Enigma Project Report

Jakub Šmejkal, Suweyda Ali Ugas

December 2025

Contents

1 Encrypt() function

1.1 Logic and Functionality

The initial steps of working on the encrypt function were to simplify the exercise at hand into simpler steps that can be tackled one by one. This process is also known as the top-down method. The assignment has been broken down into the following steps:

1. Find a way to set the initial offset of the Enigma Machine to the offset given in the form of a 3-letter string *rotors*.
2. Simulate the running of the Enigma Machine with the initial offset *rotors* and the letters in *message* typed in individually.
3. Find a way to record the output of the encryption process of each individual letter and return.

Setup

Before we started with writing any new code, we made sure to *import EnigmaModel from EnigmaModel*, which allowed us to use functions and variables defined in the *EnigmaModel.py* file.

Next, we chose to define the variable *alphabet* as the string consisting of all capital letter in the english alphabet. This was used several times as the backbone to many functions and steps in EnigmaModel.py.

Step 1

We needed to have a way of accessing the rotors' offsets from the EnigmaModel.py file in order to change their initial values from "AAA" or 000 to the values given in *rotors*. We chose to do that by assigning *EnigmaModel()* to a variable *model*.

Now we could start focusing on the translation of *rotors* into the initial rotor offsets. What we needed to do was to turn letters into numbers into offset fx. "ABC" → "012" → 0 and 1 and 2, the important observation here is that **the index of a given letter in the alphabet is also the offset of the rotor when showing that specific letter**. Thanks to that, we could implement the following code:

```
{model.rotorX.offset = alphabet.index(rotors[X])}
```

To put it simply, this finds the offset variable of a given rotor in the EnigmaModel.py and assigns to it the index of the first occurance of a given letter in the alphabet string, which is also the desired initial offset for a given rotor.

Step 2

We had to find a way to simulate the functionality of the EnigmaModel.py without running the file and without the need for additional GUI that the program uses.

The logic here is that we can simulate the user clicking on the keyboard using a simple for-loop that cycles through the individual characters in the *message* and runs the key_pressed() function. This acts on the same logic as the EnigmaModel.py where the user's clicking on a specific key also triggered the same function. The for-loop looks as follows:

```
for i in message:
    model.key_pressed(i)
```

Step 3

For this step, we had to first find the output of each encrypted letter, record it, and once the full message has been encrypted, return the cipher in a form of a string.

To find the output of each encrypted letter we had to first realise what it is, in the EnigmaModel.py, that actually returns, or displays, the encrypted letter. As hinted, in this case, it is the lamps that display the encrypted letter by lighting up. So in order to access the encrypted letter we used the is_lamp_on() function. We implemented said function in a simple for-loop that cycles through the alphabet and checks whether any of the corresponding lamps are lit up. If we find that a lamp is on, we append its corresponding letter to a list *result*.

Because the Enigma Model only takes in one argument (letter/click) at a time, once the for-loop finds a lit up lamp, it will not find another one, and only append one letter. After we exit this loop we use the key_released() function to turn the lamp off again, so that we can start the process again and only have one lit up lamp at a time. At last, once all the letters are added into *result*, we turn it into a string using the "".join() function to connect all the letters with an empty space ("") in between and return. The whole section then looks as follows:

```
result = []

for i in message:
    model.key_pressed(i)

    for j in alphabet:
        if model.is_lamp_on(j):
            result.append(j)

    model.key_released(i)

return "".join(result)
```

1.2 Debugging

The process was straightforward without any major issues. One issue did however come up in the early stages of writing this function. Initially the code had EnigmaModel() written at every instance instead of the variable *model*. That however did not lead to the desired outcome, and as we later found out, that was due to EnigmaModel() being called every time it was written, which means that the code did not lead anywhere. In other words, the code just kept on creating new EnigmaModel() again and again, with new lamps, new keyboards, new rotors, and new offsets set to 0. After assigning it to the variable *model*, we kept making changes to the same Enigma Model, meaning that the offsets were actually set to *rotors*, and we could act on the keyboard and the lamps.

2 Find_rotors() function

2.1 Functionality and Logic

The Enigma machine is a so-called encryption device that transforms an original message/plaintext into an encrypted cipher/ciphertext using rotating wheels known as *rotors*. When the starting positions of the three *rotors* are set and an original message is typed on the keyboard, the machine lights up the lamps that together form the encrypted cipher.

The purpose of the *find_rotors* function is to determine the initial *rotor settings* of the Enigma machine when both the original message and the correctly encrypted cipher are known.

To approach this task, the problem is first simplified using a top-down method, where the overall problem is divided into smaller and more manageable steps.

1. **Find all possible *rotor settings***
   The code systematically iterates through all combinations of three letters from the alphabet using three nested loops.

2. **Test each *rotor setting* using encryption**
   For each generated *rotor* configuration, the original message is encrypted using the encrypt function.

3. **Select the correct result**
   The encrypted output is compared with the given cipher. If the two match, the corresponding *rotor setting* is returned. If no match is found after all combinations have been tested, the function returns ”Not found”.

Step 1
The function uses three nested *for* loops, each iterating over the variable `ALPHABET`, which contains all 26 letters of the alphabet. In our implementation, the loops are written as:

```
for a in ALPHABET:
   for b in ALPHABET:
       for c in ALPHABET:
```

The variables **a**, **b**, and **c** serve as placeholders for the three rotor positions. As shown in the code above, the first loop assigns a letter from the alphabet to the variable **a**, the second loop assigns a letter to **b**, and the third loop assigns a letter to **c**. In other words, during each iteration, the variables temporarily store a single letter from `ALPHABET`. They do therefore not represent fixed letters but instead allow the program to systematically iterate through all possible rotor positions.

The nested loops follow a structured iteration pattern. The innermost loop updates the variable **c** on every iteration. After 26 iterations of **c**, the variable **b** advances by one position, and **c** resets to the beginning of the alphabet. Similarly, after **b** has completed 26 iterations, the variable **a** advances by one position. This behavior is similar to the stepping mechanism of the Enigma machine's fast, medium, and slow rotors.

As a result, the generated *rotor settings* progress in a systematic sequence, for example:

```
AAA → AAB → AAC ...  → AAZ → ABA → ABB → ABC → ...  → AZZ → BAA → BAB
→ BAC → ...
```

Since `ALPHABET` contains all 26 letters, the use of three nested loops ensures that every possible three-letter rotor combination is tested. This iteration makes it possible to search for the correct *rotor settings.*

Step 2
The three nested loops collectively define the rotor positions by assigning a letter from the alphabet to each placeholder. Once all three placeholders have been assigned a value, the program creates a three-letter *rotor setting.* This is done by concatenating the placeholders, as shown in the following code:

```
        rotors = a + b + c
```

This means that during each iteration, a single three-letter rotor setting is generated, such as `"AAA"`, `"AAA"`, `"AAB"`, `"AAC"`, and so on. For each generated rotor setting, the original message is encrypted using the line:

```
        encrypt(rotors,message)
```

This simulates how the Enigma machine would encrypt the message if it were initialized with the given rotor configuration.

Step 3
The encrypted output is then compared to the given cipher using the condition:

```
         if encrypt(rotors, message) == cipher:
```

If the encrypted result matches the known cipher, the corresponding rotor configuration is returned immediately, as it represents the correct initial *rotor settings.* If no matching configuration is found after all possible combinations have been tested, the function then returns `"Not found"`.

The implemented solution is able to find the correct *rotor settings* by testing all possible three-letter combinations. However, because this method relies on a so-called brute-force search, it has a limitation in terms of execution time. The time required to find the correct setting depends on when it appears in the search order. For instance, a configuration such as "AAA" will be found very quickly, while a configuration such as "ZZZ" will only be identified after all other combinations have been tested, which leads to much longer runtime.

2.2 Debugging

Writing this program was not a major problem in itself, but setting up the code structure correctly proved to be somewhat challenging. In particular, there was some confusion related to variable names, as they resembled letters from the alphabet, which initially made the logic harder to follow. We also encountered a few indentation errors within the nested loops, which eventually caused the code to fail until they shortly after were corrected.

3 Appendix

3.1 encrypt()

```
from EnigmaModel import EnigmaModel

def encrypt(rotors: str, message: str) -> str:

    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    model = EnigmaModel()

    model.rotor1.offset = alphabet.index(rotors[0])
    model.rotor2.offset = alphabet.index(rotors[1])
    model.rotor3.offset = alphabet.index(rotors[2])

    result = []

    for i in message:
        model.key_pressed(i)

        for j in alphabet:
            if model.is_lamp_on(j):
                result.append(j)

        model.key_released(i)

    return "".join(result)
```

3.2 find_rotors()

```
from Encrypt import encrypt

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def find_rotors(message:str, cipher:str) -> str:

    for a in ALPHABET:
        for b in ALPHABET:
            for c in ALPHABET:
                rotors = a + b + c
                if encrypt(rotors, message)==cipher:
                    return rotors

    return "Not found"
```