Enigma Machine

Eric Oliver

December 6, 2017

```
#!/usr/bin/env python3

import copy
class Enigma:       #Define Enigma Class


    rotors = [ [3, 15, 20, 22, 20, 12, 24, 6, 4, 18, 11, 7, 19, 10, 1, 11, 16, 20, 15, 15, 8, 24, 14, 23, 16, 10],
           [9, 14, 8, 16, 24, 18, 23, 7, 13, 24, 6, 1, 12, 17, 3, 19, 4, 14, 7, 18, 2, 11, 22, 3, 15, 2],
           [16, 24, 12, 14, 11, 13, 20, 12, 5, 3, 17, 9, 16, 24, 7, 14, 20, 5, 12, 16, 3, 10, 12, 1, 8, 8]]
    rotorsRev = [[0 for i in range(26)],[0 for i in range(26)],[0 for i in range(26)]]
    plugboard = [i for i in range(26)]
    checkSwap = [0 for i in range(26)]
    tick = [ 0, 0, 0]
    slot = [-1, -1, -1]
    rotorPosition = [0,0,0]
    rotorSafe = copy.deepcopy(rotors)
    rotorsRevSafe = copy.deepcopy(rotorsRev)
    plugboardSafe = copy.deepcopy(plugboard)
    tickSafe = copy.deepcopy(tick)
    slotSafe = copy.deepcopy(slot)
    rotorPositionSafe = copy.deepcopy(rotorPosition)
    keyArray = [0]
    codeArray = [0]
    codeLength = 0
    textArray = [0]
    newText = " "

    # Constructor Function
    def __init__(self):
        for i in range(0, 3):
            for j in range(0, 26):
                self.rotorsRev[i][(j + (self.rotors[i][j])) % 26] = 26 - (self.rotors[i][j])

    # Function resets Enigma: Does NOT change the key.
    def resetEnigma(self):
        self.rotors = copy.deepcopy(self.rotorSafe)
        self.rotorsRev = copy.deepcopy(self.rotorsRevSafe)
        for i in range(0, 3):
            for j in range(0, 26):
                self.rotorsRev[i][(j + (self.rotors[i][j])) % 26] = 26 - (self.rotors[i][j])
        self.rotorPosition = copy.deepcopy(self.rotorPositionSafe)
        self.tick = copy.deepcopy(self.tickSafe)
        self.plugboard = copy.deepcopy(self.plugboardSafe)
        self.slot = copy.deepcopy(self.slotSafe)
        self.Enigma_setup()
        return


    # Function that establishes the order of the rotors for encryption
    def rotor_Order(self, a, b, c):
        self.slot[0] = a
        self.slot[1] = b
        self.slot[2] = c
        return

```

```
    # Function that places the rotors and the reverse rotors at their beginning positions
    def set_rotorPosition(self):
        for i in range(0, 3):
            temp1 = self.rotors[self.slot[i] - 1][:]
            temp2 = self.rotorsRev[self.slot[i] - 1][:]
            for k in range(0, 26):
                self.rotors[self.slot[i] - 1][k] = temp1[(k + self.tick[i]) % 26]
                self.rotorsRev[self.slot[i] - 1][k] = temp2[(k + self.tick[i]) % 26]
        return

    # Function that turns the rotors and reverse rotor one position
    def rotor_moveOne(self, i):
        temp1 = self.rotors[self.slot[i] - 1][:]
        temp2 = self.rotorsRev[self.slot[i] - 1][:]
        for k in range(0, 26):
            self.rotors[self.slot[i] - 1][k] = temp1[(k + 1) % 26]
            self.rotorsRev[self.slot[i] - 1][k] = temp2[(k + 1) % 26]
        return

    # Function that establishes the beginning position of the rotors and reverse rotors
    def rotor_Set(self, a, b, c):
        self.tick[0] = a
        self.tick[1] = b
        self.tick[2] = c
        self.set_rotorPosition()
        return

    # Function that sets up the plug board
    def plug_board(self, plugs):
        for i in range(0, plugs):
            letterOne = ord(self.keyArray[10 + (2 * i)]) - 97
            letterTwo = ord(self.keyArray[11 + (2 * i)]) - 97
            self.checkSwap[letterOne] = 1
            self.checkSwap[letterTwo] = 1
            temp = self.plugboard[letterOne]
            self.plugboard[letterOne] = self.plugboard[letterTwo]
            self.plugboard[letterTwo] = temp
        return

    # Function that swaps code numbers based on the plugboard settings
    def plugSwap(self, codeIndex):
        self.codeArray[codeIndex] = self.plugboard[self.codeArray[codeIndex]]
        return

    # Function that recodes an incoming letter with the current rotor
    def rotorCoding(self, codeIndex, rotorNum):
        self.codeArray[codeIndex] = (self.codeArray[codeIndex] + self.rotors[self.slot[
rotorNum] - 1][self.codeArray[codeIndex]]) % 26
        return

    # Function that recodes an incoming letter with the current rotor in the reverse
    direction
    def rotorCodingRev(self, codeIndex, rotorNum):
        self.codeArray[codeIndex] = (self.codeArray[codeIndex] + self.rotorsRev[self.slot[
rotorNum] - 1][self.codeArray[codeIndex]]) % 26
        return

    # Function Setup Enigma
    def Enigma_setup(self):
        self.rotor_Order(int(self.keyArray[0]), int(self.keyArray[1]), int(self.keyArray
```

```
    [2]))
        rotorP1 = [self.keyArray[3], self.keyArray[4]]
        rotorP2 = [self.keyArray[5], self.keyArray[6]]
        rotorP3 = [self.keyArray[7], self.keyArray[8]]
        rotorP1 = ''.join(rotorP1)
        rotorP2 = ''.join(rotorP2)
        rotorP3 = ''.join(rotorP3)
        self.rotor_Set(int(rotorP1), int(rotorP2), int(rotorP3))
        plugs = int(self.keyArray[9])
        self.plug_board(plugs)
        return

    # Function Encrypt/Decrypt message
    def encrypt_decrypt(self):
        # This is the coding section. It calls the functions of rotors and reverse rotors
        # The encoding/decoding happens for letter in the array

        for m in range(0, self.codeLength):
            # Plugswap happens at the beginning based on the plugboard settings
            self.plugSwap(m)

            # Sends the numbers through all 3 rotors in forward order
            for n in range(0, 3):
                self.rotorCoding(m, n)

            # Simple Caesar shift for the reflector
            self.codeArray[m] = (self.codeArray[m] + 13) % 26

            # Sends the numbers through all 3 rotors in reverse order
            for n in range(2, -1, -1):
                self.rotorCodingRev(m, n)

            # Once the numbers have exited, they go through the plugboard again
            self.plugSwap(m)

            # Increments the first rotor by one after each letter is encoded
            # if the first rotor goes past 25, then the next rotor is incremented by one
            # if not then the range is maxxed to exit the loop
            for p in range(0, 3):
                temp = (self.tick[p] + 1) % 26
                self.rotor_moveOne(p)
                if (temp > self.tick[p]):
                    self.tick[p] = temp
                    p = 3
                else:
                    self.tick[p] = temp
        # Converts the 0-25 numbers into ASCII numbers and then back into characters
        # for the textArray String
        for m in range(0, self.codeLength):
            self.textArray[m] = chr(self.codeArray[m] + 97)
        return

    # Function sets Key Array then sets Enigma Machine
    def setKey(self, keyString):
        self.keyArray = keyString
        self.resetEnigma()
        return

    # Function takes text string and prepares it for encryption/decryption
    def prepareText(self, textString):
```

```
        self.textArray = list(textString)
        self.codeLength = len(self.textArray)
        self.codeArray = [0 for i in range(self.codeLength)]
        for i in range(0, self.codeLength):
            self.codeArray[i] = ord(self.textArray[i]) - 97
        self.encrypt_decrypt()
        self.newText = ''.join(self.textArray)
        return
```

```
from socket import *      #for establishing a network connection
from queue import Queue   #Used as an inbox
import threading          #Used to readincoming traffic and store it in the inbox queue


"""When creating an EnigmaNetwork object, it must
    be specified whether it will run as a server or
    a client"""
class SocketType:
    SERVER = 1  #If running as a server, pass in 'SocketType.SERVER'
    CLIENT = 2  #If running as a client, pass in 'SocketType.CLIENT'



class EnigmaNet (threading.Thread):
    'Network class for the enigma machine project'

    disconnect_key = "!!!!!!!!!!"   #this gets passed to end the conversation.
    lock = threading.Lock()         #used to safely access resources shared my multiple
    threads

    #initializes an EnigmaNet object
    #PARAMS:
    #       socket_type: the type of socket, whether it be a server or client socket
    #       address: the IP address to use with the socket. if server, use 0.0.0.0
    #       port: the port number to use with the socket
    #PRECONDITIONS:
    #       socket_type must be either a 1 for server socket, or a 2 for client socket
    #       address must be a string
    #       port must be an int
    #RETURNS:
    #       nothing
    def __init__(self, socket_type, address, port):
        if socket_type != 1 and socket_type != 2:       #testing whether a valid
    socket_type was passed in
            raise ValueError("Invalid Socket Type")
        if type(address) is not str:                    #testing if address is a string
            raise TypeError("address must be a string")
        if type(port) is not int:                       #testing if port is an int
            raise TypeError("port must be an int")
        threading.Thread.__init__(self)                 #calling base class constructor
        self.socket_type = socket_type                  #setting the socket type
        self.address = address                          #saving the address
        self.port = port                                #saving the port
        self.inbox = Queue()                            #initializing inbox queue
        self.connection_established = False


    #puts a message in the outbox. The other thread will handle it.
    #PARAMS:
    #       message: the message to send
```

```
    #PRECONDITIONS:
    #       message must not be null, and must be a string
    def send_message(self, message):
        if message is None or type(message) is not str:
            raise ValueError("message must be a non-null string")
        #sending a message is different depending on whether it is a server socket or a client socket
        if self.connection_established:
            if self.socket_type == 2:
                self.sock.send(bytes(message, 'UTF-8'))
            else:
                self.c.send(bytes(message, 'UTF-8'))


    #Will return the oldest message in the inbox, or nothing if there are no messages
    #RETURNS: the message at the top of the inbox
    def recieve_message(self):
        if not self.inbox.empty():
            return self.access_inbox(self.inbox.get)


    #call this method to close the socket and stop the thread.
    def disconnect(self):
        self.send_message(self.disconnect_key)
        self.connection_established = False


    #will return true if there is a message in the inbox, false otherwise
    def have_mail(self):
        return not self.inbox.empty()



    #private helpers
    #-----------------------------------------------------------------------------------------
    #this method gets called after you call <instance>.start()
    #starts the thread which initializes either a client or server socket
    def run(self):
        if self.socket_type == SocketType.SERVER: #start server socket
            self.__start_server()
        else:                                     #start client socket
            self.__start_client()


    #creates a server socket.
    def __start_server(self):
        try:
            self.sock = socket()                                    #creating the socket
            self.sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #needed to prevent errors that may happen when binding to a recently used port
            self.sock.bind((self.address, self.port))         #binding it the the address and port
            self.sock.listen(1)                                     #listening for 1 connection
            c, addr = self.sock.accept()                            #establishing connection with client
            self.c = c
            print ("connection established")
            self.connection_established = True
        except Exception:
            print("An error occurred when creating the server")
```

```
106         while self.connection_established:
107             try:
108                 message = c.recv(1024)                          #if recv doesn't return
    anything, an exception will be raised
109 
110                 #if the disconnect_key is recieved, the connection was closed by the other
     user
111                 if message.decode(encoding="utf-8", errors="strict") == self.
    disconnect_key:
112                     self.connection_established = False
113                     print("The other user has disconnected")
114                     break;
115                 self.access_inbox(self.inbox.put, message)    #puts a message into the
    inbox, gets skipped if exception gets called
116             except Exception:
117                 print("Connection lost")
118                 self.connection_established = False
119         c.close()
120         self.c = ''
121 
122 
123     #starts a client socket and attempts to connect to a server socket
124     #If a it can't connect to the server socket for whatever reason, an error will occur
125     def __start_client(self):
126         try:
127             self.sock = socket()                                #creating the socket
128             self.sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #needed to prevent errors
    that may happen when binding to a recently used port
129             self.sock.connect((self.address, self.port))      #attempting to connect
130             print("connection established")
131             self.connection_established = True
132         except Exception:
133             print("an error occurred when trying to connect to the server")
134         while self.connection_established:
135             try:
136                 message = self.sock.recv(1024)                #the thread will stall here
    and wait for data
137 
138                 #if message = disconnect_key, the connection was closed by the other user
139                 if message.decode(encoding="utf-8", errors="strict") == self.
    disconnect_key:
140                     self.connection_established = False
141                     print("The other user has disconnected")
142                     break
143                 self.access_inbox(self.inbox.put, message)    #puts a message into the
    inbox, gets skipped if exception gets called
144             #except socket.timeout:
145             #   continue
146             except Exception:
147                 print("Connection lost")
148                 self.connection_established = False
149         self.sock.close()
150 
151 
152     #this method is meant to either put a message into the inbox queue, or get a message
    out of the inbox queue
153     #because inbox is a shared resource between threads, it had to be more complicated
    than I would have liked.
154     #PARAMS
155     #   function: a function pointer. pass in either Queue.gets, or Queue.puts
```

```
    #   message(optional): the message to put into the queue
    def access_inbox(self, function, message = 0):
        accessed = False                    #set accessed to false, that way it will loop until it is able to gain access
        while not accessed:
            EnigmaNet.lock.acquire()        #thread locked so only one thread can use the inbox at one time
            if message == 0:
                message = function()        #a message gets read from the inbox
            else:
                function(message.decode(encoding="utf-8", errors="strict")) # message gets put into the inbox
            accessed = True                 #set accessed to true
            EnigmaNet.lock.release()        #release it so the other thread can use
        return message
=======
    'Network class for the enigma machine project'

    host='localhost'
    port=5250

    lock = threading.Lock() #used to safely access resources shared my multiple threads

    #initializes an EnigmaNet object
    #PARAMS:
    #       socket_type: the type of socket, whether it be a server or client socket
    #       address: the IP address to use with the socket. if server, use 0.0.0.0
    #       port: the port number to use with the socket
    #PRECONDITIONS:
    #       socket_type must be either a 1 for server socket, or a 2 for client socket
    #       address must be a string
    #       port must be an int
    #RETURNS:
    #       nothing
    def __init__(self, socket_type, address, port):
        if socket_type != 1 and socket_type != 2:       #testing whether a valid socket_type was passed in
            raise ValueError("Invalid Socket Type")
        if type(address) is not str:                    #testing if address is a string
            raise TypeError("address must be a string")
        if type(port) is not int:                       #testing if port is an int
            raise TypeError("port must be an int")
        threading.Thread.__init__(self)                 #calling base class constructor
        self.socket_type = socket_type                  #setting the socket type
        self.address = address                          #saving the address
        self.port = port                                #saving the port
        self.inbox = Queue()                            #initializing inbox queue
        self.connection_established = False


    #puts a message in the outbox. The other thread will handle it.
    #PARAMS:
    #       message: the message to send
    #PRECONDITIONS:
    #       message must not be null, and must be a string
    def send_message(self, message):
        if message is None or type(message) is not str:
            raise ValueError("message must be a non-null string")
        #sending a message is different depending on whether it is a server socket or a client socket
```

```
        if self.connection_established:
            if self.socket_type == 2:
                self.sock.send(bytes(message, 'UTF-8'))
            else:
                self.c.send(bytes(message, 'UTF-8'))


    #Will return the oldest message in the inbox, or nothing if there are no messages
    #RETURNS: the message at the top of the inbox
    def recieve_message(self):
        if not self.inbox.empty():
            return self.access_inbox(self.inbox.get)


    #call this method to close the socket and stop the thread.
    def disconnect(self):
        self.connection_established = False


    #will return true if there is a message in the inbox, false otherwise
    def have_mail(self):
        return not self.inbox.empty()

    def Set_Host(newHost):
        host=newHost

    def Set_Port(newPort):
        return(1)

    #private helpers
    #-------------------------------------------------------------------------------
    #this method gets called after you call <instance>.start()
    #starts the thread which initializes either a client or server socket
    def run(self):
        if self.socket_type == SocketType.SERVER: #start server socket
            self.__start_server()
        else:                                     #start client socket
            self.__start_client()


    #creates a server socket.
    def __start_server(self):
        try:
            self.sock = socket()                                    #creating the socket
            self.sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #needed to prevent errors
    that may happen when binding to a recently used port
            self.sock.bind((self.address, self.port))         #binding it the the address
    and port
            self.sock.listen(1)                                     #listening for 1 connection
            c, addr = self.sock.accept()                            #establishing connection
    with client
            self.c = c
            print ("connection established")
            self.connection_established = True
        except Exception:
            print("An error occurred when creating the server")
        while self.connection_established:
            try:
                message = c.recv(1024)                              #if recv doesn't return
    anything, an exception will be raised
```

```
267                 if message.decode(encoding="utf-8", errors="strict") == '': #if zero bytes are recieved, the connection was closed by the other user
268                     self.connection_established = False
269                     print("The other user has disconnected")
270                     break;
271                 self.access_inbox(self.inbox.put, message)    #puts a message into the inbox, gets skipped if exception gets called
272             except Exception:
273                 print("Connection lost")
274                 self.connection_established = False
275         c.close()
276         self.c = ''
277 
278 
279     #starts a client socket and attempts to connect to a server socket
280     #If a it can't connect to the server socket for whatever reason, an error will occur
281     def __start_client(self):
282         try:
283             self.sock = socket()                                    #creating the socket
284             self.sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #needed to prevent errors that may happen when binding to a recently used port
285             self.sock.connect((self.address, self.port))      #attempting to connect
286             print("connection established")
287             self.connection_established = True
288         except Exception:
289             print("an error occurred when trying to connect to the server")
290         while self.connection_established:
291             try:
292                 message = self.sock.recv(1024)                #the thread will stall here and wait for data
293                 if message.decode(encoding="utf-8", errors="strict") == '': #if zero bytes are recieved, the connection was closed by the other user
294                     self.connection_established = False
295                     print("The other user has disconnected")
296                     break
297                 self.access_inbox(self.inbox.put, message)    #puts a message into the inbox, gets skipped if exception gets called
298             #except socket.timeout:
299             #   continue
300             except Exception:
301                 print("Connection lost")
302                 self.connection_established = False
303         self.sock.close()
304 
305 
306     #this method is meant to either put a message into the inbox queue, or get a message out of the inbox queue
307     #because inbox is a shared resource between threads, it had to be more complicated than I would have liked.
308     #PARAMS
309     #   function: a function pointer. pass in either Queue.gets, or Queue.puts
310     #   message(optional): the message to put into the queue
311     def access_inbox(self, function, message = 0):
312         accessed = False                    #set accessed to false, that way it will loop until it is able to gain access
313         while not accessed:
314             EnigmaNet.lock.acquire()        #thread locked so only one thread can use the inbox at one time
315             if message == 0:
316                 message = function()        #a message gets read from the inbox
```

```
317             else:
318                 function(message.decode(encoding="utf-8", errors="strict")) # message gets
    put into the inbox
319             accessed = True                 #set accessed to true
320             EnigmaNet.lock.release()        #release it so the other thread can use
321         return message
```

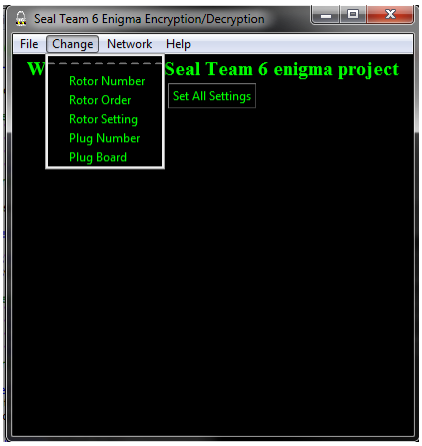

Figure 5.1: Settings for the GUI

Mathematical analysis [edit]

The Enigma transformation for each letter can be specified mathematically as a product of permutations.[16] Assuming a three-rotor German Army/Air Force Enigma, let P denote the plugboard transformation, U denote that of the reflector, and L, M, R denote those of the left, middle and right rotors respectively. Then the encryption E can be expressed as

$$E = PRMLUL^{-1}M^{-1}R^{-1}P^{-1}.$$

After each key press, the rotors turn, changing the transformation. For example, if the right-hand rotor R is rotated i positions, the transformation becomes $\rho^i R\rho^{-i}$, where ρ is the cyclic permutation mapping *A* to *B*, *B* to *C*, and so forth. Similarly, the middle and left-hand rotors can be represented as j and k rotations of M and L. The encryption transformation can then be described as

$$E = P(\rho^i R\rho^{-i})(\rho^j M\rho^{-j})(\rho^k L\rho^{-k})U(\rho^k L^{-1}\rho^{-k})(\rho^j M^{-1}\rho^{-j})(\rho^i R^{-1}\rho^{-i})P^{-1}.$$

Combining three rotors from a set of five, the rotor settings with 26 positions, and the plugboard with ten pairs of letters connected, the military Enigma has 158,962,555,217,826,360,000 (nearly 159 quintillion) different settings.[17]

The Enigma Uhr attachment

Figure 5.2: Complexity Functions

A proper key would be: 3210203033abcdef

3210203033abcdef
‘3/2/1’ – first 3 characters are the rotor order. They need to be different numbers

3210203033abcdef
‘02/03/03 – these numbers are the rotor settings of each rotor. They do not need to be different

3210203033abcdef
‘3’ – number of plugs to attach

3210203033abcdef
‘ab/cd/ef’ – since the number of plugs is 3, you need 3 pairs of letters to connect. Do not repeat letters

Figure 5.3: Keyword Structure

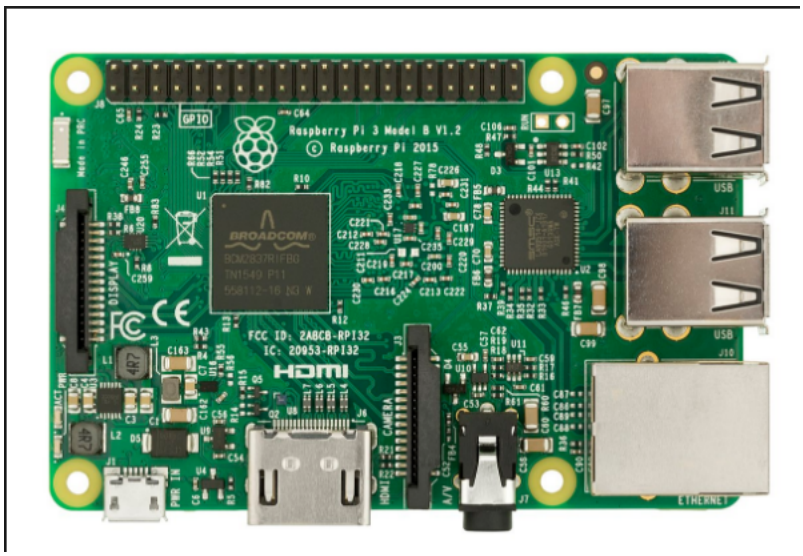

Figure 5.4: Raspberry Pi