# Computer Science NEA

Contents

[Computer Science NEA 1](#_Toc119059001)

[Analysis 3](#_Toc119059002)

[The Problem and what makes it Computational: 3](#_Toc119059003)

[Stakeholders, Questionnaire and Responses 3](#_Toc119059004)

[Stakeholders 3](#_Toc119059005)

[Questions 4](#_Toc119059006)

[Responses – Inioluwa 4](#_Toc119059007)

[Responses – Simbo 4](#_Toc119059008)

[Responses – Akin 5](#_Toc119059009)

[Existing solutions to the problem 5](#_Toc119059010)

[Essential features of my solution 6](#_Toc119059011)

[Limitations of my solution 7](#_Toc119059012)

[Requirements of my solution 7](#_Toc119059013)

[Hardware 7](#_Toc119059014)

[Software 7](#_Toc119059015)

[General Requirements 8](#_Toc119059016)

[Success criteria 8](#_Toc119059017)

[Design 9](#_Toc119059018)

[Breaking down the Problem 9](#_Toc119059019)

[Encrypt 10](#_Toc119059020)

[Decrypt 10](#_Toc119059021)

[Usability Features 11](#_Toc119059022)

[Textboxes 11](#_Toc119059023)

[Buttons 11](#_Toc119059024)

[GUI 11](#_Toc119059025)

[Algorithms of the code 12](#_Toc119059026)

[Main menu 12](#_Toc119059027)

[Encrypt 12](#_Toc119059028)

[Decrypt 13](#_Toc119059029)

[Key variables and data structures 14](#_Toc119059030)

[Test plan: 15](#_Toc119059031)

[Questionnaire 15](#_Toc119059032)

[Development 16](#_Toc119059033)

[Stage One Prototype: 16](#_Toc119059034)

[Encrypt function: 16](#_Toc119059035)

[Decrypt function 17](#_Toc119059036)

[GUI 17](#_Toc119059037)

[Stakeholder feedback: 19](#_Toc119059038)

[Improvements to be made for next prototype 19](#_Toc119059039)

[Stage Two Prototype: 20](#_Toc119059040)

[Encrypt Function 20](#_Toc119059041)

[Decrypt subroutine: 21](#_Toc119059042)

[GUI: 21](#_Toc119059043)

[Stakeholder feedback: 24](#_Toc119059044)

[Improvements to be made for next prototype: 24](#_Toc119059045)

[Stage Three prototype: 25](#_Toc119059046)

[Encrypt function: 25](#_Toc119059047)

[Decrypt function: 25](#_Toc119059048)

[GUI: 25](#_Toc119059049)

[Output of the encrypt function to the GUI: 29](#_Toc119059050)

[Output of the decrypt function to the GUI: 29](#_Toc119059051)

[Evaluation 31](#_Toc119059052)

[Final testing 31](#_Toc119059053)

[Usability testing 32](#_Toc119059054)

# 

# Analysis

## The Problem and what makes it Computational:

Securing data on your device is a big deal for people – people often keep confidential information like passwords on their device that, if stolen, would lead to the confidential information becoming the knowledge of malicious parties. While your device may be protected by a PIN, password or even biometrics, individuals may want an extra layer of security for the data on their device. This is what I plan to provide. In addition to this, there are times that unauthorised users need to read a file that has been encrypted, such as law enforcement investigating criminals or other legal reasons.

I believe this problem is especially suited for computation because what the people want is essentially encryption – they want a way of ensuring that, even if their device is accessed by a malicious actor, their data will not be compromised. Encryption is a way of ensuring this by transforming the data into an unreadable form, from which it cannot be translated back from without a key. This is a task especially suited for computers, because manually changing every single character in a file to another based on a sequence is not only tedious and time-consuming, but prone to human error. In addition, a human may forget the key, meaning an authorised viewer cannot view the file. A computer, however, will happily process large amounts of data, editing every character according to a key, quickly, accurately and without complaint (for the most part). A computer can also create pseudo-random keys that will be more difficult for an unauthorised viewer to derive, as well as store the key indefinitely so authorised viewers can decrypt and view the file.

Decrypting a file without the encryption key is extremely difficult for a person to do, especially if they take a brute force approach to it. This is extremely time consuming for a human to do, but a computer can brute force a key in a much shorter time than a human, again, with very little complaint in an ideal scenario. Even if the user doesn’t have the key, a sufficiently powerful computer can decrypt a file by repeatedly trying every possible key until the correct option is found, and a “sufficiently powerful” computer isn’t especially hard to find in the majority of cases in theory. In practice, a robust method of encryption will generate keys that a computer will take too long to crack to make the effort worthwhile. If we want the intended recipient to be able to read the file, we must ensure that the intended recipient has the key.

Another important objective is to ensure ease of use for users. This will require the use of an interactive GUI which is clear and easy for the users to understand, especially users who are unfamiliar with technology such as the elderly – they have confidential data as well, after all. The GUI must have responsive buttons and text boxes for users to input their text, as well as outputting text in a form where users can copy and paste it into a menu of their desire.

## Stakeholders, Questionnaire and Responses

### Stakeholders

My first stakeholder is my younger brother, Inioluwa. He is a 14-year-old GCSE student, who, like me is interested in computer science. He wishes to encrypt some confidential data that he stores on his phone, such as banking information (card PIN, Online Banking password etc.) and he wonders how he can keep them as a note on his phone while still encrypting them, as if they are stored insecurely and stolen banks will have very little sympathy for you. Therefore, he would use this program to encrypt the confidential data that he stores on his phone, meaning that even if his phone is stolen, his details are safe. Throughout the development of this algorithm, I will be consulting him to test the program and give feedback relating to how the program lives up to his expectations and performs the purpose of encrypting information that the user deems confidential, and successfully decrypting it when the user needs it.

My second stakeholder will be my mother, Simbo. She is a 50-year-old Project Manager who is not extremely familiar with computers and technology. She wishes to also encrypt confidential pieces of data on her various personal devices which are currently in a plaintext form, where anyone who has access to the device can view it. Individuals like her who are not familiar with technology will require a robust GUI to be able to effectively make use of such an algorithm. This is in contrast to Inioluwa, who is a skilled programmer for his age and would have little to no issues running the program off of a command line interface, something that users like Simbo would not be able to understand, even if it was explained to them (and most users in a similar position may not have someone to explain a command-line interface to them. It isn’t commonly used outside of programming). As with Inioluwa, I will be consulting her throughout the development of the program to test it and give her feedback to establish whether the program achieves the objective of being easy to use and understand for those who are not very familiar with technology.

My third stakeholder will be my father, Akin. He is a 53-year-old IT consultant who specialises in management. As someone who is familiar with cybersecurity, he will be familiar with methods of encryption and what a good encryption standard should and shouldn’t do. I will consult him throughout the development so he can give me feedback on the capabilities of the program and possible improvements that he thinks could be made, as well as feedback on the GUI, as he is experienced in creating programs that need to allow those unfamiliar with technology to be able to perform complicated computing functions, even if they cannot grasp the complexity.

### Questions

1. What makes an interface engaging to you?
2. What measures would need to be taken for you to feel your data is secure?
3. What is your understanding of what encryption is and how it works?
4. Is any of the data on your device(s) currently encrypted? If so, what standard of encryption is used?
5. What do you think makes a robust encryption standard?

### Responses – Inioluwa

1. An engaging interface, for me, must be easy to read and easy to understand. If I am unable to immediately understand the interface, I will become disinterested.
2. For me to feel my data is secure, it must be somewhere I can be certain that only authorised people (those who know the password to my phone, for example) can access it, and, even then, preferably the data should be encrypted so only those with the key can decrypt and access it. When all these conditions are met, I will feel my data is truly secure.
3. My understanding is that encrypting a text is changing it to a form where it is illegible using a method defined the standard and a key. The key allows the text to be returned to its original form so those who are authorised (through possession of the key) can read the text.
4. On my devices (a phone and a laptop), any passwords that I have saved to password managers are encrypted, and these are stored on my device. Apart from these, no data on either of my devices is encrypted, and the stuff that is perhaps the most important (bank info) is not encrypted as it is not stored by password managers.
5. A robust encryption standard should be resistant to brute forcing (by generating keys too long to brute force in a human lifetime). Beyond this, there is not much the cryptographers who wrote the standard can do in my opinion

### Responses – Simbo

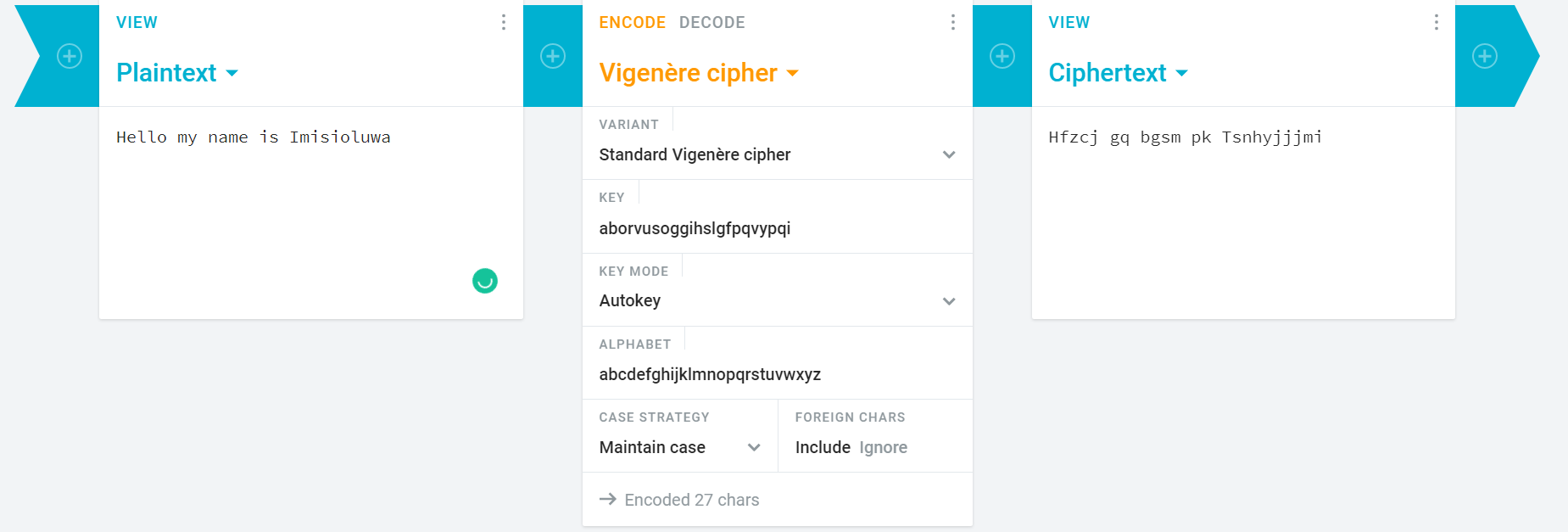
1. An engaging interface must be easy to read and understand, and avoid using complicated jargon – I want to see information I can understand.
2. For me to feel my data is secure, it must be stored on a device that is accessible to only me and other authorised people. Having sensitive data encrypted, for me, is not a necessity on my own personal device
3. My understanding is that encrypted text is changed in some way which makes it difficult for unauthorised people to understand. Encrypted text also has a key that authorised people can use to translate it back to a readable form
4. On my devices I keep all my passwords stored in a notepad, which isn’t encrypted. Adding an extra layer of security to the most sensitive information (bank info, NI number etc) cannot hurt however.
5. A good method of encryption should ensure that it does more than shifting each character 1 or 2 letters in the alphabet – text encrypted like this be deciphered by a moderately intelligent child, and regularly is in 11+ exams.

### Responses – Akin

1. An engaging interface must make sure all information is presented clearly and in a simple form
2. For my data to be secure, it must be stored on my device and my device only to ensure I have sole access to it, and the device must be secured. The data should also be encrypted.
3. Encryption transforms data into a form known as ciphertext which is unreadable – it can be decrypted using a key.
4. All my passwords are stored in a password manager, which uses the AES-256 standard of encryption
5. A good method of encryption must be virtually immune to brute force attacks through a large key which is randomly generated so it cannot be easily guessed based on information like the users' details

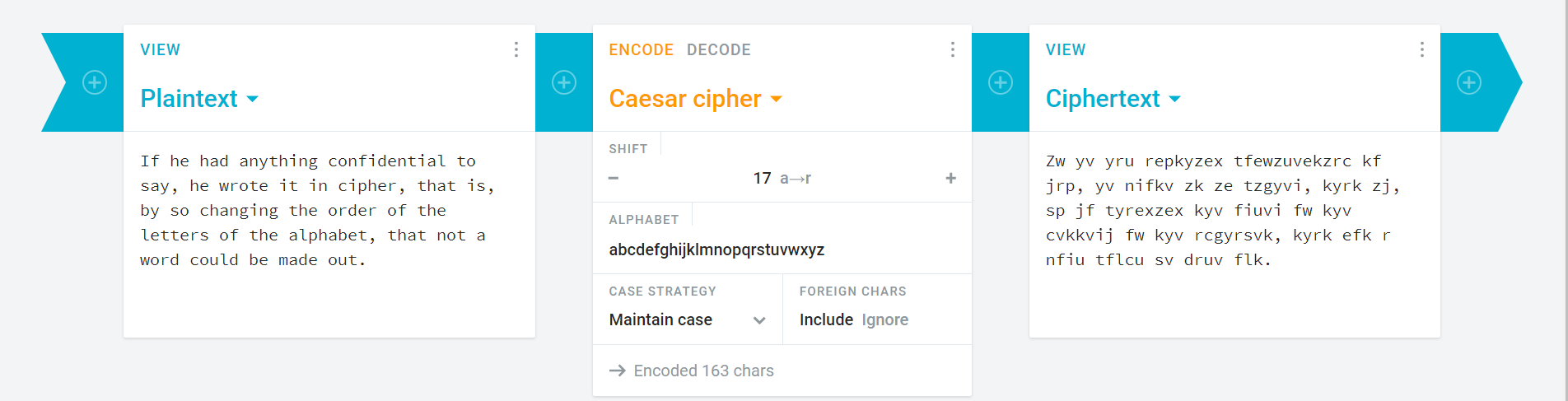
## Existing solutions to the problem

One solution to the problem of encrypting data is the Vigenère cipher. This is a form of encryption where a key is generated for each character in a string, so the final key is a string of approximately the same length as the original string. Different versions of this cipher have the key interact with the text in different ways. Some do an alphabetic shift. Others perform an XOR operation with each letter and its key. Because each character is encrypted using a different key, this prevents decryption through frequency analysis, a method where you use the frequency of occurrence of a specific character to find the key for the text by matching it up with a common letter. This means that an attacker will either have to get a hold of the key or decrypt the ciphertext using brute force. This has 26n possible solutions assuming only alphabetic characters are encrypted, where n is the number of characters in the text. A text with a thousand characters would probably take a few hundred years for a high-end PC to crack using brute force. This is extremely suitable for computation because a computer can easily perform these operations for documents with thousands of characters, accurately and without complaint (for the most part). This cipher works best when the key is pseudorandom and is only used for one text. When this is done, the key is known as a one-time pad and the cipher is theoretically perfectly secure.



This is an example of a Vigenère cipher hosted at the website [Cryptii.com](https://cryptii.com/pipes/vigenere-cipher). It allows the user to enter the plaintext in the left box, and outputs the ciphertext in the right box. The middle box allows users to choose what cipher they wish to use and create a key, which will produce the ciphertext through an alphabetic shift that is performed using the ASCII values of the characters in the plaintext and key. My own program, while not being a website, will be very similar to this, as I believe that the Vigenère cipher approach of giving each letter a different key is very good and secure, however I will prefer to randomly generate my key instead of allowing the user to create one so it is as close to truly random as possible. This website also has a decryption option, which I intend to include in my program as well for the cipher

Another, slightly more simplistic option is a Caesar cipher. This is a method of encryption where a single random number between 1 and 26 is generated or chosen by the user, and the entire text is shifted along the alphabet according to the value of the key. This type of encryption was made famous due to its use by the Roman general Julius Caesar – he was said to have used the cipher with a key of 4 during his military operations. This cipher can quite easily be broken through a brute force attack, as there are only 26 possible plaintexts – a computer can run through that in no time at all. In addition, these types of problems can be seen in exams sat by children (such as the 11 plus), who are expected to (and frequently do) solve them without the use of a computer. Even without brute forcing the key, the cipher can be broken through frequency analysis; letters or combinations of letters that occur often in the ciphertext can be assumed to correspond to letters or combinations of letters that occur often in the plaintext (these can be deduced from lists of the most commonly occurring letters in the language used), and the whole text can be decrypted on this basis, which, again, can be done without a computer.



This is an example of a Caesar cipher, hosted at a site called [Cryptii.com](https://cryptii.com/pipes/caesar-cipher). It allows the user to enter plaintext in the left box, and set the key of the cipher and other conditions of encryption in the middle box, and outputs the ciphertext in the right box. My program, while not a website, would probably have a similar layout, however, I plan to use the more complex Vigenère cipher to ensure the data is secure and the ciphertext cannot easily be decrypted through brute force. I also do not plan to allow the user to choose the key to ensure that it is pseudorandom. The reason the Caesar cipher is so weak is because, however long the text is, there are only 26 possible plaintexts/ciphertexts, compared to the Vigenère cipher, which has 26n possibilities, which makes it unsuitable for satisfying the requirements of my stakeholders (specifically Inioluwa and Akin), as a secure cipher is required.

## Essential features of my solution

One essential feature to my solution is a pseudorandom key. This is because a perfectly secure method of encryption has a truly random key, so it cannot be figured out through any means other than brute forcing the ciphertext. While a truly random key (and indeed true randomness) is mostly just a theoretical concept, as the key is generated using a set process defined in the encryption algorithm, a pseudorandom key can be generated by the majority of high-level languages available for me to use to attempt this task.

Another essential feature is the ability to encrypt data that is not alphabetical. Most of the data my stakeholders have mentioned which must remain confidential, such as debit card PINs and other banking information, are numerical in part, if not in full. If the algorithm can only handle alphabetic data, then it will have a very limited scope in the eyes of my stakeholders, and indeed, many other potential users.

Yet another essential feature is the ability to decrypt data, given the key. This will allow those using my platform to decrypt their own text using the algorithm, which makes the program more useful to them. If it can encrypt data, but not decrypt it, then users may decide that it is better to use a different resource to secure their data, one that allows them to read it as well when they have need of it.

Furthermore, the final essential feature I will mention is the fact that the key must not be constant for every character in the text. This means the algorithm to generate the key should generate a single key for each character in the text, before adding this to a string that will be the final key given to the user. This means any key will have the same number of characters as the plaintext, and since each character will be manipulated mathematically with a different value, frequency analysis cannot be used with great reliability to decrypt the program.

## Limitations of my solution

My program is limited in the fact that consecutive characters in a string may have the same key. A key of this nature will not be truly random, and thus the security of the cipher could be regarded as compromised. While this may not be seen as a big issue by those not familiar with technology, such as Simbo, Akin and Inioluwa will likely be able to pick up on this as a problem with the cipher. Once, however, you decide a character cannot appear twice in a key, you end up with a dilemma when the plaintext has more characters than there are in the character set it is using, and as such, I will make a small concession to avoid compromising the usability of the cipher in a bigger way.

My program is also limited by the fact that users will not be able to move the file containing the key to a location of their choice, unless they move the file containing the source code as well or edit the source code. This is because to access a file in a different folder, the location of the file must be passed to the program, which users who aren’t familiar with the programming language I will use may not be able to do, so the program and the key must stay in the same folder.

The final limitation of my program is the lack of an ability to brute force ciphertext. The program will only decrypt text when the key is provided by the user. This limitation is because decrypting a Vigenère cipher, assuming all Unicode characters are encrypted (and must be decrypted), there are 140,000n possible solutions to the decryption, where n is the number of characters. A ten-character text would have a 51-digit number of solutions; a hundred-character text might as well have an infinite number of solutions. It would not be worth the development time to create such a tool considering how useful it would be, and as such users will have to live without this feature

## Requirements of my solution

### Hardware

* Any computer used to run this game must have basic graphical and processing capabilities
  + The computer must be able to handle windows with solid colour backgrounds, so a dedicated GPU or even a high-end CPU is not necessary – a 10-year-old computer should be able to run this program with no issues
  + The program will not be very resource intensive so the vast majority of computers in use today should have enough memory to run the program comfortably
* The computer should either have a keyboard and the ability to copy and paste text
  + This is so the user will be able to enter text into the program so it can encrypt it, and so they can store the cipher-text themselves
  + To copy and paste text the user will either need a keyboard (which will also allow them to type text) or a mouse to select and right-click text they wish to make use of

### Software

* The computer must have a Python interpreter installed, with the pygame library available as well
  + This is because I will write the program in Python, which, being an interpreted language, means the source code must be downloaded by the user and they must be able to run it for them to use the program
  + The pygame library is the library used to create a GUI in python without which the program will not run
* The computer must have the storage space to store their key as a text file
  + For texts with a small number of characters like a debit-card PIN this will unimportant, but for larger files (let’s say someone wished to encrypt this document) the amount of storage space needed to store the key, which will be the same length as the encrypted text, may need to be carefully considered
  + The program will create a text file on which the key will be stored, which the user will always have access to – files are created in the same folder as the program

### 

### General Requirements

1. The program must generate a pseudo-random key which will be used to encrypt a plaintext. The key must be cryptographically secure to be in keeping with the requirements of my stakeholders. This means it **must be** generated by a computer, not decided by human input, and must be as close to truly random as possible
2. The program must have a functional graphical user interface – not many people, even those who are proficient with technology, would be happy to use a program which can only be used in a command-line interface, so it must open in a separate window which appears more engaging than a command-line interface. The interface must be easy to use and understand, while avoiding the presence of complicated jargon which users may have difficulty understanding
3. The program should avoid shifting characters by only one or two characters. This is to ensure the data is properly secured – a shift of one or two characters will be fairly easy to decode for a person. This must be a condition of the key in the code that generates it to ensure that no generated key for an individual character that breaks this rule is passed to the final key.
4. The program should write the key to a file when it has been generated. This is so the user will have access to the key when they need it – for most encrypted texts the key will be too long to reasonably expect the user to remember having seen it only once – if they cannot recall the key then their text cannot be decrypted for them to use, not in a reasonable amount of time anyway.
5. The program should be able to decrypt text encrypted through this method if given the key. This will increase the utility of the program as it is able to help users encrypt and decrypt their text, as opposed to just encrypting.

### Success criteria

* The solution should securely encrypt the data of users
  + The program must contain a subroutine to generate a pseudorandom key for a Vigenère cipher to encrypt the plaintext inputted by the user
    - This subroutine should generate a different key for each letter, which will be added to a string containing all the keys for each individual character – this will be the final key
* The solution should use an engaging interface
  + The interface should have a coloured background to prevent it looking to bland and unengaging
  + The interface should have a clear and easy to read layout, with obvious textboxes for inputted and outputted text and a lack of technical jargon to ensure users can understand how to use the program whether they are proficient with technology or not
  + The interface must be responsive to action by the user – it should open and close when requested and accept inputs to the textbox when the user wishes to input text
* The solution must be able to decrypt users’ text, given the key
  + The program should then perform an XOR operation using the ciphertext and the key
    - Each letter of the key corresponds to the letter in the same position in the ciphertext, so the operation will be performed letter by letter

# Design

## Breaking down the Problem

### Encrypt

*Key generation:*

To generate the key, the secrets library in Python will be loaded. This library allows the generation of pseudo-random numbers suitable for cryptographic purposes, in contrast to the pseudo-random numbers generated using the random library, which are not suitable for cryptographic purposes. For every character in the plaintext (including spaces), a random character in the Unicode character set will be generated so the key will be as long as the text character wise. The key will not be passed as a parameter, but generated within the subroutine itself, to ensure that it is not user inputted, and so suitable for cryptographic purposes.

*Write key to text file:*

The key will be written in it’s entirety to a text file created to store it. This will allow the program to recall the key and use it to decrypt text with it. The text file will also be passed to the user which will allow them to do with it as they wish

*Plaintext input::*

The user will input the string they wish to encrypt into a textbox provided by the GUI. This string will be stored in a variable called Plaintext which will be passed as a parameter in the encrypt subroutine. There will not be a limit as to the length of this string, as ideally the program should be able to encrypt strings of any length.

*Plaintext XOR Key*

The Plaintext will interact with the key in an XOR operation. As the two are the same length, each character in the key corresponds with a character in the Plaintext. Each letter in the Plaintext will perform an XOR bitwise operation with the key to, character by character, create a variable called Ciphertext, which will hold the encrypted text.

*Returned Ciphertext*

The Ciphertext will be returned by the encrypt subroutine, along with the key, to the main program. This text will be the same length as the Plaintext initially passed to the subroutine as a parameter and the key generated by the subroutine.

*Text file creation:*

A text file will be created and opened in write mode. This file is intended to store they key for the user, which will be written to the file, and the file will be available for them to access and pass on to the intended recipient of any message written, or use in another program/subroutine. A text file to store the ciphertext will also be created, for a similar purpose.

### Decrypt

*Key input/read from file:*

The user has two options here – one is to input the key themselves, and the other is to let the program read the key from a text file created when encrypting the text. If inputting the key, a text box will be available in the GUI, and there will be no character limit. Care must be taken with spaces to ensure each character in the key corresponds to the correct character in the Ciphertext. The key will be passed as a parameter in the subroutine as it is central to the operation of this solution to an age-old problem.

*Ciphertext input:*

The user will enter the text into a text box provided in the GUI, which will be stored in a variable called Ciphertext so it can be passed as a parameter in the subroutine. There will be no character limit on the text, however, a warning may be given to users if their string exceeds a certain amount of characters to close other programs open on their computer so more memory will be available to ensure the decryption is executed as quickly as possible.

*Ciphertext XOR Key:*

The ciphertext and the key will interact in an XOR bitwise operation. This operation will essentially be the reverse of the operation which created the ciphertext in the first place; each character in the ciphertext will be evaluated against the corresponding character in the key through an XOR operation. This should result, when all the individual results are combined into one variable, in the plaintext, the actual message which was encrypted to prevent prying eyes from reading it.

*Return Plaintext:*

The plaintext created in the previous step will be returned by the decrypt subroutine to the main program. This text should be the same length as the Ciphertext initially passed to the subroutine.

## Usability Features

### Textboxes

One of the most important usability features in this program will be textboxes. The user will input text into this box for the program to encrypt/decrypt, and will read text that has been encrypted/decrypted through text boxes as well. This will make it easy for the user to interact with the program concerning text that needs to inputted and outputted. Textboxes must be clearly distinct from buttons, and of an appropriate size for the magnitude of text that will be entered into them and outputted; a scrolling function may be needed for ease of use.

### Buttons

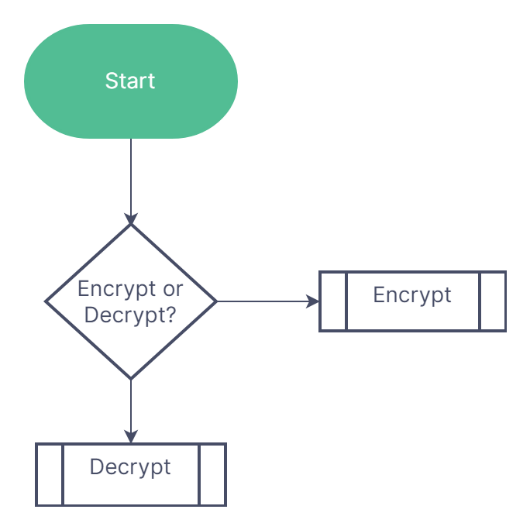
Another important usability feature is the use of buttons. The usage of buttons will make it easier for the user to chose the exact functions that they wish to perform using the program; buttons can be used to select whether the user wishes to encrypt or decrypt, or whether the user wishes to enter the key themselves or have it read from the text file, amongst other things. The buttons must be clearly labelled to ensure that users are aware of exactly what function the button is meant to select.

### GUI

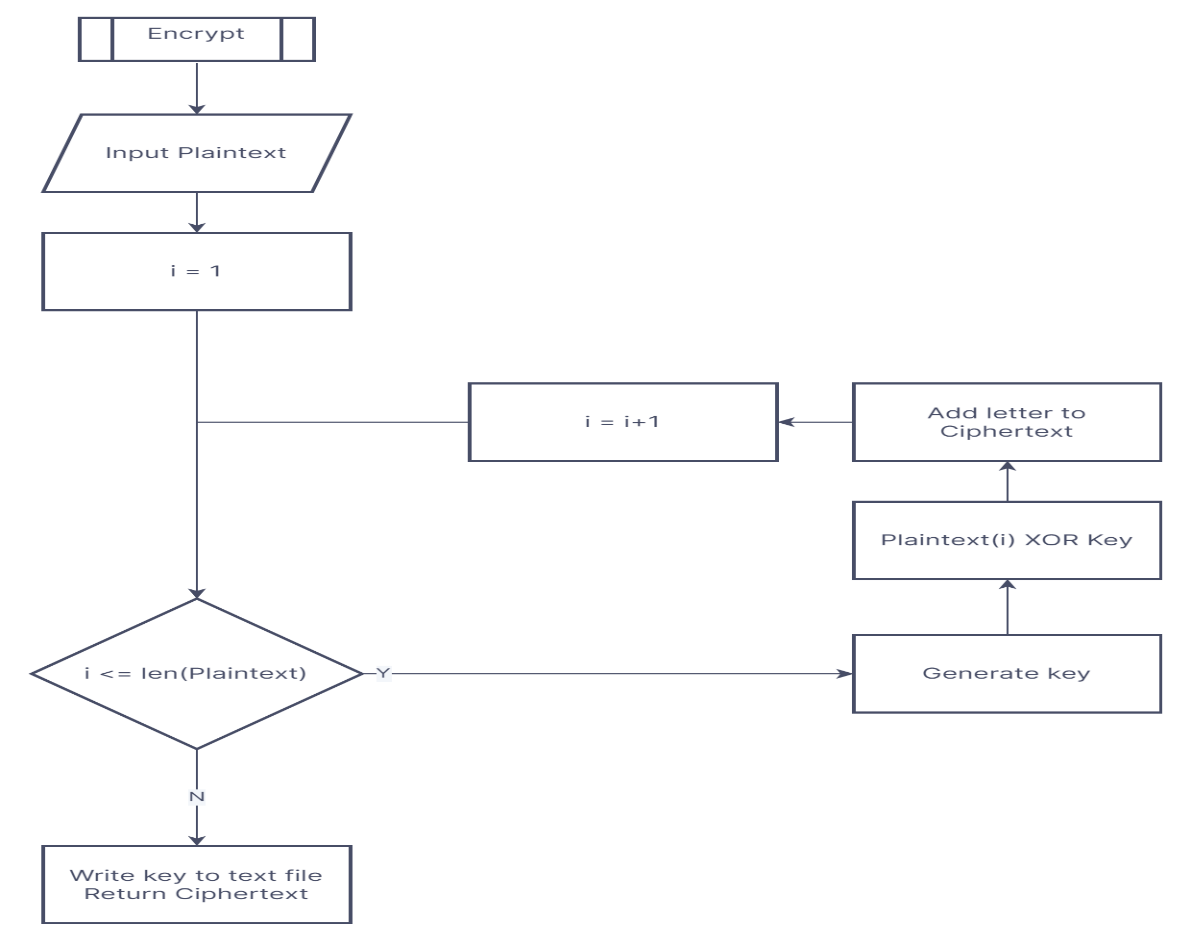
The program should have a Graphical User Interface, as this will make it very simple for users to interact with the program. Those who are unfamiliar with technology may find it difficult or disconcerting to use the program through a command line interface, so a GUI will be necessary. This interface will contain all the buttons and textboxes needed, and should be laid out in a clear and well spaced manner to ensure users are not confused about what does what and can clearly distinguish between buttons and textboxes.

## Algorithms of the code

### Main menu



### Encrypt



Function vigenere(x):

ciphertext = []

key = []

for letter in x:

individualkey = random\_ascii\_letter # in the actual

# python code I will choose a random character

# in the alphabet

newletter = letter Xor individualkey

key.append (individualkey)

ciphertext.append (newletter)

for element in key:

keyfile.write(str(element))

keyfile.write("\n")

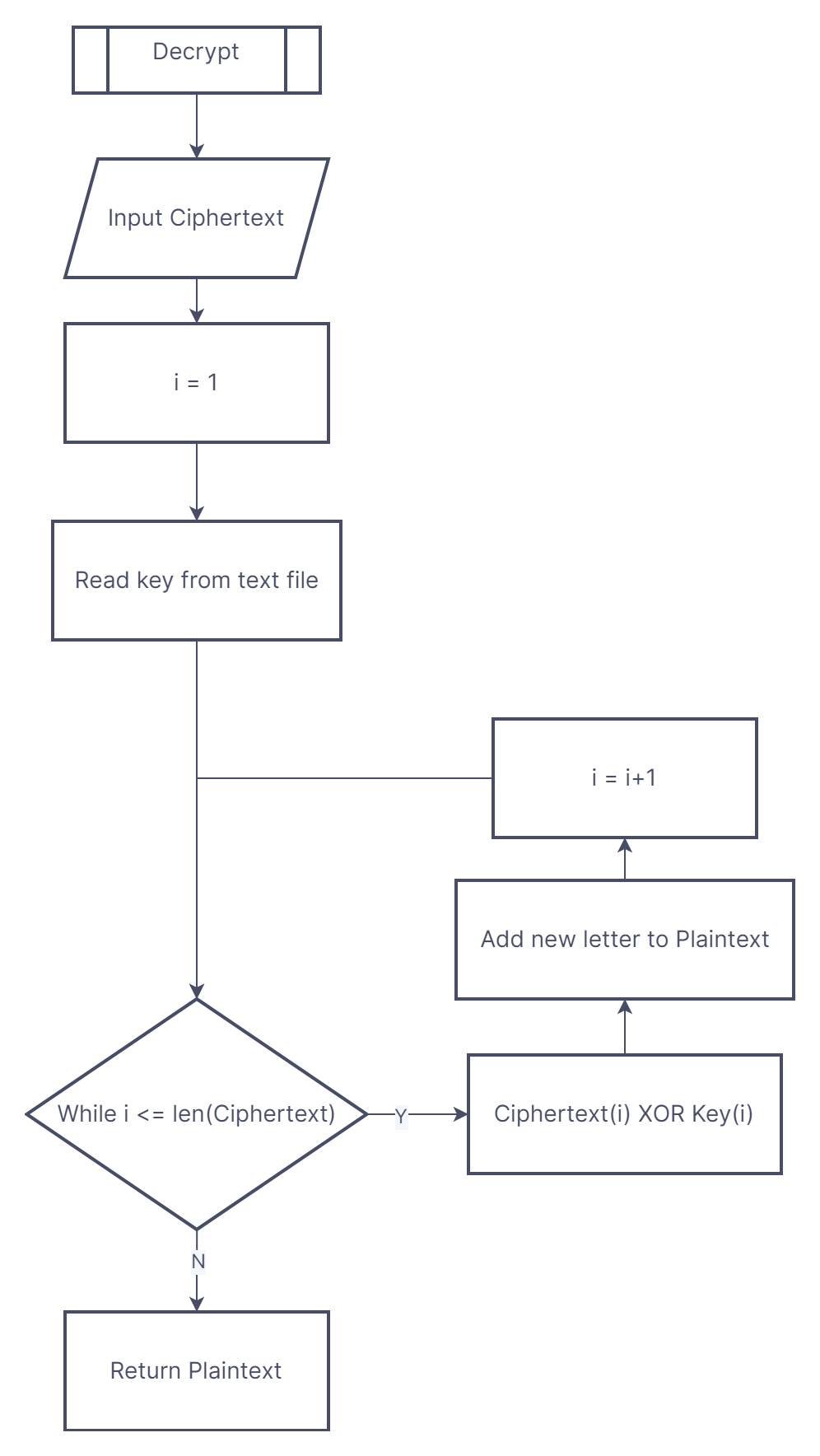
for element in ciphertext:

ciphertext\_file.write(str(element))

ciphertext\_file.write("\n")

return ciphertext

### Decrypt



Function vigerne\_decrypt(y):

Plaintext = ""

keylist = keyfile.read().splitlines()

for i in range(Len(y)):

newletter = y[i] Xor key[i]

Plaintext += newletter

return Plaintext

endfunction

## Key variables and data structures

Plaintext: This will be passed as the parameter “x” in the encrypt function. This will be inputted by the user, as this variable will be the text that they wish to encrypt. To allow for the input of any character this variable will be of the string datatype. In the decrypt function, this will be created by performing an XOR operation on the corresponding characters in the variable keylist and the ciphertext to decrypt the ciphertext and will still be of the string data type.

Individualkey: This variable is generated as a random letter in the English alphabet for every iteration of the first for loop (so every letter will have a different key). This variable is then appended to a list so every key is stored for later use in the decrypt function. In the decrypt function this name is used again to refer to the key corresponding to a certain character in the ciphertext to allow for the ciphertext’s decryption, As this variable is a letter, it will be of the string data type

Key: This list will store the generated key for each individual letter of the plaintext, so it can be decrypted when this list is written to a file.

Ciphertext: When an XOR operation is used between a letter in plaintext and its individual key, this is where the result is stored. Every single letter’s encryption is stored in this list, which will be stored in a text file to be used later by the decrypt function. In the decrypt function this variable is passed as the parameter “y” so it can be decrypted by this subroutine. To avoid the print – display problem ([see Stage One of the Development section](#_Encrypt_function:)) the elements of this list will be integers.

Keyfile: This is the text file the list vigkey will be written to line by line, to allow for the use of the keys for each character in the plaintext later when decrypting the ciphertext. Its contents will be written to the file vernam\_key to allow for their use in the decrypt subroutine

Ciphertext\_file: This is the text file the variable ciphertext will be written to allow for it to be used by the decrypt function at another time, in conjunction with the vigernkey file, to decrypt it. This file can be written to the variable ciphertext in its entirety to be used by the decrypt function as its parameter “y”.

Keylist: This list will store the key used to encrypt letters of the plaintext for use when decrypting them; its contents will be written to it form the file vigernkey.

## Test plan:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Item | Test Number | Type | Test Input | Justification | Expected Outcome |
| Buttons | 1 | Valid | Button clicked | To check that the buttons respond to being pressed | The screen changes depending on which button was pressed |
| 2 | Invalid | Click elsewhere on the screen | To ensure the program only responds to the button being clicked | The screen doesn’t change at all |
| Text box | 3 | Valid | Text being typed is displayed | To ensure the user is able to see the text they’re entering | The letters assigned to the keys being pressed are displayed on the screen |
| Encrypt function | 3 | Valid | Type any combination of characters | The ciphertext variable should be treated at string, so any character should be acceptable | The function should take the string and encrypt it |
| Decrypt function | 4 | Valid | Encrypt any combination of characters using the encrypt function | The decrypt function must be able to decrypt anything the encrypt function can take as input | The original text entered into the encrypt function should be displayed on the screen |

## Post-development Questionnaire

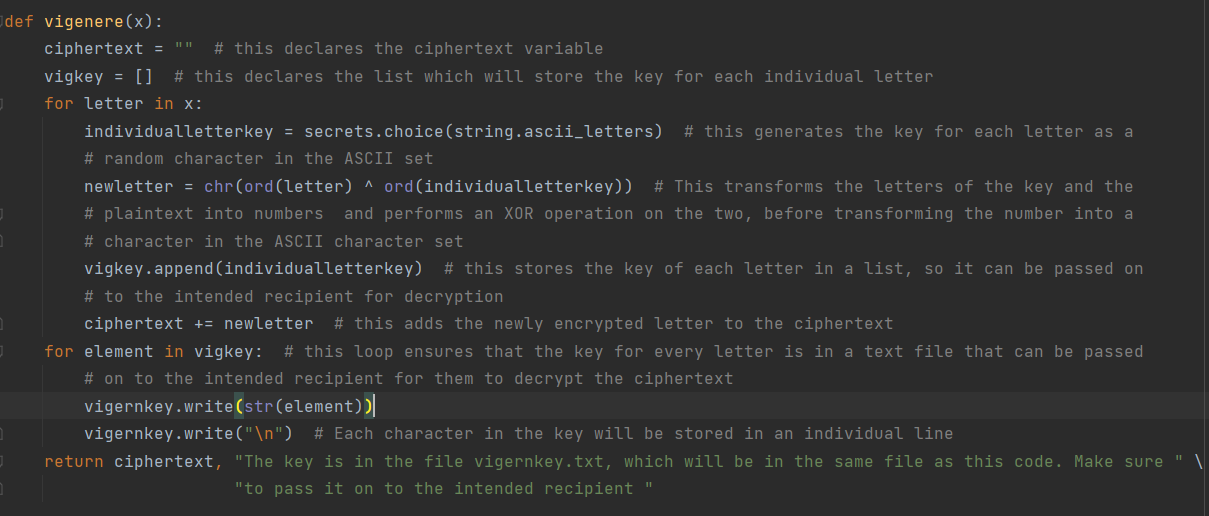
By using a questionnaire after completing the development of my program and having my stakeholders complete it to allow for them to offer me targeted feedback on what they like and dislike about my program. This will allow me to make any necessary improvements to the general function of my program and also help me to make any Quality-Of-Life (QOL) changes that will allow for easier use of my program.

1. What do you think about the interface of the program? Do you like the colour scheme?
   1. Is all the text visible and of a decent size?
   2. Do the buttons respond accurately when you press them
   3. Do you have issues using the textbox?
2. Can you find the files where the ciphertext and key are stored? Are they in the same area as the program?
3. Are you able to easily input data to be encrypted?
4. Is your data accurately decrypted when you use the decrypt function?
   1. Is this text visible on the screen?

# Development

## Stage One Prototype:

### Encrypt function:



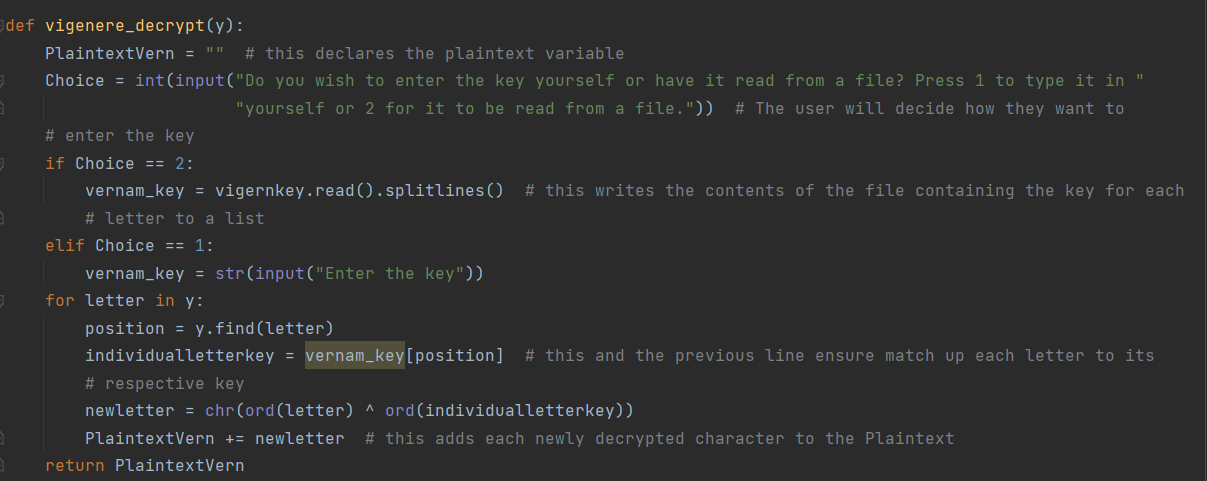
The output of this function can be seen here:



What has happened here is that the subroutine, taking the ciphertext as its parameter, has randomly generated a key that contains characters that are uppercase and lowercase letters. The corresponding characters in the ciphertext and key are converted to their Unicode encoding using the ord function in python. Then, the two numbers interact in an XOR calculation; the result of this calculation is the encoding of the corresponding character in the ciphertext.

As you can see, at this stage of the prototype, we have encountered an issue known as the print/display problem, where some of the characters in the ciphertext cannot be displayed, and as such their hexadecimal codes are given instead. This has not been accounted for before reaching this point, and as such the decrypt subroutine isn’t able to work with ciphertext in a hexadecimal format – it will treat each individual character in the hexadecimal encoding as an individual letter key, which is obviously not the intention.

### Decrypt function



The output of this function can be seen here

 In this scenario the key used was d(L%].

This key and ciphertext were taken from [Isaac Computer Science](https://isaaccomputerscience.org/questions/data_encrypt_07_aqa?examBoard=all&stage=all) to be used as test data for this function due to the dilemma caused by the print/display problem affecting the encrypt subroutine. What has happened here is that both the key and the ciphertext have been converted to Unicode using the ord function in python, and then the number encodings for corresponding clues have interacted with each other through an XOR operation. The result of this calculation is therefore the Unicode value which encodes the corresponding value in the plaintext. For example, 6^d (The character “^” is used as the XOR operator in python) gives a result of R.

### GUI

WIDTH, HEIGHT = 900, 500 # this declares the size of the window the GUI will open in  
WIN = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("Vigenere cipher") # this titles the window  
FPS = 60 # this declares the magnitude of the framerate of the window  
  
  
class Button: # This class will be used as buttons to allow the user to make selections in the program  
 def \_\_init\_\_(self, x, y, image):  
 width = image.get\_width()  
 height = image.get\_height()  
 self.image = image  
 self.rect = self.image.get\_rect()  
 self.rect.topleft = (x, y)  
  
 def draw(self):  
 WIN.blit(self.image, (self.rect.x, self.rect.y))  
  
  
encryptimg = pygame.image.load(os.path.join("Assets", "Encrypt\_button.png")) # This line and the next contain the  
# images that will be the program's buttons and allow them to be utilised in the programs  
decryptimg = pygame.image.load(os.path.join("Assets", "Decrypt\_button.png"))  
  
encrypt\_button = Button(100, 150, encryptimg) # These lines create the buttons  
decrypt\_button = Button(550, 150, decryptimg)  
  
  
def draw\_window(): # this function colours the window  
 GREY = (54, 45, 45)  
 WIN.fill(GREY) # this colours the window grey  
 encrypt\_button.draw()  
 decrypt\_button.draw()  
 pygame.display.update() # this updates the colouring of the window the next time the frame cycles  
  
  
def Coursework(): # the application loop that all pygame programs require to function  
 clock = pygame.time.Clock() # this creates a variable, that, when a value is assigned to it, it will be the  
 # refresh rate of the application  
 run = True  
 while run: # everything in this loop will only happen when the window is open  
 clock.tick(FPS) # this refreshes the frame every 1/FPS seconds, setting a framerate for the program  
 for event in pygame.event.get(): # the event loop that ensures that any events are properly managed  
 if event.type == pygame.QUIT:  
 run = False # this ends the loop if the user quits the program  
 draw\_window() # this ensures that when this function is called the window is coloured grey by calling the  
 # draw window function  
  
 pygame.quit()  
  
  
if \_\_name\_\_ == "\_\_main\_\_": # this ensures that the GUI will open when this specific program is running  
 Coursework()

The output of the above code can be seen below:



This prototype of the UI is a simple grey background with two buttons – Encrypt and Decrypt. These buttons, when fully functional, will allow for the user to choose whether they wish to use the encrypt or decrypt function by clicking the respective button. This prototype, while not functional, acts as a demonstration of how the GUI should appear when opened. The buttons ensure that all inputs are valid in this section – it is possible that the user could spell either word incorrectly if they were required to type in their selection, but with buttons the user can only choose from a set of already valid options.

I have fulfilled requirement 1 in this prototype of the program only partially, as it generates a pseudo-random key from all the alphabetical characters in the ASCII character set in the encrypt subroutine, meaning there are 52n possible complete keys, where n is the number of characters in the key, but this key has not been used to encrypt text due to the unforeseen print/display problem. The program has a graphical user interface which is very simple to understand, however it is not functional and as such this requirement (number 2) has not been fully fulfilled by this prototype. This program is, however, able to decrypt text encrypted through the method used to encrypt text by this program, and as such, the program has fulfilled requirement number 5.

### Stakeholder feedback:

|  |  |
| --- | --- |
| Stakeholder | Feedback |
| Inioluwa | The decrypt subroutine is working but the encrypt subroutine is not currently usable, meaning the use of the program is limited at best; if I have to go elsewhere to encrypt my data why not decrypt it elsewhere as well? |
| Simbo | The general layout of the GUI so far is good; the buttons are of a good size and the text on them is easily readable. I would like the background to be of a brighter colour though, but that’s a novelty, not a need |
| Akin | The method used to encrypt the ciphertext is cryptographically secure in theory, but in practice this is useless if the subroutine doesn’t work. The GUI is also simplistic and easy to understand and this template will help make it easy to use when it is made fully functional |

### Improvements to be made for next prototype

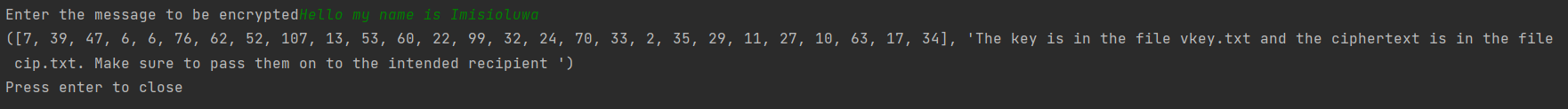
* The buttons in the GUI should be made functional, and should respond when clicked; perhaps they will bring up a textbox or similar entity
* The encryption subroutine should output usable text; this will mean that the print/display problem should be solved so that the ciphertext can be easily manipulated by the decrypt subroutine
  + Perhaps the ciphertext can be left in numerical form, allowing it to be displayed by the program and manipulated by the decrypt subroutine
  + The decrypt subroutine must be able to accurately work with the ciphertext generated by this subroutine

## Stage Two Prototype:

### Encrypt Function

def vernam(y):  
 ciphertext = []  
 vernamkey = []  
 for letter in y:  
 individualletterkey = secrets.choice(string.ascii\_letters) # this generates a different key for each letter  
 vernamkey.append(individualletterkey) # each key is added to a list which will be written to a file  
 newletter = ord(letter) ^ ord(individualletterkey) # crucially, the characters in the ciphertext are left in  
 # numeric form to avoid the print/display problem  
 ciphertext.append(newletter)  
 for element in vernamkey: # this loop ensures that the key for every letter is in a text file that can be passed  
 # on to the intended recipient for them to decrypt  
 vkey.write(str(element))  
 vkey.write("\n")  
 for element in ciphertext: # this loop writes the ciphertext to a text file that the user will retrieve  
 ciphertext\_file.write(str(element))  
 ciphertext\_file.write("\n")  
 return ciphertext, "The key is in the file vkey.txt and the ciphertext is in the file cip.txt. Make sure to pass " \  
 "them on to the intended recipient "

The output of this function can be seen below



As you can see here, the plaintext I have entered has been encrypted using a randomly generated key. To avoid the issues of the print/display problem, in this iteration of this subroutine, I have chosen not to convert the ciphertext from a numerical format to actual characters, as some of the characters returned couldn’t be represented by my interpreter and as such their hexadecimal encodings were returned instead, as seen in the first iteration of this subroutine in the Stage One prototype of the code. I had not anticipated this and as this I couldn’t work with this so I decided to keep it in ASCII format. To get round the potential issue of invalid inputs the ciphertext is a string variable so it can effectively take any character – this means there should be no invalid inputs.

### Decrypt subroutine:

def vernam\_decrypt(y):  
 PlaintextVern = ""  
 vernam\_key = vkey.read().splitlines()  
 for i in range(len(y)):  
 individualletterkey = vernam\_key[i] # this ensures each letter in the ciphertext is matched with its key  
 newletter = chr(y[i] ^ ord(individualletterkey))  
 PlaintextVern += newletter # this adds each newly decrypted character to the Plaintext  
 return PlaintextVern

The output of this function can be seen below



The subroutine has successfully taken the key and ciphertext generated by the encrypt subroutine above and decrypted it, so the plaintext that was originally entered. This has been made possible due to the fact that the print/display problem previously seen in the encrypt subroutine has now been solved, however the decrypt subroutine was established as working perfectly in the last prototype; this test is just to ensure that it works with the data passed to it by the encrypt subroutine.

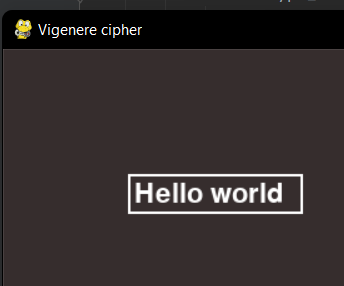
### GUI:

WIDTH, HEIGHT = 900, 500 # this declares the size of the window the GUI will open in  
WIN = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("Vigenere cipher") # this titles the window  
FPS = 60 # this declares the magnitude of the framerate of the window  
TRANSPARENT = (0, 0, 0, 0) # This tuple allows objects on the screen to "disappear" when they are no longer needed  
WHITE = (255, 255, 255)  
BLUE = (0, 0, 255)  
  
  
class Button: # This class will be used as buttons to allow the user to make selections in the program  
  
 def \_\_init\_\_(self, x, y, image):  
 width = image.get\_width()  
 height = image.get\_height()  
 self.image = image  
 self.rect = self.image.get\_rect()  
 self.rect.topleft = (x, y)  
  
 def draw(self): # This method allows the button to draw itself on the screen  
 WIN.blit(self.image, (self.rect.x, self.rect.y))  
  
 def isclicked(self, event):  
 x, y = pygame.mouse.get\_pos()  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 if self.rect.collidepoint(x, y):  
 return 1  
 else:  
 return 0  
  
  
class InputBox:  
  
 def \_\_init\_\_(self, x, y, w, h, text=''):  
 self.rect = pygame.Rect(x, y, w, h)  
 self.color = WHITE  
 self.text = text  
 self.txt\_surface = FONT.render(text, True, self.color)  
 self.active = False  
  
 def handle\_event(self, event):  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 # If the user clicked on the input\_box rect.  
 if self.rect.collidepoint(event.pos):  
 # Toggle the active variable.  
 self.active = not self.active  
 else:  
 self.active = False  
 self.color = WHITE if self.active else BLUE  
 if event.type == pygame.KEYDOWN:  
 if self.active:  
 if event.key == pygame.K\_RETURN:  
 print(self.text)  
 self.text = ''  
 elif event.key == pygame.K\_BACKSPACE:  
 self.text = self.text[:-1]  
 else:  
 self.text += event.unicode  
 # Re-render the text.  
 self.txt\_surface = FONT.render(self.text, True, self.color)  
  
  
 def draw(self, screen):  
 # Blit the text.  
 screen.blit(self.txt\_surface, (self.rect.x + 5, self.rect.y + 5))  
 # Blit the rect.  
 pygame.draw.rect(screen, self.color, self.rect, 2)  
  
  
FONT = pygame.font.Font(None, 32)  
  
encryptimg = pygame.image.load(os.path.join("Assets", "Encrypt\_button.png")) # This line and the next contain the  
# images that will be the program's buttons and allow them to be utilised in the programs  
decryptimg = pygame.image.load(os.path.join("Assets", "Decrypt\_button.png"))  
  
encrypt\_button = Button(100, 150, encryptimg) # These following two lines create the buttons for the two main  
# subroutines  
decrypt\_button = Button(550, 150, decryptimg)  
  
textbox = InputBox(100, 100, 140, 32)  
GREY = (54, 45, 45)  
  
  
def draw\_window(): # this function colours the window  
 WIN.fill(GREY) # this colours the window grey  
 encrypt\_button.draw()  
 decrypt\_button.draw()  
 pygame.display.update() # this updates the colouring of the window the next time the frame cycles  
  
  
def reset\_window():  
 WIN.fill(GREY)  
 textbox.draw(WIN)  
 pygame.display.update()  
  
  
def Coursework(): # the application loop that all pygame programs require to function  
 text = ""  
 clock = pygame.time.Clock() # this creates a variable, that, when a value is assigned to it, it will be the  
 # refresh rate of the application  
 run = True  
 encrypt = False  
 decrypt = False  
 draw\_window()  
 while run: # everything in this loop will only happen when the window is open  
 clock.tick(FPS) # this refreshes the frame every 1/FPS seconds, setting a framerate for the program  
 for event in pygame.event.get(): # the event loop that ensures that any events are properly managed  
 if event.type == pygame.QUIT:  
 run = False # this closes the window if the user quits the program  
 textbox.handle\_event(event)  
 if encrypt\_button.isclicked(event) == 1: # The two following loops create a text box on the screen  
 # when one of the options is chosen  
 reset\_window()  
 encrypt = True  
 if decrypt\_button.isclicked(event):  
 reset\_window()  
 decrypt = True  
 if encrypt:  
 reset\_window()  
 elif decrypt:  
 reset\_window()  
  
 pygame.quit()  
  
  
if \_\_name\_\_ == "\_\_main\_\_": # this ensures that the GUI will open when this specific program is running  
 Coursework()

The output of this can be seen in the images below.



When I click a button (the encrypt button for this example), a textbox is created that the user can type text in.



A text box that the user can type in has successfully been drawn onto the screen. This will be incredibly helpful for the next prototype, as I can encrypt and decrypt the text inputted by the user and they can now see it in a user-friendly environment which is easy to use and understand.

I have completely fulfilled requirements 1 and 5 of my general requirements – my program encrypts and decrypts user-inputted data accurately and without fault using a pseudo-random key generated by the algorithm. Requirement 2 has been partially fulfilled – there is limited functionality to the GUI – buttons can be clicked and text can be entered into the text box, but the outputs of the respective subroutines aren’t outputted to the screen for the user to see.

### Stakeholder feedback:

|  |  |
| --- | --- |
| Stakeholder | Feedback |
| Inioluwa | This prototype works quite well, however the textbox is very small, so larger amounts of text can’t be encrypted – both of the cryptographic functions are also working however so I believe the next prototype will fix these issues |
| Simbo | The buttons and textbox are extremely useful, but the lack of colour of the GUI makes it look a bit lifeless |
| Akin | The layout of the buttons and textbox are very good and allow for easy use of the program, and now the cryptographic functions need to be integrated with it |

### Improvements to be made for next prototype:

* The text the user inputs must be stored in a variable that can be passed as a parameter to the encrypt function
  + The plaintext from the decrypt subroutine should be outputted into the GUI so the user can easily see the text
* The GUI should be made more colourful and vibrant (not really necessary, but a nice touch)
* The text box should expand if the text entered by the user threatens to pass its present boundaries

## Stage Three prototype:

### Encrypt function:

def vigenere(x):  
 ciphertext = []  
 vigkey = [] # this declares the list which will store the key for each individual letter  
 for letter in x:  
 individualletterkey = secrets.choice(string.ascii\_letters) # this generates the key for each letter as a  
 # random character in the ASCII set  
 newletter = ord(letter) ^ ord(individualletterkey) # This transforms the letters of the key and the  
 # plaintext into numbers and performs an XOR operation on the two to create the encoding of the ciphertext  
 # character  
 vigkey.append(individualletterkey) # this stores the key of each letter in a list, so it can be passed on  
 # to the intended recipient for decryption  
 ciphertext.append(newletter) # this adds the newly encrypted letter to the ciphertext  
 for element in vigkey: # this loop ensures that the key for every letter is in a text file that can be passed  
 # on to the intended recipient for them to decrypt the ciphertext  
 vigernkey.write(str(element))  
 vigernkey.write("\n") # Each character in the key will be stored in an individual line  
 for element in ciphertext:  
 ciphertext\_file.write(str(element)) # Each character in the ciphertext will be stored in a line in a file  
 ciphertext\_file.write("\n")  
 global finish\_message  
 finish\_message = "The key is in the file vigernkey.txt, and the ciphertext is in the file cip.txt"  
 return ciphertext, finish\_message

### Decrypt function:

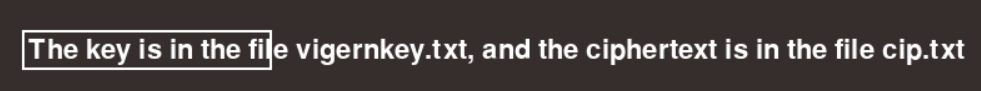
def vigenere\_decrypt(y):  
 y = [int(x) for x in y]  
 Plaintext = "" # this declares the plaintext variable  
 vigkey = vigernkey.read().splitlines()  
 for i in range(len(y)):  
 newletter = chr(y[i] ^ ord(vigkey[i]))  
 Plaintext += newletter # this adds each newly decrypted character to the Plaintext  
 return Plaintext

### GUI:

# GUI creation:  
WIDTH, HEIGHT = 900, 500 # this declares the size of the window the GUI will open in  
WIN = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("Vigenere cipher") # this titles the window  
FPS = 60 # this declares the magnitude of the framerate of the window  
TRANSPARENT = (0, 0, 0, 0) # This tuple allows objects on the screen to "disappear" when they are no longer needed  
WHITE = (255, 255, 255)  
BLUE = (0, 0, 255)  
  
  
class Button: # This class will be used as buttons to allow the user to make selections in the program  
  
 def \_\_init\_\_(self, x, y, image):  
 self.image = image  
 self.rect = self.image.get\_rect()  
 self.rect.topleft = (x, y)  
  
 def draw(self): # This method allows the button to draw itself on the screen  
 WIN.blit(self.image, (self.rect.x, self.rect.y))  
  
 def isclicked(self, event):  
 x, y = pygame.mouse.get\_pos()  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 if self.rect.collidepoint(x, y):  
 return 1  
 else:  
 return 0  
  
  
class InputBox:  
  
 def \_\_init\_\_(self, x, y, w, h, text=''):  
 self.rect = pygame.Rect(x, y, w, h)  
 self.colour = WHITE  
 self.text = text  
 self.txt\_surface = FONT.render(text, True, self.colour)  
 self.active = False  
 self.done = False  
  
 def handle\_event(self, event):  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 # If the user clicked on the input\_box rect.  
 if self.rect.collidepoint(event.pos):  
 # Toggle the active variable.  
 self.active = not self.active  
 else:  
 self.active = False  
 self.colour = WHITE if self.active else BLUE  
 if event.type == pygame.KEYDOWN:  
 if self.active:  
 if event.key == pygame.K\_RETURN:  
 self.done = True  
 elif event.key == pygame.K\_BACKSPACE:  
 self.text = self.text[:-1]  
 else:  
 self.text += event.unicode  
 # Re-render the text.  
 self.txt\_surface = FONT.render(self.text, True, self.colour)  
  
 def update(self):  
 # Resize the box if the text is too long.  
 width = max(200, self.txt\_surface.get\_width() + 10)  
 self.rect.w = width  
  
 def draw(self, screen):  
 # Blit the text.  
 screen.blit(self.txt\_surface, (self.rect.x + 5, self.rect.y + 5))  
 # Blit the rect.  
 pygame.draw.rect(screen, self.colour, self.rect, 2)  
  
  
FONT = pygame.font.Font(None, 32)  
  
encryptimg = pygame.image.load(os.path.join("Assets", "Encrypt\_button.png")) # This line and the next contain the  
# images that will be the program's buttons and allow them to be utilised in the programs  
decryptimg = pygame.image.load(os.path.join("Assets", "Decrypt\_button.png"))  
  
encrypt\_button = Button(100, 150, encryptimg) # These following two lines create the buttons to select the two main  
# subroutines  
decrypt\_button = Button(550, 150, decryptimg)  
  
textbox = InputBox(100, 100, 140, 32)  
GREY = (54, 45, 45)  
  
  
def draw\_window(): # this function colours the window  
 WIN.fill(GREY) # this colours the window grey  
 encrypt\_button.draw()  
 decrypt\_button.draw()  
 pygame.display.update() # this updates the colouring of the window the next time the frame cycles  
  
  
def reset\_window(): # this functions draws a version of the window with a textbox instead of buttons  
 WIN.fill(GREY)  
 textbox.draw(WIN)  
 pygame.display.update()  
  
  
def Coursework(): # the application loop that all pygame programs require to function  
 clock = pygame.time.Clock() # this creates a variable, that, when a value is assigned to it, it will be the  
 # refresh rate of the application  
 run = True  
 encrypt = False  
 decrypt = False  
 finished = False  
 draw\_window()  
 while run: # everything in this loop will only happen when the window is open  
 clock.tick(FPS) # this refreshes the frame every 1/FPS seconds, setting a framerate for the program  
 for event in pygame.event.get(): # the event loop that ensures that any events are properly managed  
 if event.type == pygame.QUIT:  
 run = False # this closes the window if the user quits the program  
 textbox.handle\_event(event)  
 if encrypt\_button.isclicked(event) == 1: # The two following loops create a text box on the screen  
 # when one of the options is chosen  
 reset\_window()  
 encrypt = True  
 if decrypt\_button.isclicked(event):  
 reset\_window()  
 decrypt = True  
  
 if finished:  
 reset\_window()  
 if encrypt:  
 textbox.txt\_surface = FONT.render(finish\_message, True, textbox.colour)  
 encrypt = False  
 textbox.active = False  
 textbox.done = False  
 elif decrypt:  
 textbox.txt\_surface = FONT.render(decr\_text, True, textbox.colour)  
 decrypt = False  
 textbox.active = False  
 textbox.done = False  
  
  
 if encrypt: # the following code encrypts the user input and outputs a message onto the screen  
 ciphertext\_file.truncate(0)  
 vigernkey.truncate(0)  
 vigernkey.seek(0)  
 ciphertext\_file.seek(0)  
 reset\_window()  
 textbox.update()  
 if textbox.done:  
 encr\_text = vigenere(textbox.text)  
 print(textbox.text)  
 finished = True  
  
 elif decrypt: # the following code decrypts the contents of cip.txt and outputs the result onto the screen  
 reset\_window()  
 textbox.update()  
 if textbox.done:  
 ciphtext = ciphertext\_file.read().splitlines()  
 decr\_text = vigenere\_decrypt(ciphtext)  
 textbox.txt\_surface = FONT.render(decr\_text, True, textbox.colour)  
 print(textbox.text)  
 finished = True  
  
 pygame.quit()  
  
  
if \_\_name\_\_ == "\_\_main\_\_": # this ensures that the GUI will open when this specific program is running  
 Coursework()

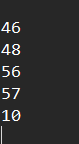
### Output of the encrypt function to the GUI:





If we look in the two files we can see the ciphertext and the key:

The ciphertext has been encoded with numbers and can be seen below:



And below we can see the key:



### Output of the decrypt function to the GUI:

If we perform a decryption on the original text entered to the encrypt function, we should receive the text we typed initially, “Hello”.



As we can see here, both the encrypt and decrypt functions are correctly outputting their respective outputs to the GUI due to the edits made to the code.

If we look to the [General Requirements](#_General_Requirements) outlined in the Analysis section, we can see that they have all been fulfilled to a high standard:

1. The key used is generated by the computer – user input is not taken for the key in any way or form
2. The GUI is now functional; the user can type their text onto it and they are able to see the text outputted by the program on the GUI as well; the colouring is a bit bland, but that can be changed at a later date and has no effect on the functionality of the GUI in any way or form
3. Requirement 3 has also been fulfilled, though it is a bit redundant as in the end I didn’t use a shift cipher. At the stage of development where the general requirements were written, my plan was to use a Caesar cipher (a shift cipher) where there is a randomly generated key for each letter. I still used the second part of that idea, but I ended up using XOR operations between corresponding letters in the key and plaintext. Nevertheless, the cipher doesn’t shift by one or two letters, so requirement 3 has been fulfilled
4. We can see that the key has been written to a text file by the function, so this requirement has been completely fulfilled by the code
5. We can also see that the program can effectively decrypt text that was inputted to the decrypt function (so stored in cip.txt) when given the key (which would be in vigernkey.txt). The requirements do not ask for the ability to decrypt without the key, and I have outlined elsewhere why that would be impractical; as such this requirement has been fulfilled

|  |  |
| --- | --- |
| Stakeholder | Feedback |
| Inioluwa | This prototype is very functional; the text is encrypted and decrypted accurately and with no issues, and the interface is very good for the most part, however the program can feel a little unpolished at times |
| Simbo | This is an extremely useful program that works very well, however the interface makes it look lifeless; as if it’s a program used for Top Secret documents and not odd bits of data users want to be a bit more secure |
| Akin | This program works very well and is easy to use for the most part. However, for commercial use some more instructions will need to be provided… maybe a README file? |

# Evaluation

## Final testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Item | Test Number | Type | Test Input | Justification | Expected Outcome | Actual Outcome |
| Buttons | 1 | Valid | Button clicked | To check that the buttons respond to being pressed | The screen changes to display the textbox when either button is pressed | This test was **PASSED** |
| 2 | Invalid | Click elsewhere on the screen | To ensure the program only responds to the button being clicked | The screen doesn’t change at all | This test was **PASSED** |
| Text box | 3 | Valid | Text being typed is displayed | To ensure the user is able to see the text they’re entering | The letters assigned to the keys being pressed are displayed on the screen | This test was **PASSED** |
| Encrypt function | 3 | Valid | Type any combination of characters | The ciphertext variable should be treated at string, so any character should be acceptable | The function should take the string and encrypt it | This test was **PASSED** |
| Decrypt function | 4 | Valid | Encrypt any combination of characters using the encrypt function | The decrypt function must be able to decrypt anything the encrypt function can take as input | The original plaintext should be displayed on the screen | This test was **PASSED** |

As there were no failed tests, no solutions are needed.

## Usability testing

|  |  |  |
| --- | --- | --- |
| Feature | Evidence | Explanation |
| Buttons |  | The use of buttons allows for the user to select exactly which function of the program they would like to use and avoids any possible issues that could arise from taking their choice as a string |
| Textbox |  | The use of a textbox allows for multiple features to enhance usability of the program; the user is able to type their inputted text into the box and information that needs to be conveyed to the user can be done via the text box |
| Colour scheme |  | A muted colour scheme with a dark grey background has been chosen intentionally. Darker coloured interfaces (not necessarily darker in terms of brightness) tend to strain the eyes less compared to interfaces with bright, bold colours. While those do have a time and place, being a sufferer of migraines that are exacerbated by eye strain myself, I have decided to minimise the eye strain caused by my program by having a darker coloured interface |

## Success criteria review

|  |  |  |
| --- | --- | --- |
| Criteria | Evidence of success | Further comments |
| The program should securely encrypt user’s data |  | The python code shown generates the key for each letter as a random letter from a list of every character in the English alphabet (so 52 possible characters). This means the number of possible keys (and plaintexts corresponding to each ciphertext) is 52n, where n is the length of the original input, key and ciphertext. This means a 5 character ciphertext will have 380 million possible plaintexts. This is not feasible to decrypt by brute force with anything but a quantum computer . In addition to all of the above, the referenced line of code is used in a loop that executes “for letter in x” where x is the input to the encrypt function, so each letter will have a different key |
| The program should have an engaging interface |  | The interface of the program is coloured dark grey for reasons outlined in the previous sections. The interface, while not as engaging as it could be, is still fairly engaging. The textbox is clearly visible |
|  |  |  |