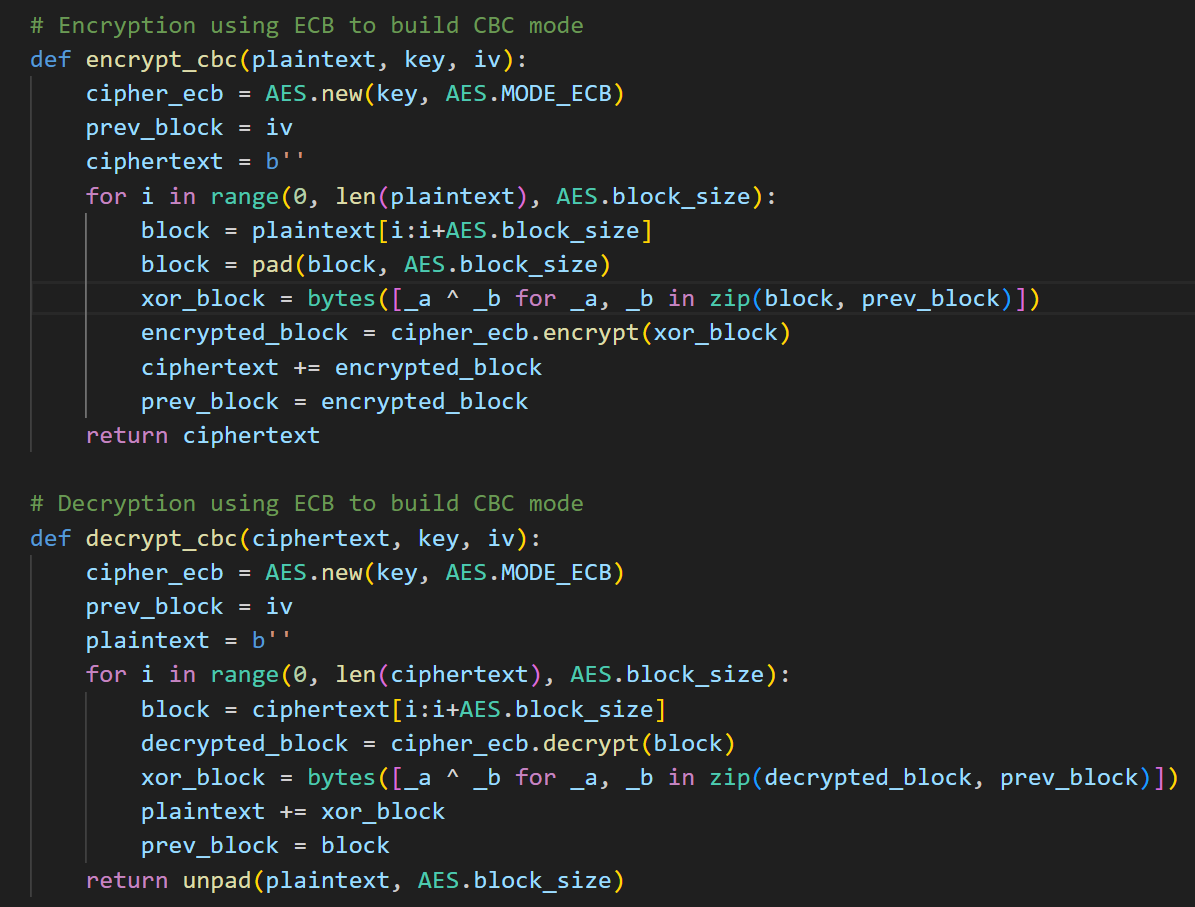
First importing needed libraries.

Load the image by assigning the path of the image and the r stands for the python to treat the "\" as a path.

After that opening the image and reading it as 'rb' with is read binary mode and treat it as object name f after that start reading that object and assigned for image\_data.



After that start to define the IV and key for random 16-bit key with b prefix to indicate byte string The length of the IV in AES-128 encryption is 16 bytes (128 bits), which is produced by the function os.urandom(16).



Encryption:

A function takes the plaintext, key and IV assign cipher text of ECB by calling build in function of AES encryption 128 to use all that to build the CBC mode by passing the key and the mode of encryption.

Assign prev\_block in the first place IV with initializing cipher text to be the final output.

Start a for loop iterated to all the plaintext blocks and the iterated by 16 bytes

After that assigning block checking and splitting the blocks for the 16 byte block also check the block if need padding (\x08) and call the pad function.

xor\_block: Conducts a byte-wise XOR operation between the block in plaintext that is currently being read and the block that was previously encrypted (or the IV for the first block).

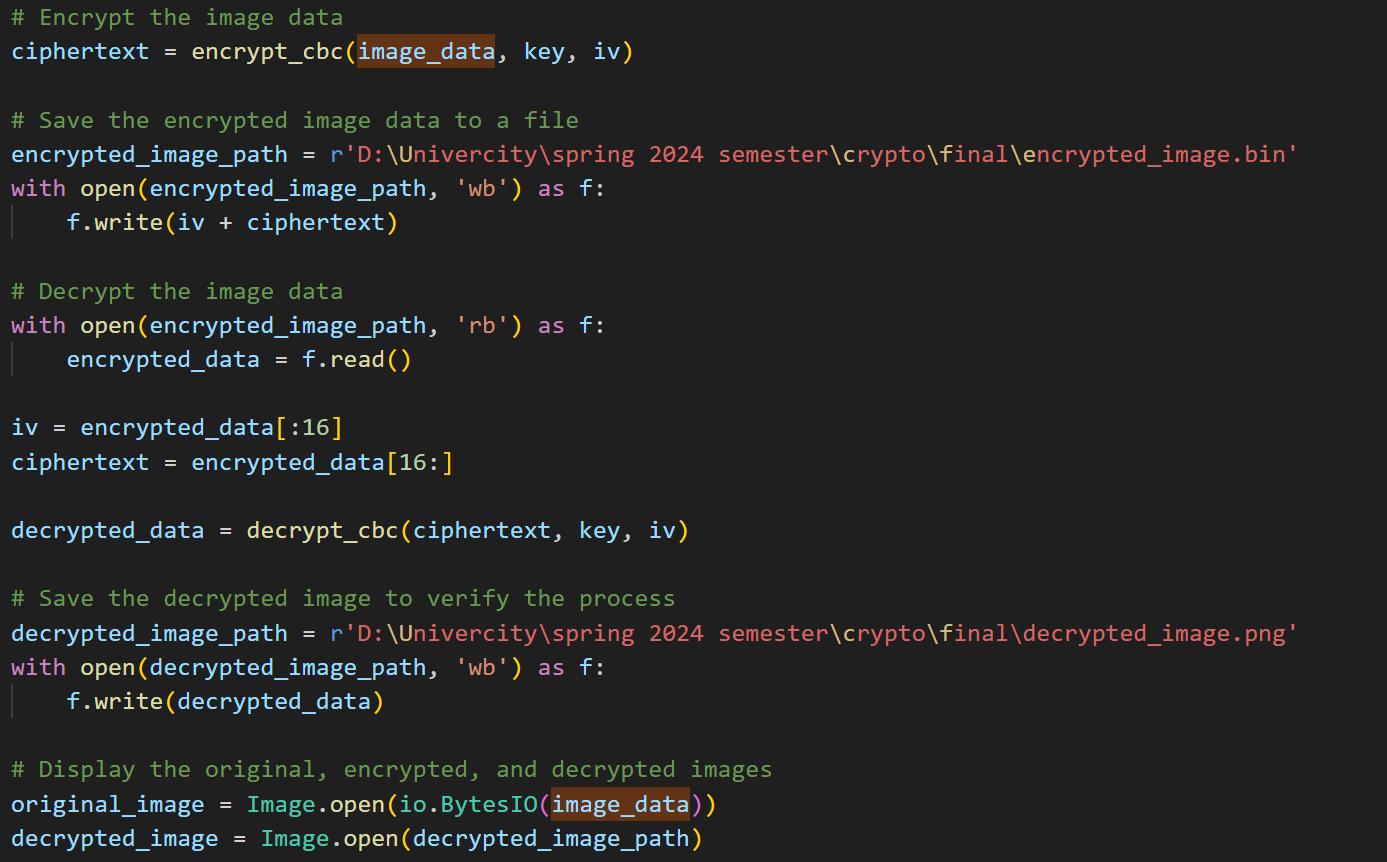
encrypted\_block: Uses the AES cipher in ECB mode to encrypt the XORed block.

Then append the encrypted blocks to the ciphertext.

Then update the value of the prev\_blocks to be the encrypted block. And return the ciphertext.

Decryption:  
the same going here:

setting up an AES encryption in ECB mode.  
Block by block iterating through the ciphertext.  
use AES in ECB mode to decrypt every block.  
XORing the preceding block of ciphertext (or IV for the first block) with the decrypted block.  
adding the block that was XORed to the plaintext.  
the prior block is updated for the upcoming iteration.  
deleting the final plaintext's padding.



We call the function of encryption and assigned to cipheretext.

As the steps of reading we assigned the path of the encryption pic and open it in write binary mode as the same object.

After that write the IV and the cipher text we write the iv to save the value of It for decryption.

In decryption read the file by read binary and assigned the encrypted\_data for the content of the file.

Assign the IV value and slice it at the first 16 bytes after it the ciphertext.

After we got the data so it' ready to be decrypted we call the function of the decryption.

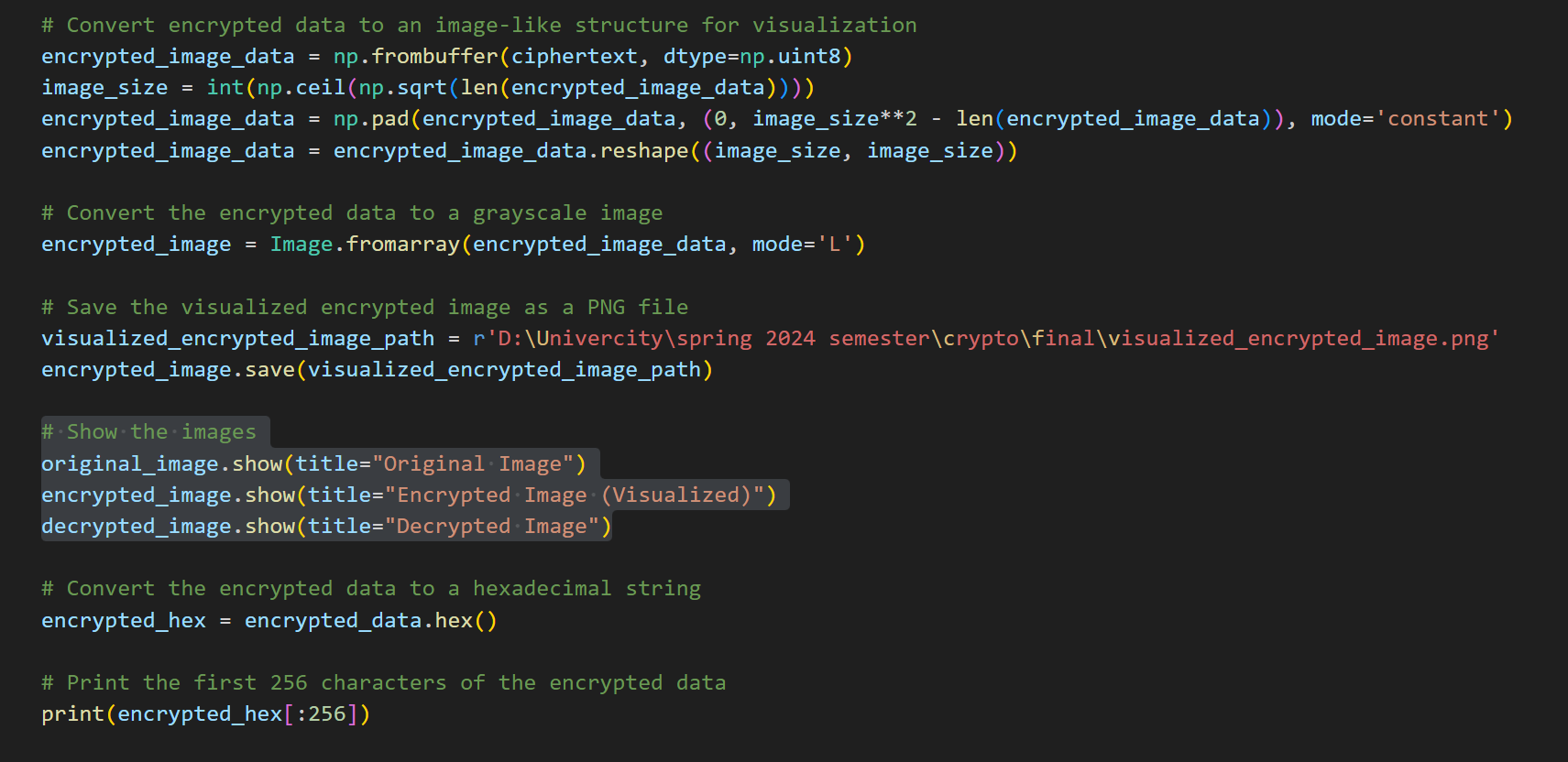
After we decrypt the image save it after that write binary mode.

For the displaying the pics we need the following:

Pillow Library: Processing images requires this. Use pip install Pillow to install it.  
The photos are shown using the matplotlib library. Use pip install matplotlib to install it.

It is expected that the picture data is contained in a byte stream called image\_data.  
IO.Using the image\_data, BytesIO(image\_data) generates an in-memory binary stream.  
The binary stream is opened as an image object by using the Pillow library's Image.open() function.

And for the decrypted image as well.



First part of the previews pic:

To make data manipulation easier, transform the encrypted text into a NumPy array.  
Determine the necessary image size by using the data length's square root and rounding up.

* encrypted\_image\_data: The array with the data that you have encrypted.
* (0, len(encrypted\_image\_data) - image\_size\*\*2):
* (0, X) refers to "add 0 elements at the start and X elements at the end" .
* Image\_size\*\*2 - len(encrypted\_image\_data) is the formula used to compute X.
* Image\_size\*\*2 is the total amount of pixels in the intended image.
* The current number of components in your encrypted data is len(encrypted\_image\_data).
* To fit the picture size, the number of elements required to pad the array is given by image\_size\*\*2 - len(encrypted\_image\_data).
* mode='constant' indicates that zeros (the default constant value) should be the additional elements.
* To make the array a perfect square, add zeros to it.  
  Convert the array back into a 2D matrix that depicts a square picture.

To make the array a perfect square, add zeros to it.  
Convert the array back into a 2D matrix that depicts a square picture.

Second part:

This line creates a PIL Image object (encrypted\_image) from a 2D NumPy array (encrypted\_image\_data) representing grayscale pixel values, with mode='L' indicating grayscale mode for each pixel value.

Third part:

For saving the encrypted pic as png or jpg stander form to be visualized.

Fourth part:

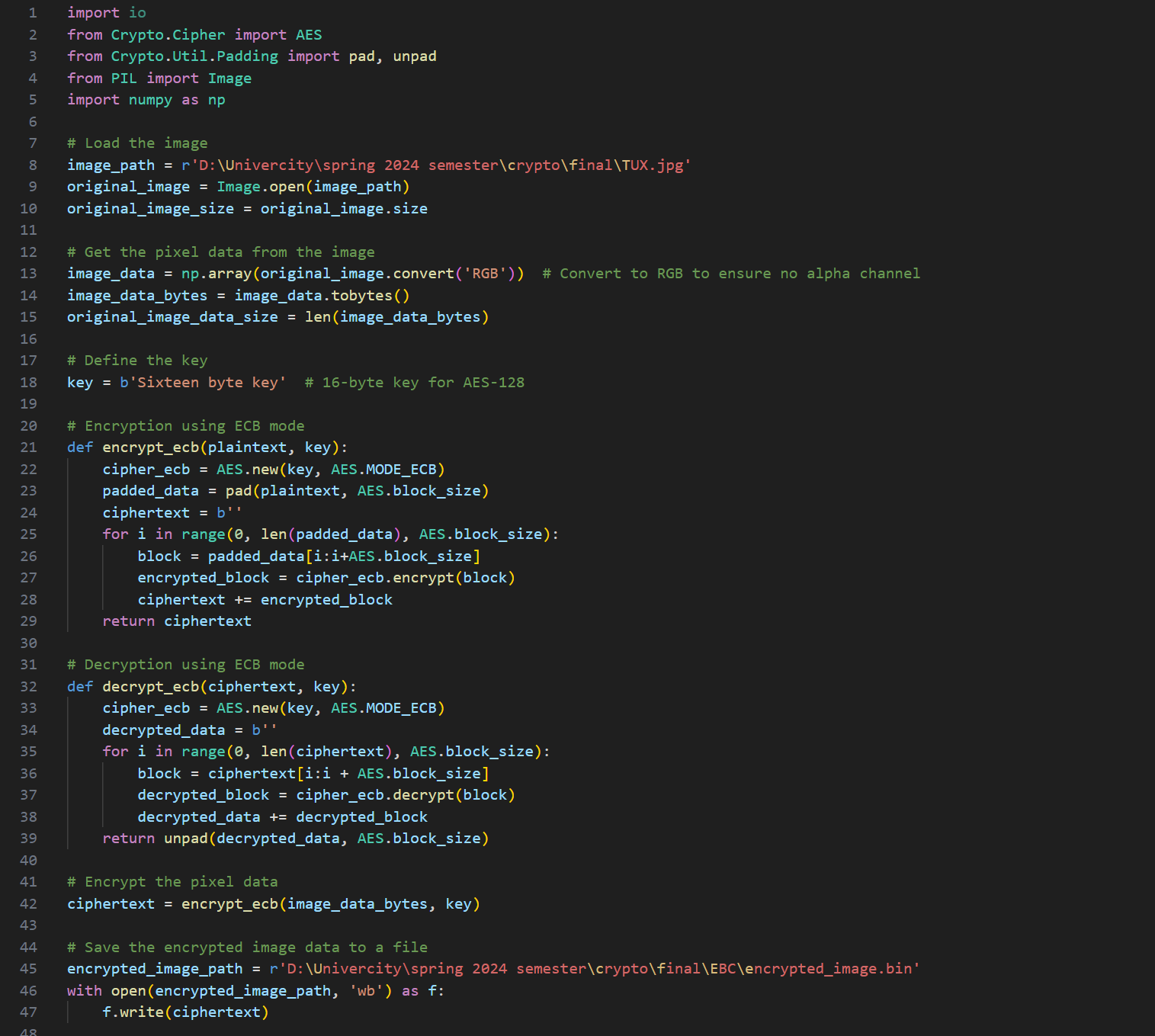
The function displays the original image stored in original\_image using the show method of the PIL Image object, with the title "Original Image" indicating the window or viewer.

Fifth and sixth part:

It's converting the binary data into hexadecimal to print it.

**Note : The code converts encrypted data into a 2D NumPy array and reshapes it into a square matrix, resulting in a grayscale image due to the intensity value**.

**EBC mode:**



Part one:

The code encrypts an image file using the Image.open method from the Pillow library, and stores the original image's size in a variable.

Part two:

The original\_image.convert() method converts an image to RGB mode without an alpha channel, ensuring transparency. The np.array() method converts the image data to a NumPy array.

Image\_data.tobytes() converts NumPy pixel array into bytes, as AES encryption process uses bytes, not structured data like arrays.

The len(image\_data\_bytes) function calculates the total size of the image data in bytes, aiding in operations like reshaping decrypted data back into original image dimensions.

The rest of the code the same as the CBC



Part 1:

Decrypt pixel data.

Part 2:

The function np.frombuffer(...) converts byte data into a NumPy array without copying it, efficiently converting it into a 1-dimensional array. The resulting array is uint8 (unsigned 8-bit integer), commonly used for image data.

The function reshape(image\_data.shape) returns the original shape of the image, based on the dimensions of the original image, which were converted to RGB.

Part 3:

The function Image.fromarray(decrypted\_image\_data, mode='RGB') converts a NumPy array into a PIL Image object, specifying RGB mode with three color channels.

The method saves a decrypted image to a specified file path, specifying the format as JPEG, using the decrypted\_image\_path from the NumPy array.

Part 4:

Show the pics by PIL library for the image.

Part 5:

The function np.frombuffer(ciphertext, dtype=np.uint8) creates a 1-dimensional NumPy array from the encrypted byte data, specifying the uint8 data type for image data.

The encrypted\_image\_data[:image\_data.size] parameter ensures the encrypted data array is truncated to the original image data size, preventing padding from potentially causing the encrypted data to be larger.

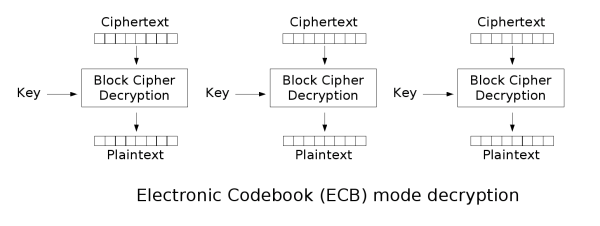
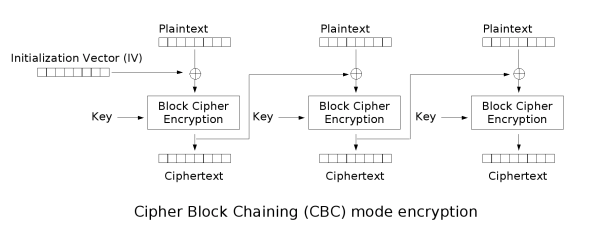
The encrypted\_image\_data.reshape function reshapes the 1-dimensional NumPy array back into its original shape, based on the original image's dimensions (height, width, 3), representing RGB color channels.

Part 6,7,8:

Convert the pic in RGB mode then save it after that show pic.

2\_ the EBC and the CBC differences:

Structure:



Comparison:

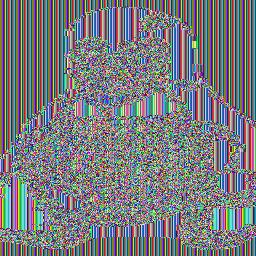
|  |  |  |  |
| --- | --- | --- | --- |
| Feature | ECB Mode | CBC Mode |  |
| Security | Less secure | More secure |  |
| Encryption Speed | Faster (potentially) | Slower |  |
| Decryption Speed | Same as Encryption | Same as Encryption |  |
| Parallelization | Possible | Not ideal |  |
| Padding Required | No | Yes |  |

I pick the CBC mode because it's a secure more than the EBC and when we talk about sending pics I think the speed not matter but security.

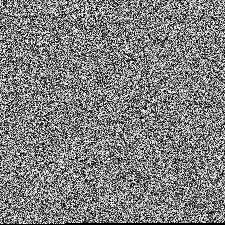
The security point is when attack happened and the key disclosure so the whole EBC mode will Collapses but the CBC is dependency and hard to control it because you need the IV but there's an error with a bit the block that get effected is 1 or 2 blocks in great extent but the EBC has not depend on the others.

By the output you can see that the EBC as same as the plaintext but CBC in not.

The output:



EBC mode output



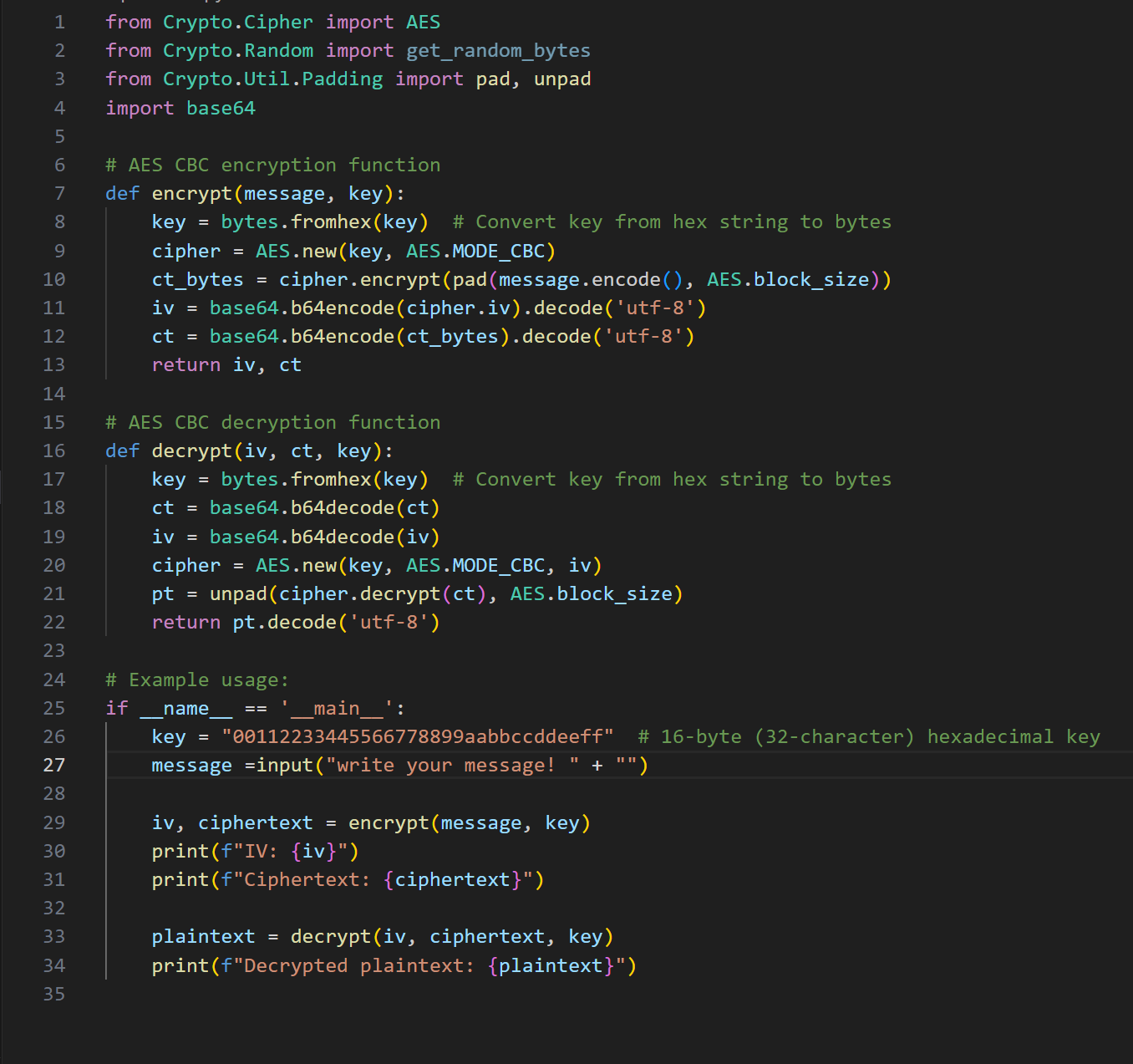
CBC mode output

Part 3:

|  |  |  |
| --- | --- | --- |
| Feature | DES | AES |
| Year Developed | 1977 | 2001 |
| Key Size | 56 bits (weak) | 128, 192, or 256 bits (strong) |
| Security | Insecure for modern applications (vulnerable to brute-force attacks) | Considered highly secure (no known successful attacks) |
| Speed | Slower | Faster than DES, especially in software implementations |
| Block Size | 64 bits | 128 bits |
| Structure | Feistel network with two halves | Substitution-Permutation Network (Substitution and Shifting) |
| Status | Deprecated (not recommended for new applications since 2017) | Current standard for symmetric key encryption |

Here's some reasons why I chose the AES the key size and the high security the software implementation which is I need and best for the network.

The code:



Output of the code:

