# Cryptography Homework 3

## Required Reading

Cryptology 3 slides

## Optional Reading

Wikipedia has good articles on stream and block ciphers, DES and AES  
<https://blog.cryptographyengineering.com/2011/12/15/whats-deal-with-rc4/> (problems with RC4)

## Friendly Advice

This is a complicated assignment. The assignment at the end of this document will be much easier if you duplicate the steps on your own computer as you read through the homework.

## Install PyCryptodome

Please use the document “PyCryptodome Installation.docx” to install PyCryptodome on your computer or VM.

## Python and Strong Cryptography

There are several Python cryptography modules, but some of them are not suitable for use in production systems (see <https://theartofmachinery.com/2017/02/02/dont_use_pycrypto.html>.) A common module you will see in books and articles is PyCrypto. However, PyCrypto has exploitable bugs, and hasn’t been supported since 2014. Don’t use PyCrypto. A new project, PyCryptodome, is a fork of PyCrypto and is supported (<https://github.com/Legrandin/pycryptodome>.) However, even though PyCryptodome has all the encryption pieces we need, it still takes experience in cryptography to assemble those pieces in a secure manner. It is easy to create code based on standard functions like AES and accidently implement flaws that severely weaken the cryptography.

One recommended code for doing symmetric encryption may be found in the pyca/cryptography module found at <https://cryptography.io/en/latest/>. It makes the primitive functions like AES available, but puts them in “The Hazardous Materials Layer.” It provides a “Recipes” layer that assembles the primitives in a secure way; for symmetric encryption, it uses Fernet. Fernet is not a new encryption method. Instead, it uses the existing AES 128 in CBC mode with Hash-based Message Authentication Codes (HMAC) in a secure manner. <https://github.com/fernet/spec/> If you want to do symmetric encryption securely in Python, the Fernet recipe in pyca/cryptography is one possibility.

Our goal in this lab is to look at the pieces (primitives) that comprise current symmetric encryption, specifically AES. While Fernet is more secure it also includes things that we haven’t covered yet, like hashes, HMAC, and tokens. Therefore, we will use the PyCryptodome Python modules for this lab.

# Symmetric Encryption with AES

## Overview

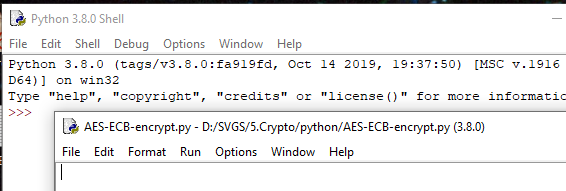
We will perform an AES encryption like that in slide 8 of the class notes, except that we will (pretend to) pass the encrypted data from sender to receiver using base64 so we don’t have to worry about sending binary data. We will do this in Idle, but you can use a terminal and your text editor if you prefer. Either Linux or Windows will be fine.

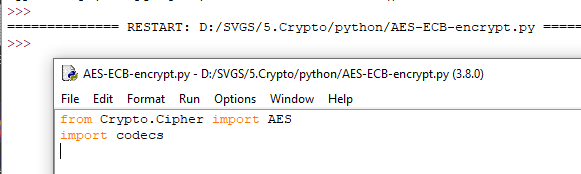
We will create a key and plaintext, and then encrypt the plaintext with the simplest AES mode, Electronic Codebook (ECB). This is not secure, but it is a simple mode to use for our first attempt. The AES module will create ciphertext. We will pretend to send it by email by converting the ciphertext to base64, and then copying it to a new terminal. In the second terminal we will convert the base64 to ciphertext, then decrypt the ciphertext with the same key we used for encryption.  
A close up of a sign

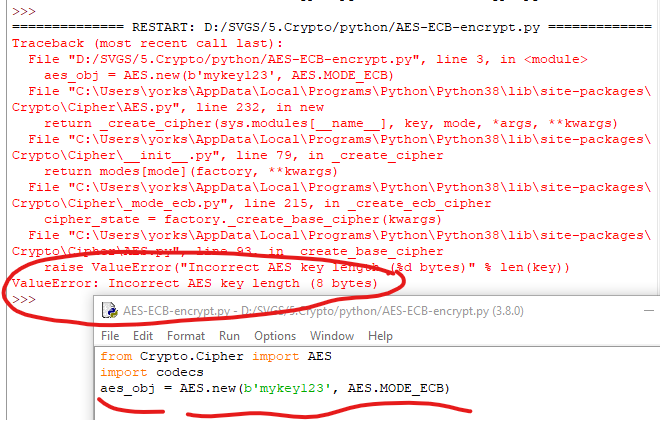
Description automatically generated

## Encryption

Note: This example uses Idle in Windows, but you could just as easily use Idle in Linux or a terminal with a text editor.

Create a new file from Idle and save it in your Python directory (Windows will try to bury it deep in the AppData directory.) I’m calling mine AES-ECB-encrypt.  


Import the AES module from Crypto.Cipher. Also import the codecs module which we will use later. Note that Python is case sensitive, even on Windows. Run your script just to make sure Pycryptodome is properly installed. No errors (no output at all) is good.  


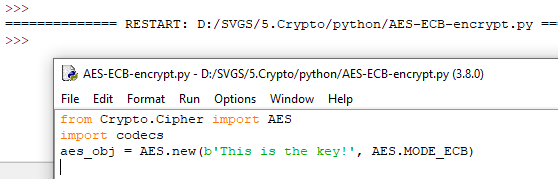
Create an AES object, and give it the key you will use to encrypt. To start with, we will use ECB mode for simplicity and get more realistic later. Remember that the key must be exactly 128, 192, or 256 bits (16, 24, or 32 bytes) long. If the length is different you will receive an error. Test your key to make sure it is 16 bytes (128 bits) long. This is an example of what happens when the key is too short.  


Note: In Python, the most relevant error message is usually the one at the bottom.

You can check the length of your string at the interactive Python prompt ( >>> ) using len().  
len(b'mykey123')  


Once you have a key that is 16 bytes long, use it to create an AES object. Use your own key!

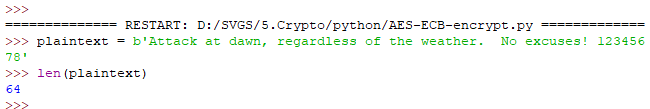
aes\_obj = AES.new(b'This is the key!', AES.MODE\_ECB)

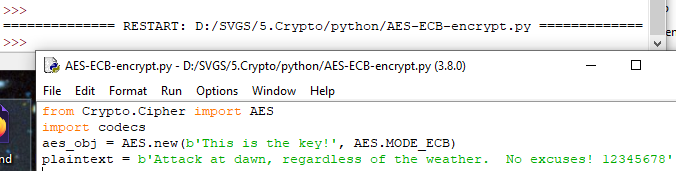


Ah, no errors. Good.

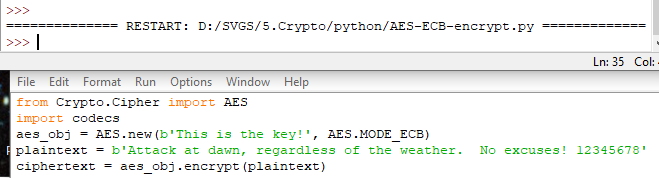
**Important.** Note that there is a “b” at the beginning of the key.  
b'This is the key!'  
The “b” tells Python3 that this is a bytes literal (like a byte array) and not a UTF-8 string object. That is necessary because the AES and codecs modules want bytes as input instead of strings.

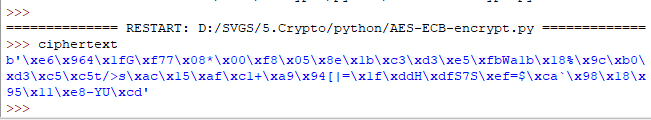
Now we will put our plaintext message into the variable plaintext. The variable ciphertext will hold the encrypted version of the message. AES will only accept blocks of 128 bits or 16 bytes. You will have to add your own padding to make the length of your message a multiple of 16 bytes. Make your own plaintext!



Once your plaintext is a multiple of 16 bytes long add it to your text editor and run it to check for errors.  


Encrypt the plaintext with this line.  
ciphertext = aes\_obj.encrypt(plaintext)

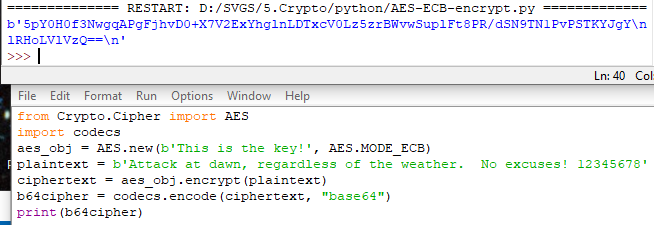


We haven’t coded anything to create output yet, so there is none. When we are working in Idle, we can examine any of the variables in the code we’ve run just by typing the variable name.  


When you examine a variable from the Python interactive prompt, Python will automatically convert it to a viewable string if it can. Since plaintext started as a string, it is readable. The ciphertext is binary data, so Python shows ASCII where it can and renders the rest in hex notation (i.e. \x96). Not pretty.

We will pretend we are sending this ciphertext via email, which transmits characters, not binary data. Rather than send a message full of “\x” characters for hex, we will change the ciphertext to base64. We can use the codecs module; one of the many encoding functions it can perform is base64. We will also print the result to the screen.

b64cipher = codecs.encode(ciphertext, "base64")  
print(b64cipher)

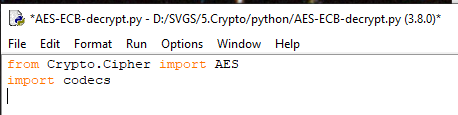


To simulate sending the message to someone, we will copy the base64 message and paste it into a new terminal. In the homework portion, you will send your encrypted message to your partner.

## Decryption

We will do our decryption in a new document and a new terminal. The process is just about the reverse of what we did for encryption. I’ll put that in a new file called AES-ECB-decrypt.py

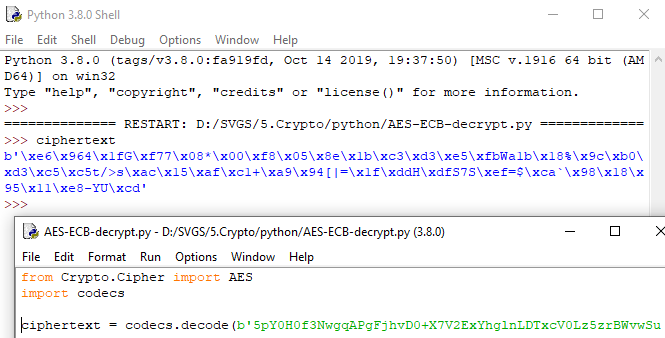
from Crypto.Cipher import AES  
import codecs



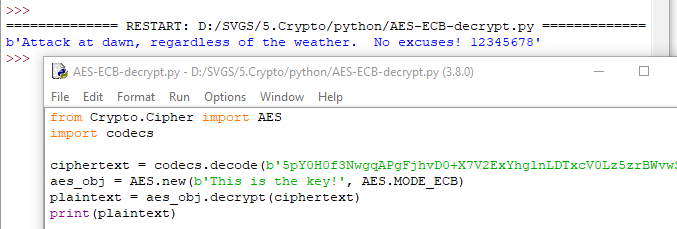
We need to convert the base64 text back to ciphertext. Paste the base64 text from the other terminal into the base64.b64decode command. (The second step, ciphertext, is just to check that the binary hasn’t changed from what was in the first terminal.)

ciphertext = codecs.decode(b'5pY0H0f3NwgqAPgFjhvD0+X7V2ExYhglnLDTxcV0Lz5zrBWvwSuplFt8PR/dSN9TN1PvPSTKYJgYlRHoLVlVzQ==', 'base64')

Hopefully you encrypted your own message, so your ciphertext and b64cipher should be different from what is shown here.

Your new ciphertext should look just like the one in the original window.  


Make an AES object with the same key as before. (The person you are sending to has to know the key, of course.)  
aes\_obj = AES.new(b'This is the key!', AES.MODE\_ECB)  
plaintext = aes\_obj.decrypt(ciphertext)  
print(plaintext)

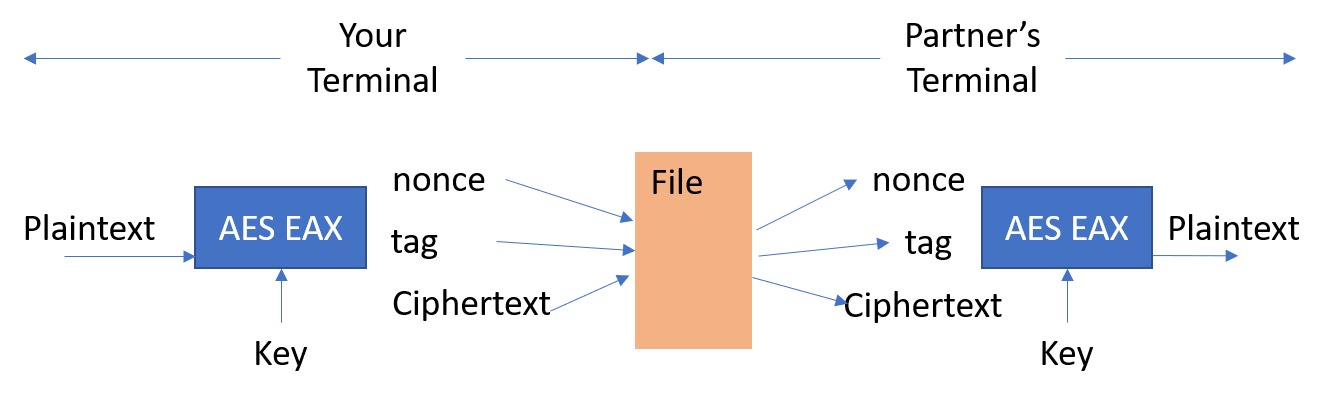


Success!

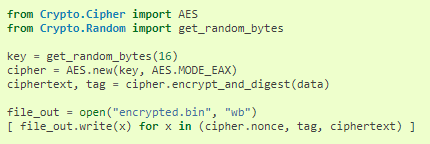
## More Advanced

A better block cipher mode can help us with two problems that the simple ECB mode we’ve been using presents. First, the new mode should introduce a random element, a nonce or Initialization Vector (IV), so that each time we encrypt an identical block of plaintext, we get a different ciphertext. (Remember the Linux penguin picture from the class notes that was encrypted with ECB mode AES, where we could see the outline of the penguin in the ciphertext.) Secondly, it can provide an authentication code so that we are alerted when someone tries to tamper with our cipher text. This example from PyCryptdome uses EAX mode, which provides both protections. <https://www.pycryptodome.org/en/latest/src/examples.html#encrypt-data-with-aes>.

A mode with a nonce (same as IV) and an authentication code (Pycryptodome calls the code a tag) does make things more complicated, though. Now we have three values, cipher.nonce, tag, and ciphertext that are saved together in one file. The decryption side extracts the three pieces from the file and then decrypts the ciphertext. The nonce gave the encryption a random start and the tag allows us to detect corruption or malicious changes to the ciphertext.



## Encryption

<https://www.pycryptodome.org/en/latest/src/examples.html#encrypt-data-with-aes> ****

In this example, the key is randomly generated; if you use that, you’ll need to import get\_random\_bytes. We must find a way to get the key to our recipient (key exchange) without it being compromised. We will work on secure key exchange in a later exercise; for now, just pretend the exchange is secure.

The AES object created in the example is cipher. Instead of using the cipher.encrypt() method we used before, the example uses cipher.encrypt\_and\_digest(). It creates the ciphertext as before, but it also creates a tag, which is an authentication code. When the ciphertext is decrypted the ciphertext and the tag must match. The method will warn us if the ciphertext or tag has been altered.

When the AES object (cipher) is created, a random number, or nonce is also created. The nonce gives the EAX mode a random starting point so that repeated encryption of the same plaintext always gives a different ciphertext. It is a property of cipher, so it is available as cipher.nonce. The result is that we have additional data that needs to travel with the ciphertext. The components are:

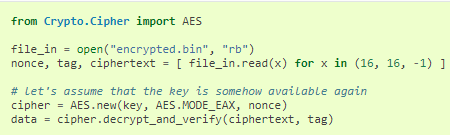
1. Ciphertext. This is the encrypted message created by the cipher.encrypt\_and\_digest() method.
2. Tag. The tag is computed separately from the ciphertext during encryption. It will allow us to verify that the ciphertext has not been tampered with. The variable tag is the second output of the cipher.encrypt\_and\_digest() method.
3. Nonce. The nonce is the random number that gives our encryption a random starting place. It is created by the AES.new() method, and is available as cipher.nonce.

The nonce, tag, and ciphertext all travel together as a package. It does not matter if an attacker sees them as they transit the network, since they are of no value unless the attacker has key.

Note: The code in the example is missing something; it opens the file, file\_out, but it does not close it. If you do this, Python will keep the file contents in buffer and will not write them to disk. Add a line to the end of your code:  
file\_out.close()

The key must be available to the receiver, and it must be sent to the receiver securely in a different channel than the message. If we made a mistake and included the key in our message, our encryption is worthless.

## Decryption

To decrypt the message, we create an AES object using the key, EAX mode and the nonce. (The key was securely exchanged by some other method.) Then we call the cipher.decrypt\_and\_verify() method with the ciphertext and the tag. If the message is corrupted or tampered with, the method will throw an error. Otherwise, it will give us the plaintext of the message.  


This line in the decryption code may look strange at first.  
nonce, tag, ciphertext = [ file\_in.read(x) for x in (16, 16, -1) ]

It just reads the first 16 bytes of the file and puts them into the variable nonce. Then it reads the second 16 bytes and puts them into tag. Everything that is left goes into ciphertext. This line is a “list comprehension”, a technique Python uses to create or modify the contents of a list without a full loop.

Note: EAX uses the Counter Mode (CTR) to randomize the ciphertext, and to avoid the “penguin problem.” It creates the tag with the One-key Message Authentication Code (OMAC) method. The technical details of the EAX mode are here: <http://web.cs.ucdavis.edu/~rogaway/papers/eax.pdf>

# An exercise for you

Split up into pairs and use AES in EAX mode to securely transmit messages of your choosing to each other. That means two messages. Each student should encrypt and send one message and receive and decrypt one message.

## Key generation

First, each person should create a key. The example uses key = get\_random\_bytes(16). If you would rather not have to type a 16-byte random key when you decrypt your partner’s message, you can create your own key.  
key = b'This is my key!1' or something. Just make sure your key is 16 bytes long.

## Key Exchange

Write your key on a scrap of paper and give it to your partner. If you can do that without the other pairs of students spying on you, we will call that “secure key exchange.”

## Create and Encrypt a message

Create variables to contain your key and message. Note that in the example, the message plaintext is stored in the variable data. Also, MODE\_EAX in the Python module does another nice thing for us; it pads the plaintext so that it fits in 128 bit/16 byte blocks.  
key = b'whatever your is'  
data = b'whatever you want the message to be, don’t worry about length. '

Use these lines from the example.  
cipher = AES.new(key, AES.MODE\_EAX)  
ciphertext, tag = cipher.encrypt\_and\_digest(data)

file\_out = open("encrypted.bin", "wb")  
[ file\_out.write(x) for x in (cipher.nonce, tag, ciphertext) ]

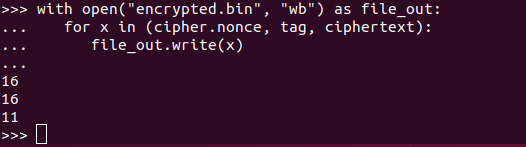
file\_out.close()

Instead of the file\_out lines above, another way is to use the with open format shown below. It will automatically close the file when it is complete.

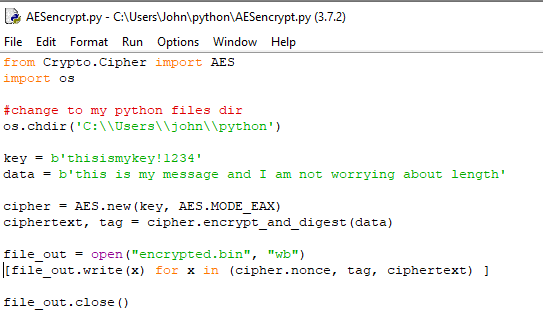
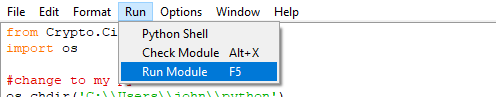
with open("encrypted.bin", "wb") as file\_out:

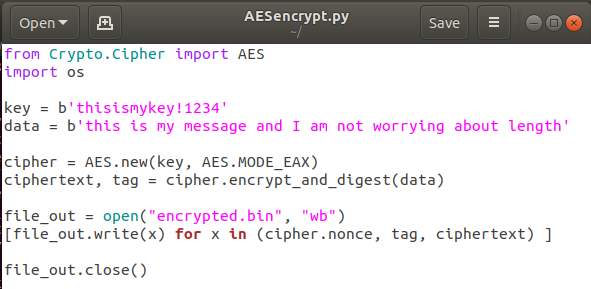
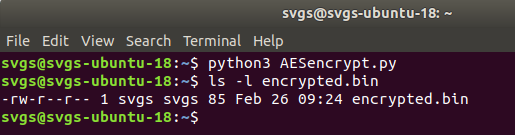
for x in (cipher.nonce, tag, ciphertext):

file\_out.write(x)

Note that the indentations (spaces at the beginning of lines 2 and 3) are important; that is the way that Python identifies script blocks.  


This stores your data in a binary file called encrypted.bin. It concatenates the nonce, tag, and ciphertext into one file. The nonce and tag are 16 bytes each, and the rest is the ciphertext.

This is what it looks like when you create a script in IDLE on Windows.  
 

Here it is as a script in gedit in Ubuntu.  
  


## Transmit the message

## Transmit the message to your partner by emailing encrypted.bin as an attachment, using sneakernet (Copy encrypted.bin to a flash drive and hand it to your partner. In the old days we used floppy disks.), or whatever method seems appropriate. (We won’t bother with base64 this time.)

## Decrypt the message

Decrypt your partner’s message using their key and the code from the example in green, above. You should probably replace the commented line, “Let’s assume that the key is somehow available…” with something like this.  
key = b'This is my partner’s key'

Once you have your partner’s key saved in the variable key, you can use the example code.

file\_in = open("encrypted.bin", "rb")

nonce, tag, ciphertext = [ file\_in.read(x) for x in (16, 16, -1) ]

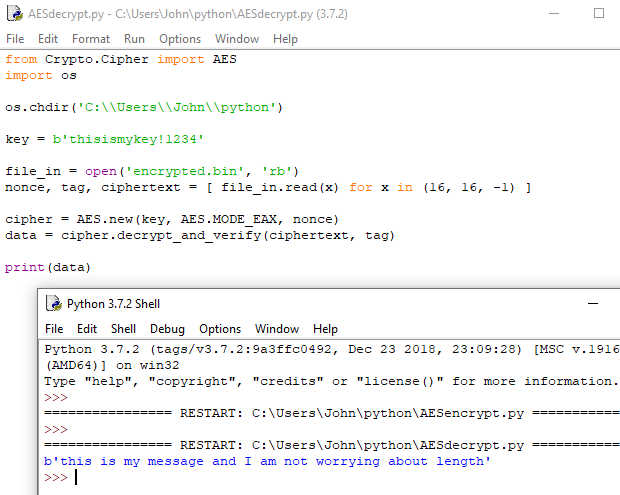
file\_in.close()

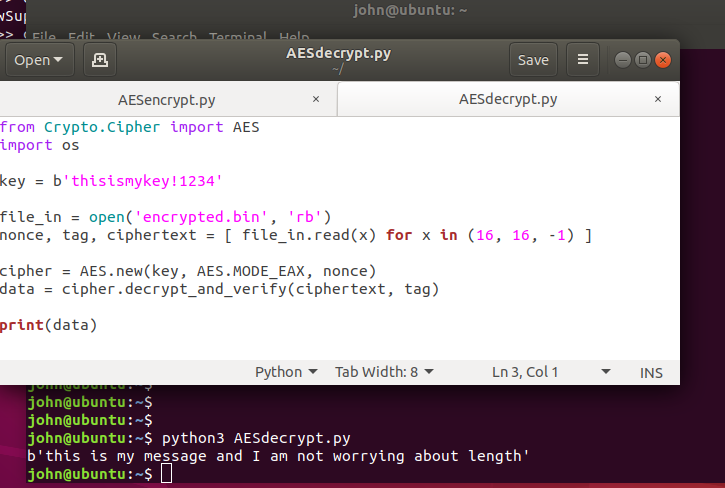
# let's assume that the key is somehow available again

cipher = AES.new(key, AES.MODE\_EAX, nonce)

data = cipher.decrypt\_and\_verify(ciphertext, tag)

The code, file\_in.read(x) for x in (16, 16, -1), creates a list by putting the first 16 bytes of the file in the first entry, the second 16 bytes in the second entry, and everything else in the last entry. Those values are then stored in nonce, tag, ciphertext.

This is what the script looks like in Windows IDLE.  
  


This is what the script looks like in Ubuntu.  
  


# Hand in

Hand in screenshots of your Python terminal as you encrypted your message and decrypted your partner’s message.