EECE 455

**Project Report**

Classical Encryption Ciphers

horizontal line

# 

# 

**TEAM MEMBERS:** **Sarjoun Radiyeh**

**Antoine Abu Faysal**

**Wael Dgheim**

**Adel El Kadi**

# Table of contents:

[1](#_Toc152146761)

[Table of contents: 2](#_Toc152146762)

[Table of Figures: 6](#_Toc152146763)

[Abstract: 11](#_Toc152146764)

[Introduction: 12](#_Toc152146765)

[What are Classical Encryption Ciphers? 12](#_Toc152146766)

[Motivation: 12](#_Toc152146767)

[Literature Review: 13](#_Toc152146768)

[Affine Cipher: 13](#_Toc152146769)

[How It Works: 13](#_Toc152146770)

[Security Implications: 14](#_Toc152146771)

[Cracking an Affine Cipher: 15](#_Toc152146772)

[Vigenère Cipher: 16](#_Toc152146773)

[How does it Work? 16](#_Toc152146774)

[Security Implications: 17](#_Toc152146775)

[Monoalphabetic Cipher: 18](#_Toc152146776)

[How does it Work? 18](#_Toc152146777)

[Security Implications: 18](#_Toc152146778)

[Playfair Cipher: 19](#_Toc152146779)

[How does it Work? 19](#_Toc152146780)

[Security Implications: 20](#_Toc152146781)

[Hill Cipher: 20](#_Toc152146782)

[How does it Work? 20](#_Toc152146783)

[Security Implications: 21](#_Toc152146784)

[Extended Euclid: 21](#_Toc152146785)

[Algorithm Overview: 22](#_Toc152146786)

[Applications: 22](#_Toc152146787)

[Security Implications: 22](#_Toc152146788)

[Methodology & Architecture 22](#_Toc152146789)

[Graphical User Interface (GUI): 23](#_Toc152146790)

[Affine Cipher: 24](#_Toc152146791)

[General Encryption and Decryption: 25](#_Toc152146792)

[Crack Using Letter Frequency: 27](#_Toc152146793)

[Crack Using Top Brute Force: 29](#_Toc152146794)

[Vigenère Cipher: 31](#_Toc152146795)

[General Encryption and Decryption: 31](#_Toc152146796)

[Monoalphabetic Cipher: 34](#_Toc152146797)

[General Encryption and Decryption: 34](#_Toc152146798)

[Playfair Cipher: 36](#_Toc152146799)

[Backend: 36](#_Toc152146800)

[Frontend 40](#_Toc152146801)

[Hill Cipher: 44](#_Toc152146802)

[HTML/CSS Interface: 44](#_Toc152146803)

[Django Forms: 45](#_Toc152146804)

[Main Page: 47](#_Toc152146805)

[Extended Euclidean Algorithm: 48](#_Toc152146806)

[HTML Interface (extendedeuclid.html): 48](#_Toc152146807)

[Backend Processing (views.py): 49](#_Toc152146808)

[Form Design (forms.py): 49](#_Toc152146809)

[Testing & Results: 50](#_Toc152146810)

[Affine Cipher: 50](#_Toc152146811)

[General Encryption and Decryption: 50](#_Toc152146812)

[Cracking using Letter Frequency: 52](#_Toc152146813)

[Cracking using Top Brute Force: 55](#_Toc152146814)

[Vigenère: 56](#_Toc152146815)

[General Encryption and Decryption: 56](#_Toc152146816)

[Monoalphabetic: 58](#_Toc152146817)

[General Encryption and Decryption: 58](#_Toc152146818)

[Playfair Cipher 59](#_Toc152146819)

[General Encryption and Decryption: 59](#_Toc152146820)

[Hill Cipher: 61](#_Toc152146821)

[Extended Euclidean Algorithm: 62](#_Toc152146822)

[Conclusion: 64](#_Toc152146823)

[References 65](#_Toc152146824)

# Table of Figures:

[Figure 1: Webpage For Home Page 24](#_Toc152186912)

[Figure 2: affine.html Snippet for General Encryption/Decryption for Affine 25](#_Toc152186913)

[Figure 3: forms.py Section for General Encryption/Decryption for Affine 25](#_Toc152186914)

[Figure 4: Webpage for General Encryption/Decryption for Affine 26](#_Toc152186915)

[Figure 5: views.py Section for General Encryption for Affine 26](#_Toc152186916)

[Figure 6: views.py Section for General Decryption for Affine 27](#_Toc152186917)

[Figure 7: crackaffine.html Snippet for Cracking using Letter Frequency Affine Cipher 27](#_Toc152186918)

[Figure 8: forms.py Section for Cracking using Letter Frequency Affine Cipher 28](#_Toc152186919)

[Figure 9: Webpage for Cracking using Letter Frequency Affine Cipher 28](#_Toc152186920)

[Figure 10: views.py Section for Cracking using Letter Frequency Affine Cipher 29](#_Toc152186921)

[Figure 11: topbruteforceaffine.html Snippet for Top Brute Force Affine Cipher 29](#_Toc152186922)

[Figure 12: forms.py Section for Cracking using Top Brute Force Affine Cipher 30](#_Toc152186923)

[Figure 13: Webpage for Cracking using Top Brute Force Affine Cipher 30](#_Toc152186924)

[Figure 14: views.py Section for Cracking using Top Brute Force Affine Cipher 30](#_Toc152186925)

[Figure 15: vigenere.html Snippet for General Encryption/Decryption for Vigenère 31](#_Toc152186926)

[Figure 16: forms.py Section for General Encryption/Decryption for Vigenère. 32](#_Toc152186927)

[Figure 17: Webpage for General Encryption/Decryption for Vigenère 32](#_Toc152186928)

[Figure 18: views.py section for General Encryption for Vigenère 33](#_Toc152186929)

[Figure 19: views.py Section for General Decryption for Vigenère 33](#_Toc152186930)

[Figure 20: monoalphabetic.html Snippet for General Encryption/Decryption for Monoalphabetic 34](#_Toc152186931)

[Figure 21: forms.py Section for General Encryption/Decryption for Monoalphabetic 34](#_Toc152186932)

[Figure 22: Webpage for General Encryption/Decryption for Monoalphabetic 35](#_Toc152186933)

[Figure 23: views.py Section for General Encryption for 35](#_Toc152186934)

[Figure 24: views.py Section for General Decryption for Monoalphabetic 36](#_Toc152186935)

[Figure 25: Key Generation Code for Playfair Cipher 36](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186936)

[Figure 26: Forming Digraphs in Playfair 37](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186937)

[Figure 27: Decryption function of Playfair 38](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186938)

[Figure 28: Encryption Function of Playfair 38](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186939)

[Figure 29: Calling the Function According to the Node 39](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186940)

[Figure 30: forms.py Section of the Playfair Cipher 39](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186941)

[Figure 31: views.py Section of the Playfair Cipher 40](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186942)

[Figure 32: Inheriting the Base Style 40](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186943)

[Figure 33: Code for Description 41](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186944)

[Figure 34: Description and Title on the Webpage 41](#_Toc152186945)

[Figure 35: Code to Display the Forms 41](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186946)

[Figure 36: Forms on the Webpage 41](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186947)

[Figure 37: Error when Key Contains Non-letters 42](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186948)

[Figure 38: Results and Table Code 42](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186949)

[Figure 39: Styles for the table 43](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186950)

[Figure 40: Display of Result and Key Matrix on Webpage 43](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186951)

[Figure 41: hillcipher.html Snippet for General Encryption/Decryption for Hill Cipher 44](#_Toc152186952)

[Figure 42: forms.py section for General Encryption/Decryption for Hill Cipher 45](#_Toc152186953)

[Figure 43: views.py section for General Encryption/Decryption for Hill Cipher 45](#_Toc152186954)

[Figure 44: Decryption function of Hill 46](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186955)

[Figure 45: Encryption function of Hill 46](#_Toc152186956)

[Figure 46: Webpage for General Encryption/Decryption for Hill 47](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186957)

[Figure 47: extendedeuclid.html Snippet for Extended Euclid 48](#_Toc152186958)

[Figure 48: views.py section for Extended Euclid 49](#_Toc152186959)

[Figure 49: forms.py section for Extended Euclid 49](#_Toc152186960)

[Figure 50: First Case using General Encryption and Decryption Affine 50](#_Toc152186961)

[Figure 51: Second Case using General Encryption and Decryption Affine 50](#_Toc152186962)

[Figure 52: Third Case using General Encryption and Decryption Affine 51](#_Toc152186963)

[Figure 53: Fourth Case using General Encryption and Decryption Affine 51](#_Toc152186964)

[Figure 54: First Case using Crack using Letter Frequency 52](#_Toc152186965)

[Figure 55: Second Case using Crack using Letter Frequency 52](#_Toc152186966)

[Figure 56: Third Case using Crack using Letter Frequency 53](#_Toc152186967)

[Figure 57: Fourth Case using Crack using Letter Frequency 53](#_Toc152186968)

[Figure 58: Fifth Case using Crack using Letter Frequency 54](#_Toc152186969)

[Figure 59: Sixth Case using Crack using Letter Frequency 54](#_Toc152186970)

[Figure 60: First Case using Crack using Top Brute Force 55](#_Toc152186971)

[Figure 61: Second Case using Crack using Top Brute Force 55](#_Toc152186972)

[Figure 62: Third Case using Crack using Top Brute Force 56](#_Toc152186973)

[Figure 63: First Case using General Decryption Vigenère 56](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186974)

[Figure 64: First Case using General Encryption Vigenère 56](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186975)

[Figure 65: Second Case using General Encryption Vigenère 57](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186976)

[Figure 66: Second Case using General Decryption Vignere 57](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186977)

[Figure 67: Third Case using General Encryption Vigenere 57](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186978)

[Figure 68: Third Case using General Decryption Vigenere 57](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186979)

[Figure 69: First Case using General Encryption and Decryption Monoalphabetic 58](#_Toc152186980)

[Figure 70: Second Case using General Encryption and Decryption Monoalphabetic 58](#_Toc152186981)

[Figure 71: Third Case using General Encryption and Decryption Monoalphabetic 58](#_Toc152186982)

[Figure 72: Fourth Case using General Encryption and Decryption Monoalphabetic 59](#_Toc152186983)

[Figure 73: First test of the Playfair cipher 59](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186984)

[Figure 74: Second test of the Playfair Cipher 60](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186985)

[Figure 75: Third test of the Playfair Cipher 60](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186986)

[Figure 76: Fourth test of the Playfair Cipher 60](file:///C:\Users\xenon\Downloads\12.docx#_Toc152186987)

[Figure 77: First Case using 3x3 key matrix for Hill Cipher 61](#_Toc152186988)

[Figure 78: Second Case using 3x3 key matrix for Hill Cipher 61](#_Toc152186989)

[Figure 79: Third Case for Hill Cipher 62](#_Toc152186990)

[Figure 80: First Case for Extended Euclid 62](#_Toc152186991)

[Figure 81: Second Case for Extended Euclid 63](#_Toc152186992)

[Figure 82: Third Case for Extended Euclid 63](#_Toc152186993)

# 

# Abstract:

This project endeavors to explore classical encryption techniques through the creation of an interactive web-based platform using Django and Python. The platform integrates various classical ciphers, including the Affine Cipher, Vigenère Cipher, Monoalphabetic Cipher, Playfair Cipher, Hill Cipher, and the Extended Euclidean Algorithm. Each cipher is meticulously implemented with backend Python logic and fronted by a Django interface, showcasing their functionalities and mathematical underpinnings.

The primary focus of this project lies in providing users with an interactive and educational experience. Through a user-friendly web interface, users can input texts, manipulate key parameters, and observe the encryption or decryption outputs. The seamless integration of Django templates, forms, and backend Python scripts ensures a coherent and efficient workflow, enabling the processing and visualization of results.

The exploration of various ciphers and algorithms within a unified platform not only serves as a practical demonstration of classical encryption techniques but also aims to educate users on their principles and applications. The project also underscores the versatility and adaptability of Django and Python in creating an interactive and educational platform for classical cryptography.

# Introduction:

## What are Classical Encryption Ciphers?

Classical cipher techniques are methods of encrypting or encoding messages that have been used historically for secure communication. These techniques were prevalent before the advent of modern computer-based encryption methods. Some prevalent techniques include: Affine Cipher, Hill cipher, Playfair cipher, Monoalphabetic cipher, Extended Euclid and Vigenère Cipher.

## Motivation:

For our project we will explore the specifics of classical encryption techniques. Our goal is to use these strategies in a concrete and useful way in addition to understanding them conceptually.

We decided to use Django, a high-level Python web framework that promotes quick development and simple, practical design, as the framework of choice for our project. In addition we wanted to turn our encryption algorithms into an intuitive cloud-hosted user experience. We focused on Azure, a cloud platform known for its scalability and consistency.

We divided the work between front and backend. For our front end implementation we used HTML and CSS in order to design our website and for our backend implementation we used Python to implement the encryption techniques.

# Literature Review:

In this literature review, we take a closer look at classical encryption techniques from the traditional substitution ciphers to those using transposition and permutation methods. Our goal is to understand their operational methods, shedding light on both their lasting significance and potential vulnerabilities. This exploration offers a comprehensive perspective on where these methods stand in the larger landscape of information security.

## Affine Cipher:

The **Affine Cipher** is a classic encryption technique that combines elements of both substitution and linear algebra. It was invented by the ancient mathematician and military engineer, Leon Battista Alberti.

### How It Works:

The **Affine Cipher** works by mapping each letter in the alphabet to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter.

**General Encryption Formula: *E*(*x*)=(*ax*+*b*)mod26**

Where :

* ***E(x)*** is the encrypted letter
* ***x*** is the numerical value of the letter
* ***a*** is a constant multiplier
* ***b*** is a constant additive term
* **26** represents the number of letters in the alphabet

**General Decryption Formula**: ***D(y)=(a^−1)(y−b)* mod26**

Where:

* ***D(y)*** is the decrypted letter
* ***y*** is the numerical value of the encrypted letter
* ***a^-1*** denotes the modular multiplicative inverse of a
* ***b*** remains the constant additive term.
* **26** represents the number of letters in the alphabet

### Security Implications:

While the **Affine Cipher** boasts **simplicity** and historical significance, a more granular examination of its security unveils inherent weaknesses, notably inherited from its status as a **monoalphabetic substitution cipher**. This connection is evident when considering the Affine Cipher's direct link to the Caesar cipher, where the encrypting function **simplifies to a linear shift.**

In the realm of English message encryption, the **Affine Cipher** manifests as a monoalphabetic substitution cipher, amplifying its **vulnerability**. The total number of non-trivial affine ciphers, excluding the **26** trivial Caesar ciphers, amounts to **286**. This calculation arises from the 12 **coprime** values with 26 (representing possible values of ***a***), each coupled with 26 different addition shifts (the value of ***b***). Consequently, the system offers a mere 312 possible keys. This limited variety raises significant concerns, aligning with **Kerckhoffs' Principle**, which asserts that the security of a cryptographic system should not rely on the secrecy of **the** **algorithm** but on the secrecy of **the** **key** [1].

A critical flaw surfaces in the Affine Cipher's vulnerability to **frequency analysis, brute force attacks, and other cryptanalytic methods**. The weakness is particularly pronounced when a cryptanalyst discerns the plaintext of two ciphertext characters, allowing for the derivation of the key by solving a simultaneous equation. Given the **coprimality** of ***a*** and **26**, this vulnerability facilitates the swift elimination of **"false"** keys, thereby compromising the system's security.

Furthermore, the resemblance between the transformation employed in affine ciphers and that in linear congruential generators, a type of **pseudorandom number generator**, raises concerns. Similar to the **Affine Cipher**, the linear congruential generator lacks cryptographic security due to shared vulnerabilities, underlining the need for caution in relying on the **Affine Cipher** for applications requiring robust encryption in contemporary contexts. As we navigate the intricacies of classical encryption techniques, these nuanced vulnerabilities guide our exploration towards more resilient cryptographic strategies [2].

### Cracking an Affine Cipher:

The **Affine Cipher**, a simple yet historically significant encryption technique. Despite its straightforward nature, this cipher harbors inherent vulnerabilities that make it susceptible to cracking. Over time, various methods have been developed to break the Affine Cipher, each exploiting its mathematical structure and the statistical properties of natural languages.

#### Method 1: Frequency Analysis and Cryptanalysis:

This approach leverages the statistical characteristics of natural languages, such as English, where certain letters appear more frequently than others. By analyzing the **frequency of letters** within the ciphertext, cryptanalysts can make educated **guesses** about the most frequent letters in the plaintext. Once these **probable** plaintext letters are identified, they can be used to derive the encryption key using the **Affine Cipher's** mathematical properties.

While this method offers a relatively straightforward approach to cracking the **Affine Cipher**, its effectiveness is heavily **dependent on the quality and length of the ciphertext**. Short or noisy ciphertexts **may not provide sufficient information** to accurately determine letter frequencies, **hindering the accuracy** of the key derivation process [3].

#### Method 2: Brute Force Attacks:

**Brute-force attacks** involve systematically testing all possible combinations of encryption keys until the correct one is found. While computationally **demanding**, this method can be particularly **effective for shorter ciphertexts** where the number of possible keys is manageable.

To enhance the efficiency of brute-force attacks, a technique called "top brute force" can be employed. This technique focuses on displaying the most likely keys first, prioritizing those that correspond to common words or phrases in the target language. By prioritizing these high-probability keys, the top brute force method can significantly reduce the inconvenience of looking through all outputs for a reasonable result, however it also becomes **dependent on the quality** due to the fact that we are relying on common words to appear [4].

#### Comparative Analysis:

The method using frequency analysis and cryptanalysis offers a simpler and more analytical approach to cracking the **Affine Cipher**. It does not require extensive computational resources and can be implemented with relative ease. However, its effectiveness depends heavily on the quality of the ciphertext and the accuracy of the initial assumptions about letter frequencies.

**Top brute force**, on the other hand, is a more methodical and exhaustive approach. While being relatively more computationally demanding, it can be highly effective, especially for **shorter ciphertexts** and with the assistance of common word or phrase lists. This method does not rely on assumptions about letter frequencies, making it more robust in certain scenarios.

#### Conclusion:

The choice between the method using frequency analysis and cryptanalysis and top brute force depends on the specific circumstances and the available resources. For **longer ciphertexts** or situations where **computational resources are limited**, the method using frequency analysis and cryptanalysis may provide a more practical solution. However, for **shorter ciphertexts** or when a **more definitive solution** is required, top brute force can be a more effective approach.

In both cases, understanding the mathematical properties of the **Affine Cipher** and the statistical characteristics of natural languages plays a crucial role in successfully cracking the cipher.

## Vigenère Cipher:

The Vigenère cipher was invented by the French cryptographer Blaise de Vigenère in the 16th century. It is a type of substitution cipher where each letter in the plaintext is shifted by a variable amount based on a keyword.

### How does it Work?

The **Vigenère cipher** enhances traditional substitution methods by introducing a keyword. The user selects a keyword, repeats it to match the message length, and encrypts the plaintext by shifting each letter based on the corresponding letter in the keyword.

**General Encryption Formula:** Ci = (Pi+Ki) mod 26

Where :

* Ci is the ith cipher text letter
* Pi is the ith plain text letter
* Ki is the ith letter in the keyword
* 26 represents the number of letters in the alphabet

**General Decryption Formula**: Pi = (Ci-Ki+26) mod 26

Where:

* Ci is the ith cipher text letter
* Pi is the ith plain text letter
* Ki is the ith letter in the keyword
* 26 represents the number of letters in the alphabet

### Security Implications:

The **Vigenère Cipher** is considered relatively insecure as it is compromised by several vulnerabilities that can be exploited by various cryptanalysis techniques; this is due to the fact that it can only use 26 keys, which makes breaking the code fairly easy. Anyone who wants to break the code needs to only try each alphabet as the priming key before they make a sensible message out of it. This can be done manually in a short time ([Intellipaat](https://intellipaat.com/blog/vigenere-cipher/#:~:text=The%20autokey%20method%20for%20Vigenere,manually%20in%20a%20short%20time.)).If the keyword is shorter than the message, it repeats. This repetition introduces patterns in the ciphertext that cryptanalysts can analyze. The longer the message, the more likely it is that repeating patterns emerge. These patterns can be exploited to deduce the length of the keyword, undermining the security of the cipher.

## Monoalphabetic Cipher:

Monoalphabetic ciphers are a type of substitution cipher where each letter in the plaintext is consistently replaced with another letter in the ciphertext. The most well-known example of a monoalphabetic cipher is the Caesar cipher, where each letter is shifted a fixed number of positions in the alphabet.

### How does it Work?

The **Monoalphabetic cipher** enhances traditional substitution methods as substitution is one to one meaning that each letter in plaintext corresponds to a letter in ciphertext.

**General Encryption Formula:** Ci=(Pi+shift)mod26

Where :

* Ci is the ith cipher text letter
* Pi is the ith plain text letter
* Shift is the amount the letter is shifted by (ex: shift of 3, 'A' becomes 'D')
* 26 represents the number of letters in the alphabet

**General Decryption Formula**: Pi=(Ci−shift)mod26

Where:

* Ci is the ith cipher text letter
* Pi is the ith plain text letter
* Shift is the amount the letter is shifted by
* 26 represents the number of letters in the alphabet

### Security Implications:

The monoalphabetic cipher is considered vulnerable as a cryptanalyst can use a statistical attack in order to decrypt the ciphertext. Statistical attacks include measuring the frequency distribution for characters, comparing those with the same statistics for English [5]. .Monoalphabetic ciphers often have a limited key space, especially in simple variations like the Caesar cipher. With a small number of possible keys, these ciphers are susceptible to brute-force attacks, where an attacker can try all possible keys to decrypt the message.

## Playfair Cipher:

In the history of cryptology, the Playfair cipher was the first to encrypt a pair of letters. The secret telegraphy cipher was created by Wheatstone, but it bears the name of his friend Lord Playfair, the first Baron Playfair of St. Andrews, who encouraged the use of it.

### How does it Work?

The **Playfair cipher** operates by encrypting pairs of letters rather than individual letters. It employs a 5x5 matrix and uses a keyword to fill in the matrix excluding duplicate letters; the remaining letters of the alphabet are used to fill the matrix.

#### General Encryption Process:

**1.Pairing of letters:**  Plaintext is grouped into pairs of letters

**Note:** if a pair is a repeated letter, insert filler like 'X’. For example: balloon is treated as ba lx lo on

**2.Matrix rules to encrypt:**

1. If both letters fall in the same row, replace each with letter to right

2. If both letters fall in the same column, replace each with the letter below it

3. Each letter is replaced by the letter in the same row and in the column of the other letter of the pair

#### General Decryption Process:

**1.Pairing of letters:**  Ciphertext is grouped into pairs of letters

**2.Matrix rules to Decrypt:** Apply the reverse of encryption rules to decrypt the matrix.

### Security Implications:

The Playfair cipher represents a notable improvement over monoalphabetic ciphers due to its use of digrams, resulting in a larger key space of 676 possible combinations. Analyzing a Playfair-encrypted message requires a frequency table with 676 entries, significantly complicating frequency analysis compared to the 26 entries needed for monoalphabetic ciphers.

However, Playfair is vulnerable. If an adversary has a sufficiently large ciphertext sample size of a few hundred letters), they might deduce patterns in the digram frequencies and guess the key. With modern computing power, numerical cryptanalysis tools can significantly reduce the time required to break the cipher.

## Hill Cipher:

The **Hill Cipher** was Invented by Lester S. Hill in 1929, it was the first polygraphic cipher in which it was practical though barely to operate on more than three symbols at once.

### How does it Work?

The **Hill cipher** encrypts messages by transforming blocks of letters into numerical vectors. These vectors are then multiplied by a specific matrix, known as the encryption key matrix.

#### General Encryption Process:

**1.Message Preparation**

The plaintext is divided into blocks, and each block is mapped to its numerical representation (ex: 'A' is 0, 'B' is 1, etc.).

**2.Matrix Multiplication:**

Each block of letters is treated as a vector and multiplied by the key matrix. The result is a new vector.

**3.Modular Arithmetic:**

The elements of the resulting vector are reduced modulo 26

**4. Convert to ciphertext:**

Numbers that are outputted represent a certain cipher text.

#### General Decryption Process:

**1.Matrix Inversion:**

Inversely multiply the ciphertext vector by the inverse of the key matrix.

**2.Modular Arithmetic:**

The elements of the resulting vector are reduced modulo 26

**3. Convert to Plaintext:**

Numbers that are computed represent a certain Plain text.

### Security Implications:

The security of the Hill Cipher hinges on several critical aspects. Primarily, its vulnerability to known-plaintext and chosen-plaintext attacks poses significant concerns, particularly when adversaries have access to ciphertexts and corresponding plaintexts. Moreover, the cipher's susceptibility to algebraic attacks, dependent on weaknesses in the key matrix properties, amplifies potential security risks. The strength of the Hill Cipher is intrinsically tied to the properties of the key matrix, including its size, invertibility, and the modulus used in modulo arithmetic operations. Successful cryptanalysis often relies on exploiting these vulnerabilities, necessitating careful consideration of key management and selection to mitigate risks associated with these known attack vectors.

## Extended Euclid:

The Extended Euclidean Algorithm (EEA) stands as a pivotal algorithm within number theory and cryptography, primarily utilized to calculate the modular multiplicative inverse of two integers and solve Diophantine equations. Originating as an extension of Euclid's Algorithm for computing the greatest common divisor (GCD), the EEA's significance in modern cryptography was highlighted by the work of Carl Friedrich Gauss and Leonhard Euler in the 18th century.

### Algorithm Overview:

1. GCD Computation: The EEA determines the GCD of two integers by iteratively applying subtraction and modulus operations until reaching a remainder of zero, echoing the principles of Euclid's original algorithm.

2. Bézout's Identity: Simultaneously while computing the GCD, the algorithm derives the coefficients (x and y) of Bézout's identity (ax + by = gcd(a, b)) via a backward substitution process.

3. Modular Inverse Calculation: Capitalizing on Bézout's coefficients, the EEA efficiently computes the modular inverse of 'a' modulo 'm' when 'a' and 'm' are coprime (gcd(a, m) = 1).

### Applications:

- Cryptography: Integral to cryptographic protocols, the EEA plays a pivotal role in RSA encryption and decryption, facilitating key generation and modular arithmetic operations.

- Error Detection and Correction: Finds practical utility in various error-detection schemes like Reed-Solomon codes and Cyclic Redundancy Checks (CRCs).

### Security Implications:

- RSA Security: The EEA's role in efficiently calculating modular inverses forms the cornerstone of RSA's security, emphasizing its significance in ensuring secure cryptographic systems.

# Methodology & Architecture

In this section, we methodically examine the backend and frontend components of our classical encryption project. The **backend** analysis, encompassing **forms.py**, **views.py**, and **urls.py** within the **architectural pattern**, delves into the implementation of cryptographic algorithms and data structures. Concurrently, the **frontend** exploration highlights the user interface within the **Django web framework** on **Azure** along with the **HTML** and **CSS files** used to create it. By distinctively addressing each aspect, we provide a precise and comprehensive understanding of our project's architecture, facilitating a nuanced evaluation of cryptographic robustness and user-centric design.

## Graphical User Interface (GUI):

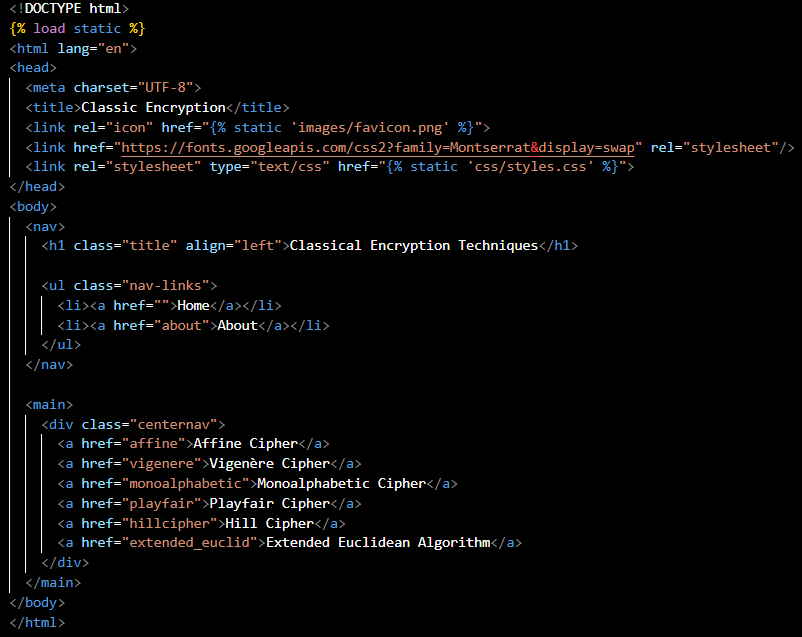


Figure 1: home.html Snippet

The HTML file ‘home.html’ establishes the primary interface of the web application. It defines the structure for the main page, comprising key elements:

* The document type and language are specified, ensuring proper rendering and interpretation in HTML.
* A navigation bar is featured at the top, presenting a header titled "Classical Encryption Techniques" and navigation links, including "Home" and "About".
* The main content section includes a centralized navigation panel. This panel contains clickable links to different encryption methods, such as "Affine Cipher," "Vigenère Cipher," "Monoalphabetic Cipher," "Playfair Cipher," "Hill Cipher," and the "Extended Euclidean Algorithm."

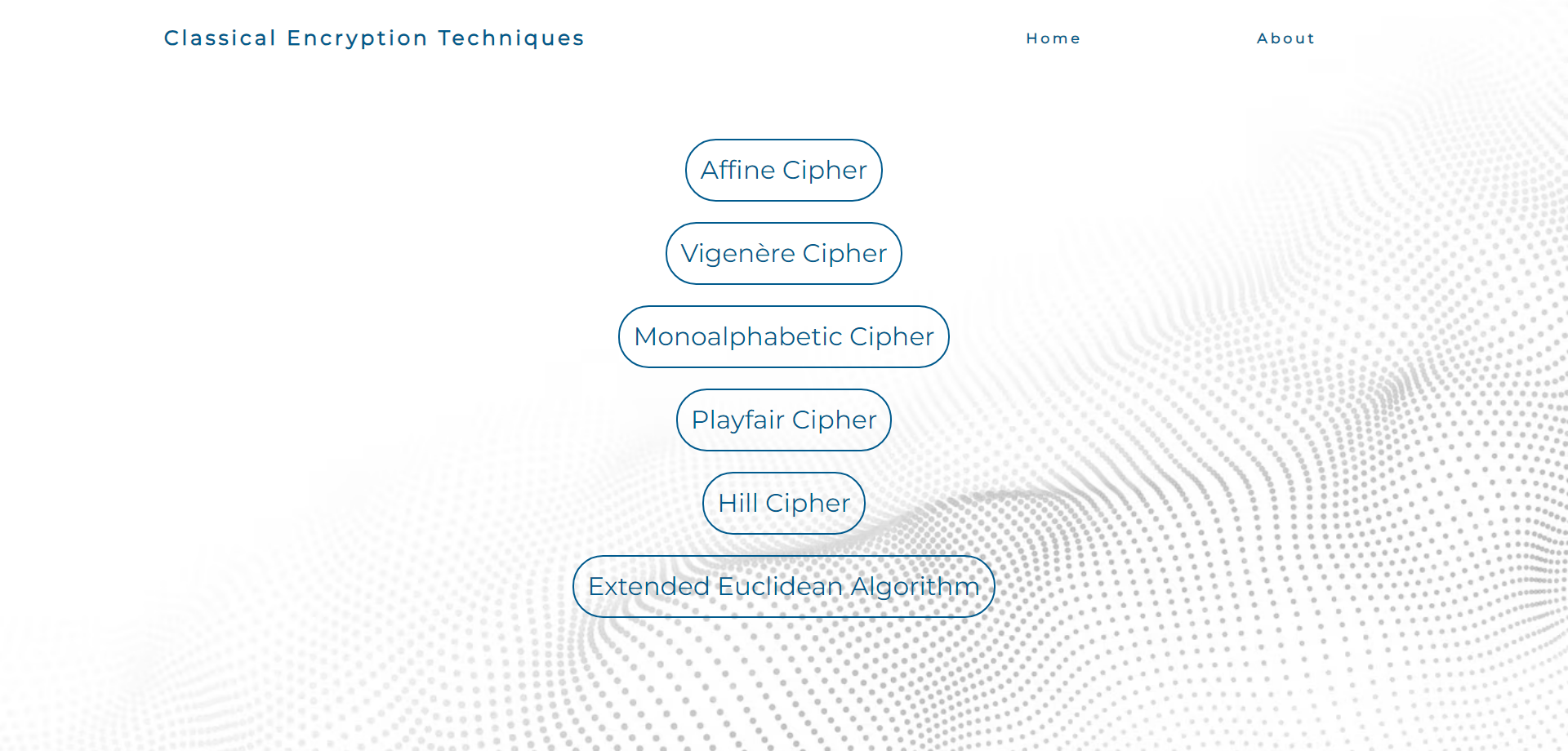


Figure 1: Webpage For Home Page

This layout enables users to navigate seamlessly between various encryption techniques, providing them direct access to specific functionalities within the application. The design emphasizes clarity and user-friendly interaction, presenting a straightforward entry point for users to engage with the diverse encryption techniques offered by the application. The integration of font styles from Google Fonts and a custom stylesheet potentially enhances the visual appeal and layout of the page. Overall, this HTML code serves as the initial gateway for users to explore and utilize different classic encryption techniques provided by the web application.

## Affine Cipher:

The Affine Cipher section of our code required four distinct functionalities: encryption, decryption, and two cracking methods—letter frequency and top brute force. Each of these functionalities demanded a separate backend and frontend implementation.

### General Encryption and Decryption:

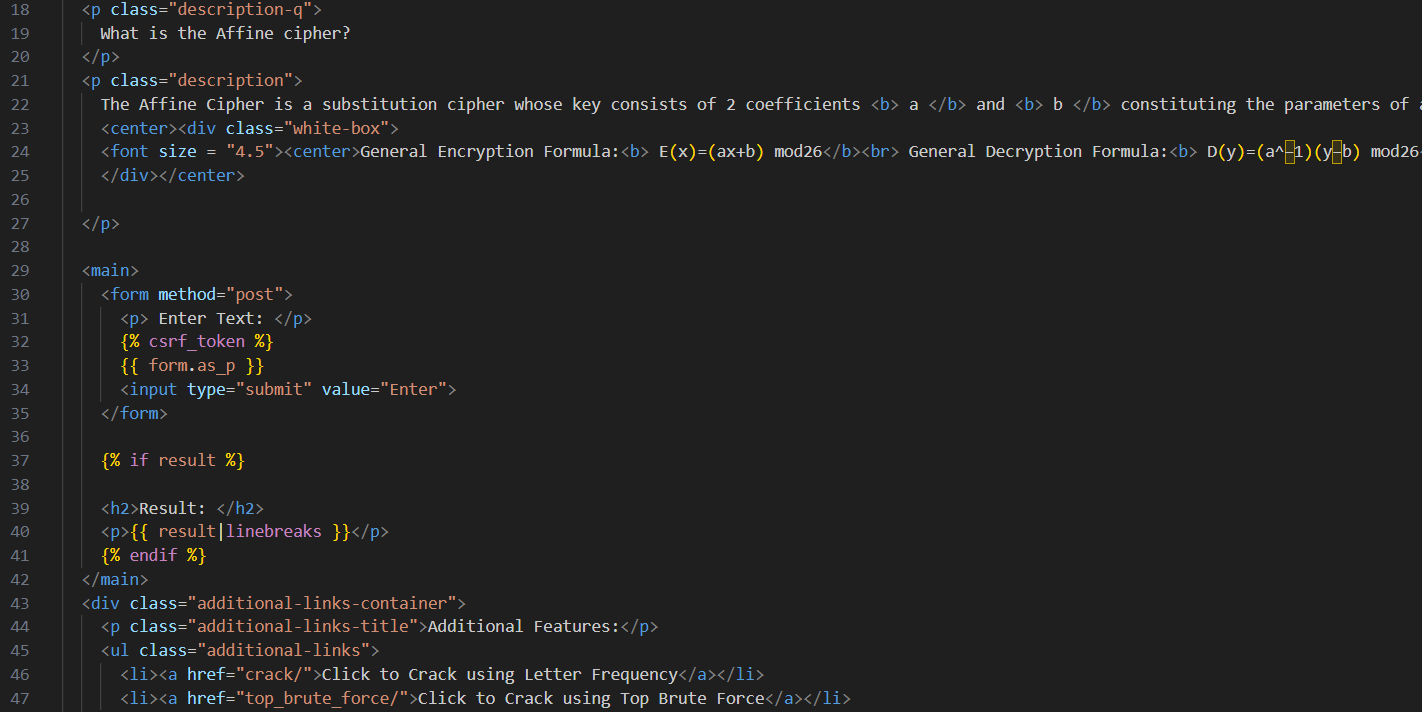
****

Figure 2: affine.html Snippet for General Encryption/Decryption for Affine

For the frontend part of the **General Encryption and Decryption** section, the page we designed provides a comprehensive user interface for both encrypting and decrypting messages using the Affine Cipher. The **HTML** code begins with the necessary **static file loading** and a well-structured navigation bar, maintaining consistency with the overall design of the application. The main content gives a short explanation of the Affine Cipher, detailing the mathematical formulas used for encryption and decryption.

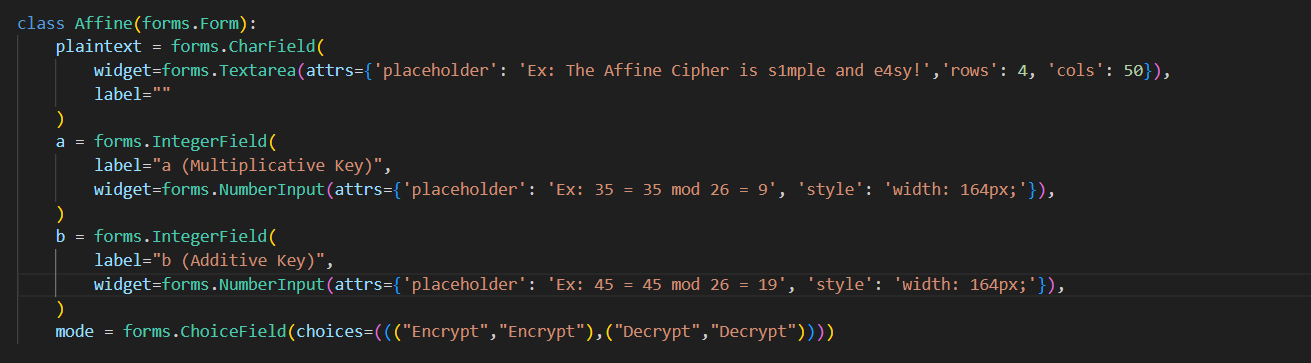


Figure 3: forms.py Section for General Encryption/Decryption for Affine

The user-friendly form captures **plaintext** input for encryption and **ciphertext** input for decryption, along with **a**, **b** and a **mode**, ensuring a straightforward interaction. Upon submission, the page dynamically displays the result, whether it be the encrypted or decrypted message. The inclusion of additional links directs users to cracking methods, enhancing the page's utility.

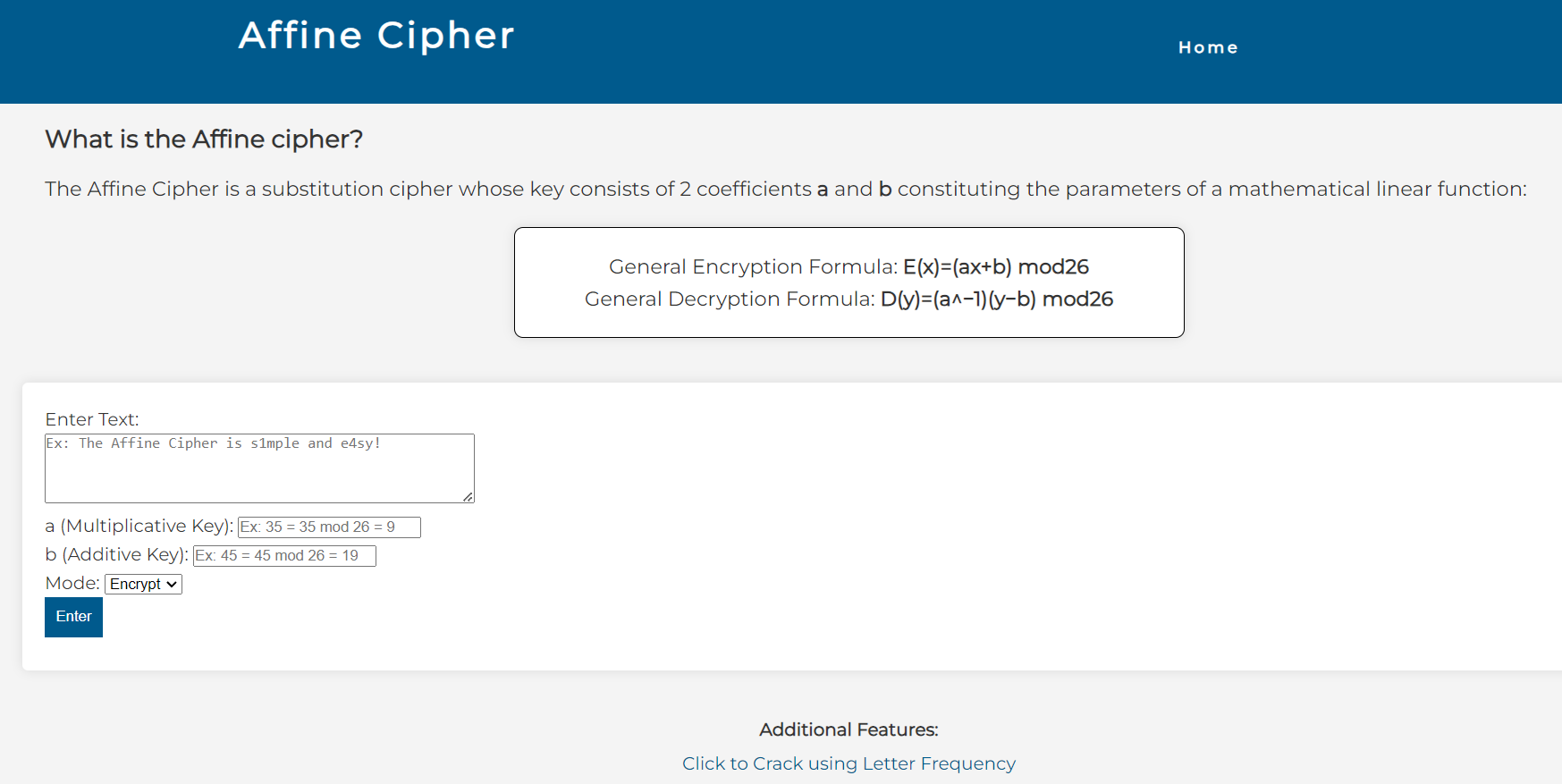


Figure 4: Webpage for General Encryption/Decryption for Affine

Notably, the page employs a clean and intuitive design, utilizing the Django framework's form handling to simplify user inputs, a grey text can be seen guiding the user through an example. The integration of external stylesheets and fonts contributes to a visually appealing presentation. The page's elegance lies in its simplicity, providing users with a seamless experience in both understanding the Affine Cipher principles and actively engaging with its encryption and decryption functionalities.



Figure 5: views.py Section for General Encryption for Affine

For the backend part, The **encrypt** function efficiently implements Affine Cipher encryption. It iterates through each character in the plaintext, handling uppercase letters while maintaining proper casing. Using the Affine Cipher formula with coefficients **a** and **b**, the function computes the encrypted index, ensuring modularity for the alphabet's bounds. The result is appended to the ciphertext, and non-alphabetic characters remain unchanged.



Figure 6: views.py Section for General Decryption for Affine

The **decrypt** function presents an effective implementation of Affine Cipher decryption. Utilizing a modular inverse (**a\_inv**) calculated through an auxiliary function, the method checks for its existence to avoid decryption issues. The subsequent loop processes each character in the ciphertext, handling uppercase letters while preserving their case. The function applies the inverse Affine Cipher formula to compute the decrypted index, ensuring modularity for the alphabet's bounds. Non-alphabetic characters remain unchanged in the final plaintext.

The integration of the encrypt and decrypt functions within the broader context of a conditional statement showcases an elegant and compact design, facilitating both encryption and decryption with minimal code duplication.

### Crack Using Letter Frequency:

Figure 7: crackaffine.html Snippet for Cracking using Letter Frequency Affine Cipher

For the frontend section dedicated to **cracking the Affine Cipher using Letter Frequency**. The concise design features static file loading, a clear navigation bar, and a brief explanation of the cracking method. Users can input ciphertext via a user-friendly form, and upon submission, the guessed plaintext is dynamically displayed.

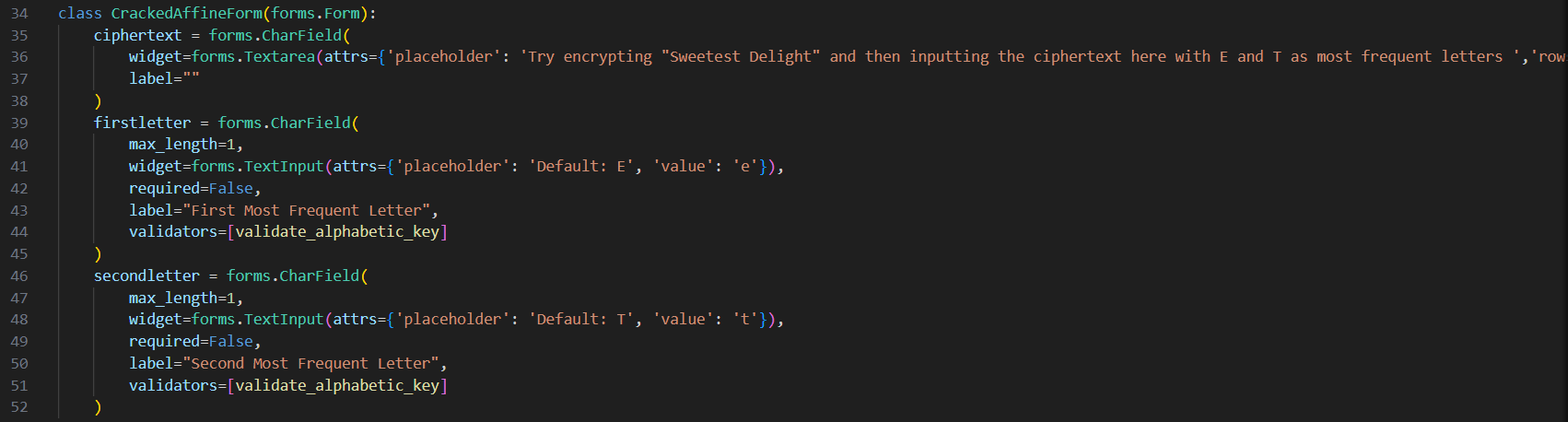


Figure 8: forms.py Section for Cracking using Letter Frequency Affine Cipher

The form used includes a **ciphertext** field, allowing users to input the encrypted message. Additionally, there are **optional** fields for users to provide **initial guesses** for the two most frequent letters, labeled **firstletter** and **secondletter**. The form employs placeholders and default values, offering guidance to users. Notably, the **firstletter** and **secondletter** fields are restricted by the **validate\_alphabetic\_key validator**, ensuring that users only input valid alphabetic characters. This meticulous validation enhances the form's reliability. The overall design of the form prioritizes user-friendliness.

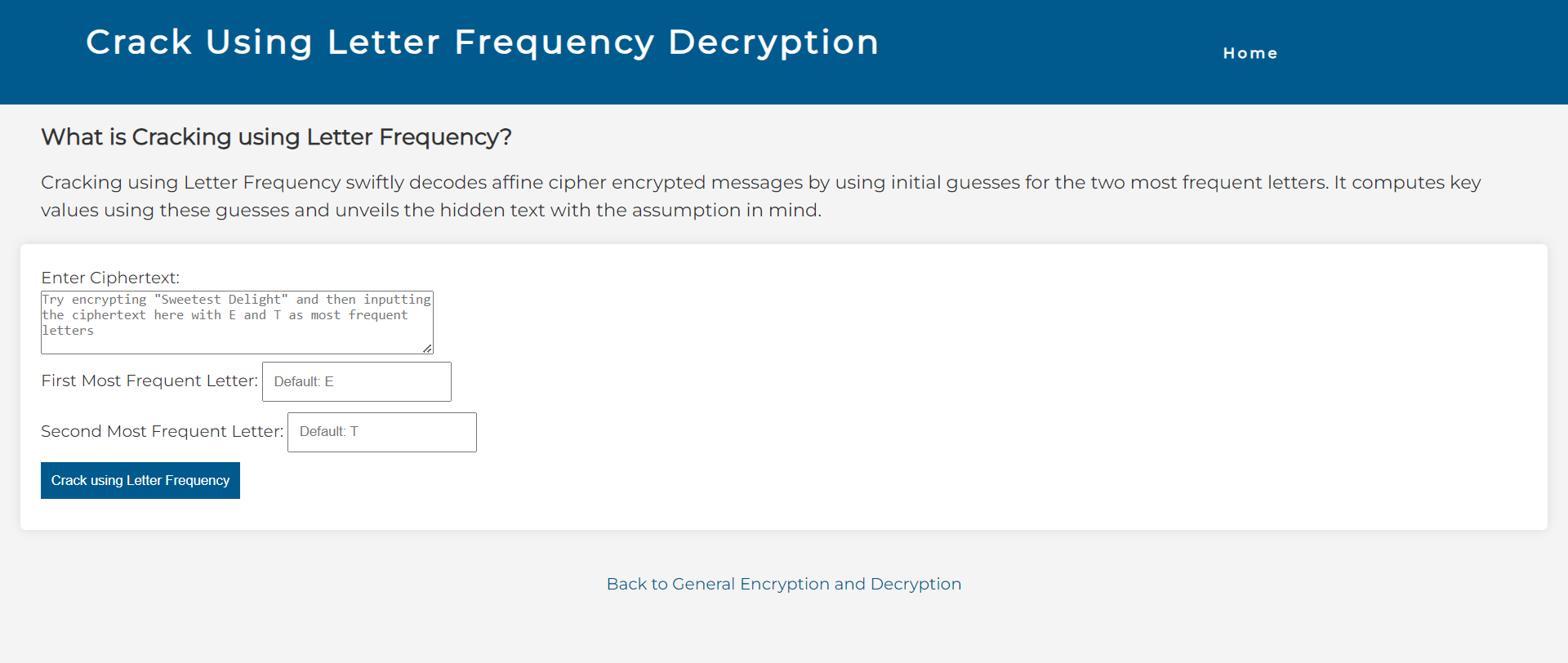


Figure 9: Webpage for Cracking using Letter Frequency Affine Cipher

The page employs a clean and intuitive design like the rest, utilizing the Django framework's form handling to simplify user inputs, a grey text can be seen guiding the user through an example. The integration of external stylesheets and fonts contributes to a visually appealing presentation.

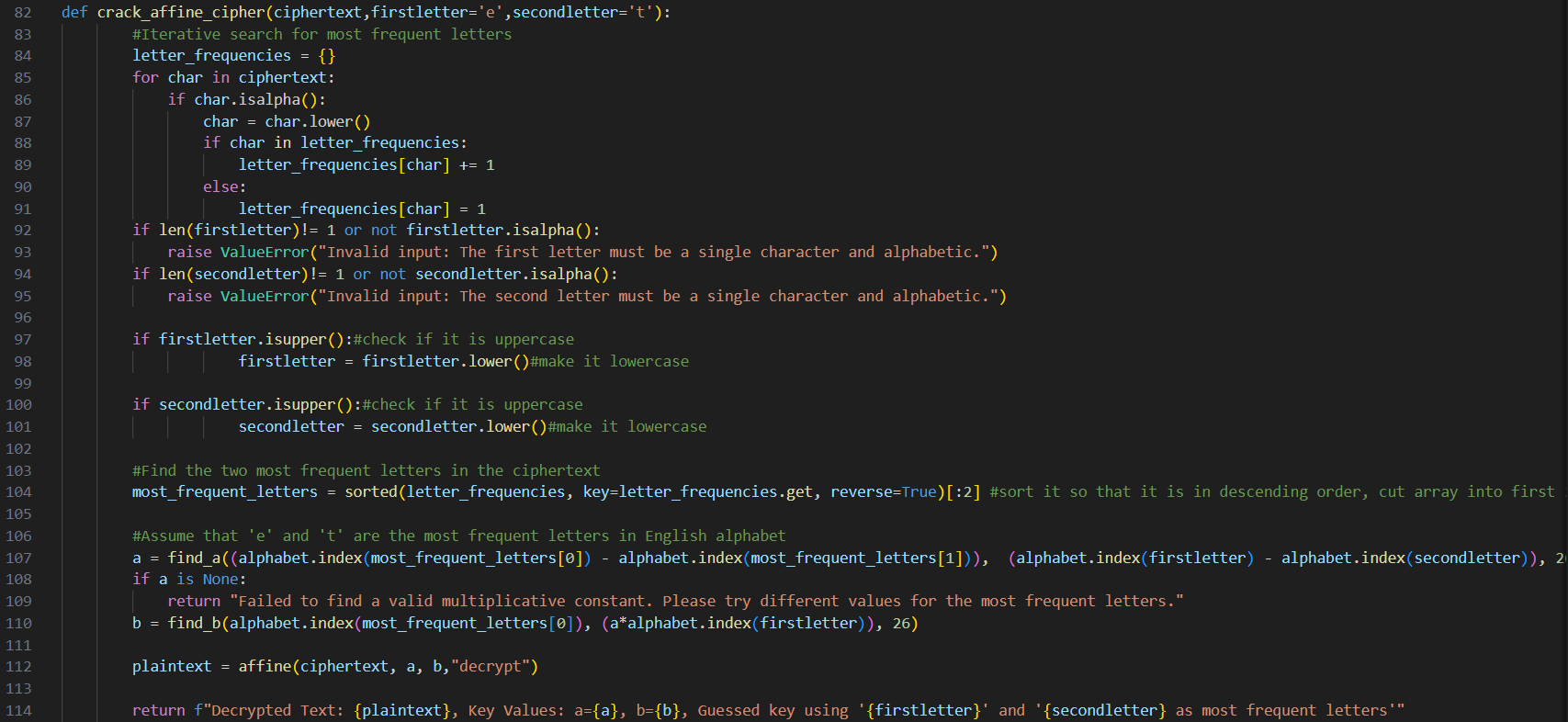


Figure 10: views.py Section for Cracking using Letter Frequency Affine Cipher

The **crack\_affine\_cipher** function employs Letter Frequency Decryption to crack an Affine Cipher using the given **ciphertext** and **optional** initial **guesses** for the two **most frequent letters**. It calculates letter frequencies, validates input for **firstletter** and **secondletter**, adjusts case if needed, and identifies the two most frequent letters. Assuming **firstletter** and **secondletter** as the **most frequent letters** in English, it calculates Affine Cipher key values (**a** and **b**). The function decrypts the ciphertext and returns the decrypted text along with the calculated key values. The result includes information about the guessed key using the provided most frequent letters, demonstrating a concise and effective approach to Affine Cipher cracking.

### Crack Using Top Brute Force:

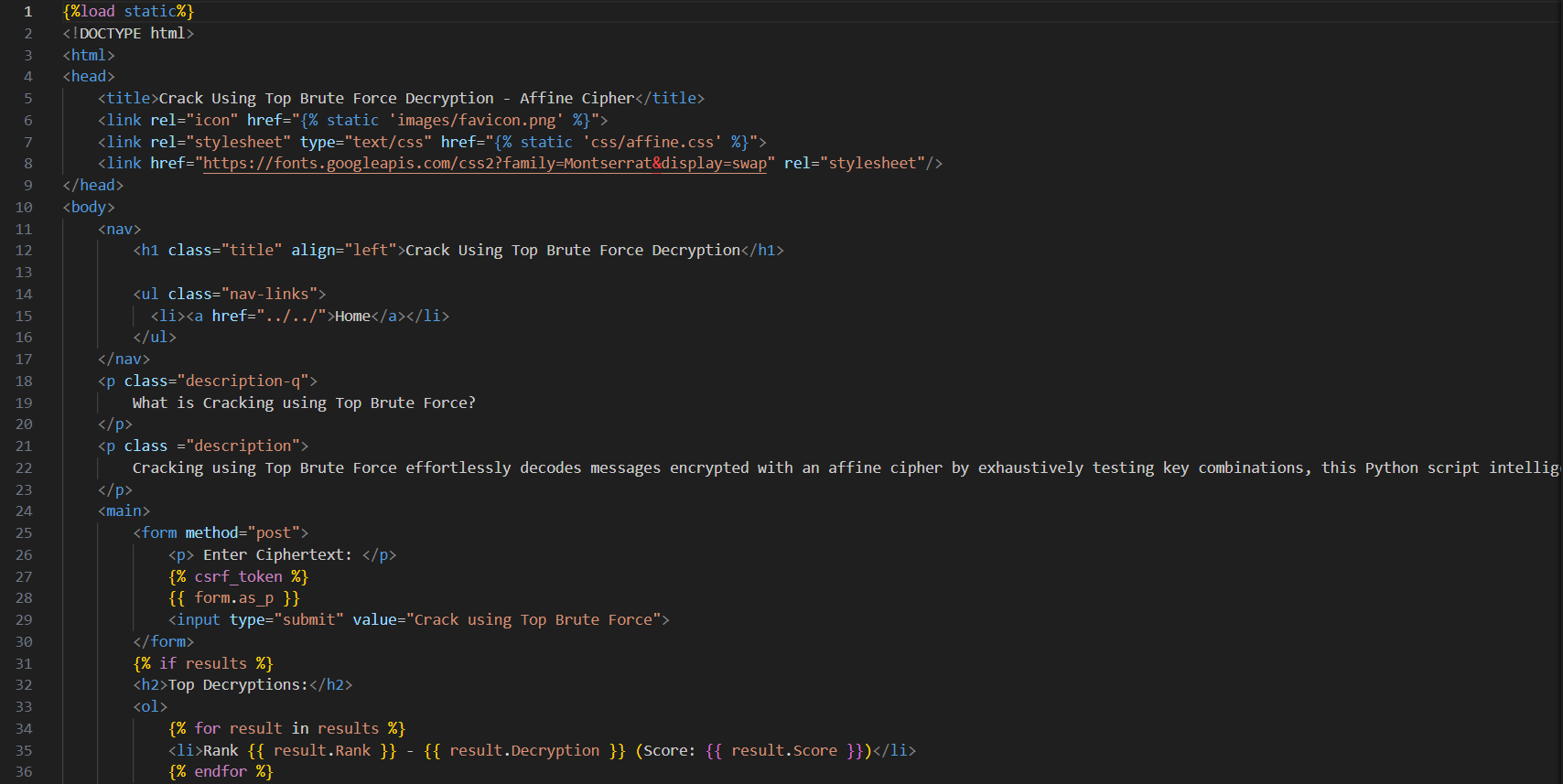
****

Figure 11: topbruteforceaffine.html Snippet for Top Brute Force Affine Cipher

For the frontend of **Top Brute Force Decryption**, our page offers an efficient and user-friendly interface for decoding Affine Cipher-encrypted messages. The HTML code, similar to the previous sections, featuring static file loading and a streamlined navigation bar, ensures visual consistency. The main content concisely introduces the Top Brute Force method's effectiveness in deciphering messages systematically. This explanation prepares users to interact with the form properly and understand the top decryption results.

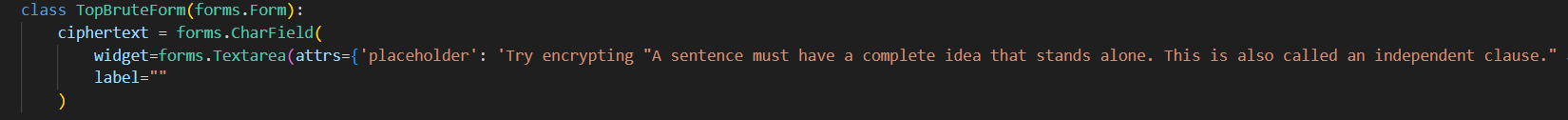


Figure 12: forms.py Section for Cracking using Top Brute Force Affine Cipher

**TopBruteForm** in **forms.py** defines a Django form for the Top Brute Force Decryption. It features a single field, **ciphertext**, with a **placeholder** in a text area widget to help the user try it. The **forms.py** section for this technique is more straightforward than the previous sections.



Figure 13: Webpage for Cracking using Top Brute Force Affine Cipher

The page employs a clean and intuitive design like the rest, a grey text can be seen guiding the user to try an example. The integration of external stylesheets and fonts contributes to a visually appealing presentation.

A black screen with text on it

Description automatically generated

Figure 14: views.py Section for Cracking using Top Brute Force Affine Cipher

The **top\_brute\_force** function systematically decrypts an Affine Cipher-encrypted message by exhaustively testing key combinations. It employs a predefined set of **common** English words to evaluate the **quality** of each decryption. The function **iterates** through possible key pairs, calculating a decryption **score** based on the **frequency** of common words in the resulting text. The **top three** scoring decryptions are then sorted and returned as a list of dictionaries, each containing the rank, decrypted text, and score. This method intelligently ranks potential decryptions, prioritizing those with a higher frequency of common English words.

## Vigenère Cipher:

The Vigenere Cipher section of our code required two distinct functionalities: encryption, and decryption. Each of these functionalities demanded a separate backend and frontend implementation.

### General Encryption and Decryption:



Figure 15: vigenere.html Snippet for General Encryption/Decryption for Vigenère

For the frontend part of the **General Encryption and Decryption** section, the page we designed provides a comprehensive user interface for both encrypting and decrypting messages using the Vigenère **Cipher**. The **HTML** code begins with the necessary **static file loading** and a well-structured navigation bar, maintaining consistency with the overall design of the application. The main content gives a short explanation of the Vigenère Cipher, detailing the mathematical formulas used for encryption and decryption.

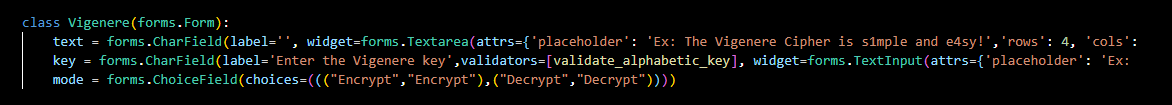


Figure 16: forms.py Section for General Encryption/Decryption for Vigenère.

This form is designed to capture user inputs for encryption and decryption using the Vigenère cipher. Once the user submits the form, the input data is processed by the web application, applying the Vigenère cipher algorithm based on the specified key and mode. The result can then be dynamically displayed on the web page.

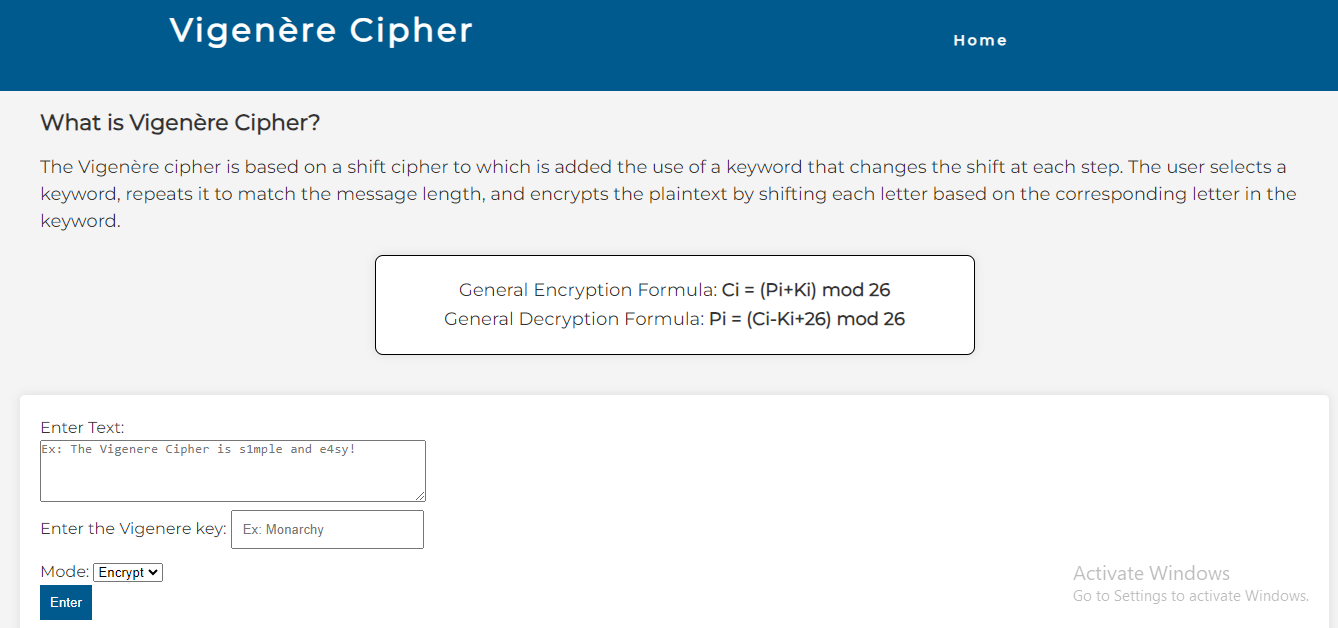


Figure 17: Webpage for General Encryption/Decryption for Vigenère

Notably, the page employs a clean and intuitive design, utilizing the Django framework's form handling to simplify user inputs, a grey text can be seen guiding the user through an example. The integration of external stylesheets and fonts contributes to a visually appealing presentation. The page's elegance lies in its simplicity, providing users with a seamless experience in both understanding the Vigenère **Cipher** principles and actively engaging with its encryption and decryption functionalities.

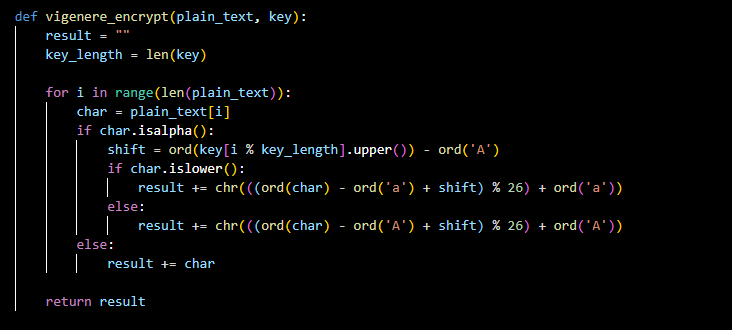


Figure 18: views.py section for General Encryption for Vigenère

For the backend part, the Vigenère encryption function efficiently applies the principles of the Vigenère cipher, where each character is shifted based on the corresponding character in the key. The function ensures proper casing for letters and handles non-alphabetic characters, making it suitable for encryption purposes.

Note: the **ord** function provides a unicode of a character.

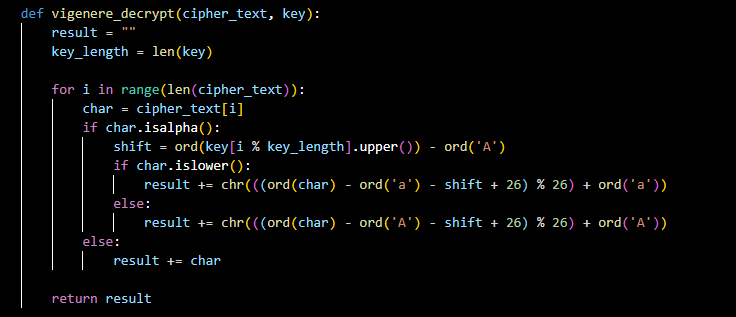


Figure 19: views.py Section for General Decryption for Vigenère

The Vigenèredecryption function efficiently applies the principles of the Vigenèrecipher in reverse, effectively shifting characters back based on the corresponding character in the key. The function ensures proper casing for letters and handles non-alphabetic characters during decryption.

## Monoalphabetic Cipher:

The Monoalphabetic Cipher section of our code required two distinct functionalities: encryption, and decryption. Each of these functionalities demanded a separate backend and frontend implementation.

### General Encryption and Decryption:

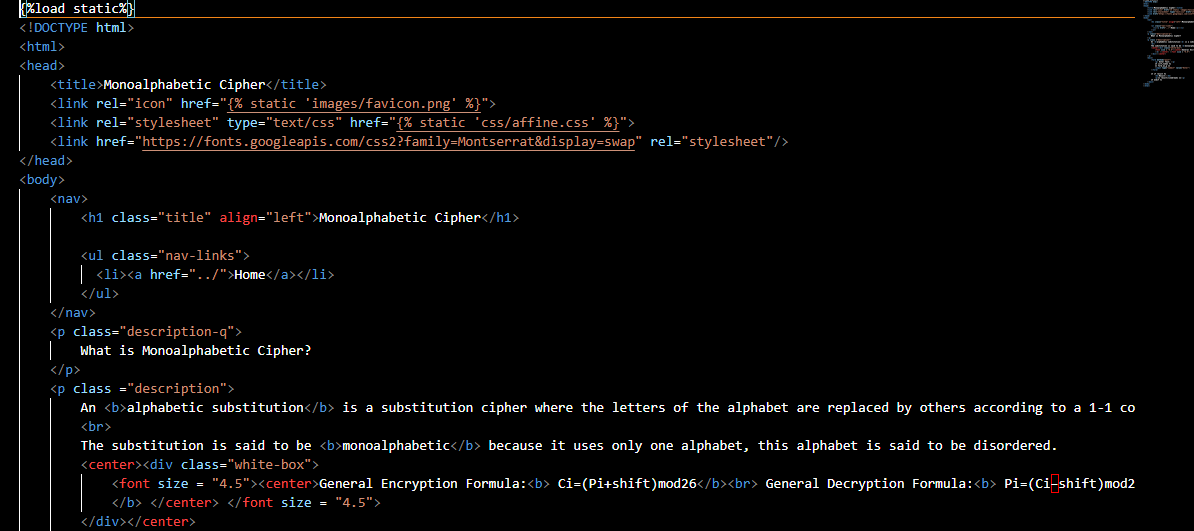


Figure 20: monoalphabetic.html Snippet for General Encryption/Decryption for Monoalphabetic

For the frontend part of the **General Encryption and Decryption** section, the page we designed provides a comprehensive user interface for both encrypting and decrypting messages using the **Monoalphabetic Cipher**. The **HTML** code begins with the necessary **static file loading** and a well-structured navigation bar, maintaining consistency with the overall design of the application. The main content gives a short explanation of the Monoalphabetic Cipher, detailing the mathematical formulas used for encryption and decryption.

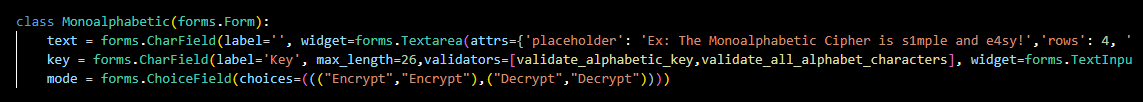


Figure 21: forms.py Section for General Encryption/Decryption for Monoalphabetic

This form is designed to capture user inputs for encryption and decryption using the Monoalphabetic cipher. Once the user submits the form, the input data is processed by the web application, applying the Monoalphabetic cipher algorithm based on the specified key and mode. The result can then be dynamically displayed on the web page.

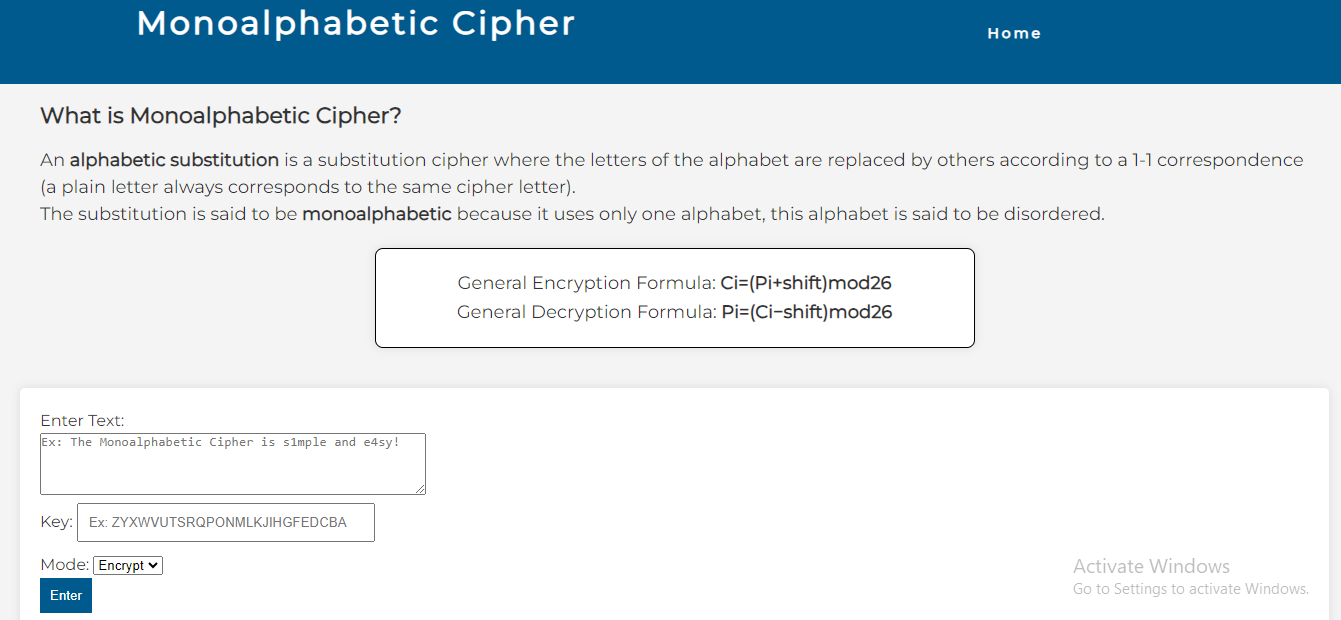


Figure 22: Webpage for General Encryption/Decryption for Monoalphabetic

Notably, the page employs a clean and intuitive design, utilizing the Django framework's form handling to simplify user inputs, a grey text can be seen guiding the user through an example. The integration of external stylesheets and fonts contributes to a visually appealing presentation. The page's elegance lies in its simplicity, providing users with a seamless experience in both understanding the **Monoalphabetic Cipher** principles and actively engaging with its encryption and decryption functionalities.

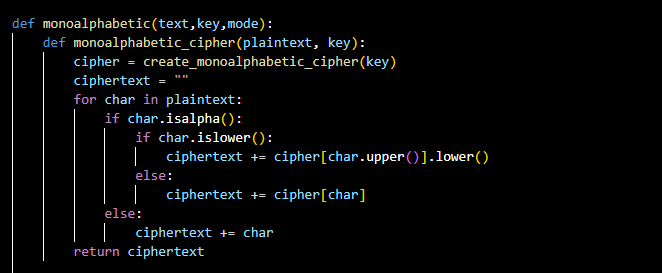


Figure 23: views.py Section for General Encryption for

The main **monoalphabetic** function acts as a wrapper, taking the input text (text), key, and mode as parameters. It delegates the actual encryption or decryption process to the nested **monoalphabetic\_cipher** function.

The nested function, **monoalphabetic\_cipher**, encapsulates the core logic for processing individual characters and applying the monoalphabetic substitution.



Figure 24: views.py Section for General Decryption for Monoalphabetic

The decryption function reverses the encryption process, utilizing an inverse cipher to map ciphertext characters back to their original plaintext counterparts. Both functions handle uppercase and lowercase letters, maintaining proper casing, and non-alphabetic characters remain unchanged.

## Playfair Cipher:

### Backend:

#### Playfair Code

For the backend aspect of the code, the Playfair cipher was implemented in a way such that it accepts non-letter characters, however, doesn’t encrypt nor decrypt them.

The code first generates the key matrix as a string. When the key is given, all whitespaces are removed and all J’s are replaced with I’s. Then the program iterates over the key and then the alphabet list. If the current letter had not been encountered previously and is found in the predefined alphabet list (excluding J) , it will be added to the string.

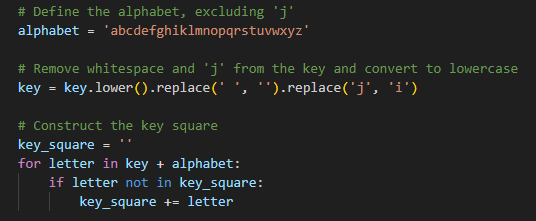


Figure : Key Generation Code for Playfair Cipher

## 

## 

## 

## 

The given text is then handled. To prevent future errors, the code iterates over the text and checks whether there are any non-letter characters. It only stores the letters of the text in a new variable maintaining the capitalized and small letters. After that the new string is divided into a set of digraphs, as the cipher requires. While dividing the string, the code checks for special cases. For example, if both letters of one pair are the same, padding is needed. We chose X to be the primary padding letter as its occurrence (having two X’s together) is not frequent. For added measure, we used Z to be our secondary padding letter, in case we had a digraph that has 2 X’s although it’s very rare to happen. These paddings are applied to the string that was formed in the previous step, say K. After iterating K, we check its length. If it is odd, padding is needed, so x is added to the end of the string. Finally, the digraphs will be stored in a list.

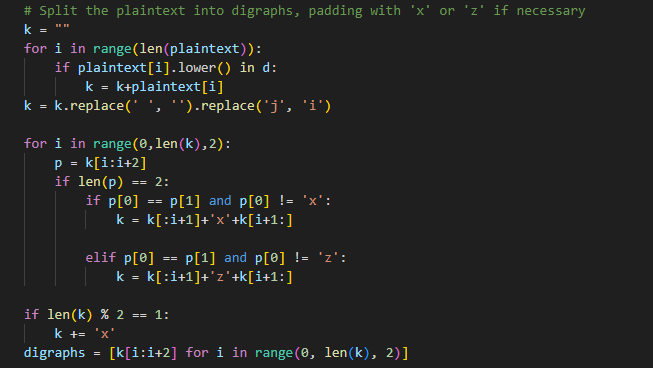
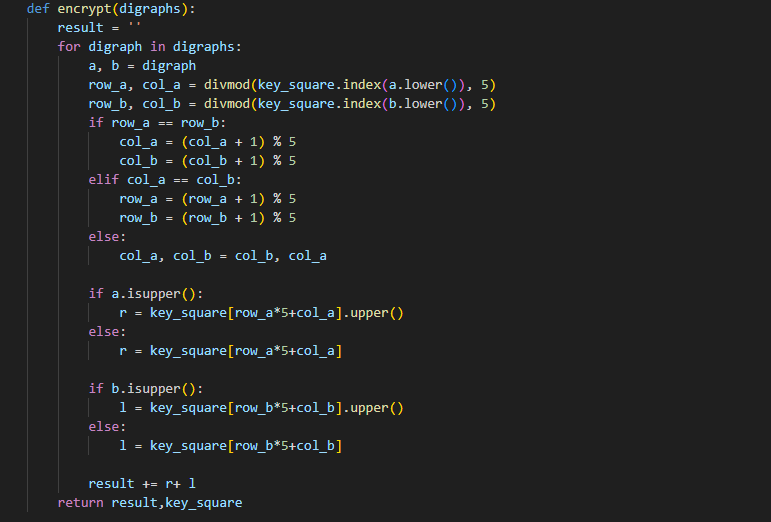
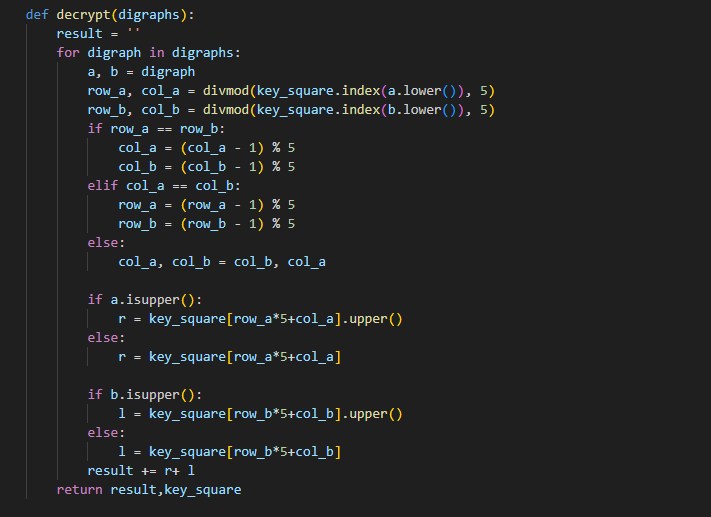
The next step is encryption and decryption. The process is as follows. For each digraph, we get the row and column position of the letter in the key matrix. This is done by using divmod(index\_of\_letter,5). Divmod returns the quotient and the remainder of the division of index\_of\_letter and 5. The row is the quotient, and the column is the remainder. This works since we’re dealing with a 5x5 matrix. For example, if the index of the letter is 16, then it is in the 3rd row and 1st column of the matrix (Note that the rows are numbered from 0 till 4 and not starting 1). If both letters are on the same row, then the column of each letter is shifted to the right or left (increment or decrement column by one) when encrypting and decrypting respectively. The same applies for letters of the same column as the rows are either shifted upward or downward. To make sure we don’t get out of index, we use the mod operator (%) to keep the value of the row or the column less than 5. If neither condition applies, the column will be swapped. This process maintains the capital and small letters by checking their original state and appending to the resultant string accordingly.

Figure : Forming Digraphs in Playfair

Figure : Decryption function of Playfair

Figure : Encryption Function of Playfair

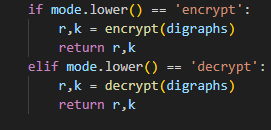
Finally, the code checks the mode to call the appropriate function. The resultant text and key matrix will be returned.

Figure : Calling the Function According to the Node

Note: The code doesn’t preserve the locations of non-letter characters since it is hard to track their new locations due to the possibility of padding.

Some parts of the code were taken from [6].

#### Forms.py

Two character fields and one choice field were used to get the necessary input from the user. The character fields are used to get the plain/cipher text and to get the key respectively. To make sure no error occurs, a validator was specified, all characters should be ASCII characters. Else, the program won’t accept the input as it would be considered invalid. As for the choice field, it allows the user to choose between encryption or decryption.

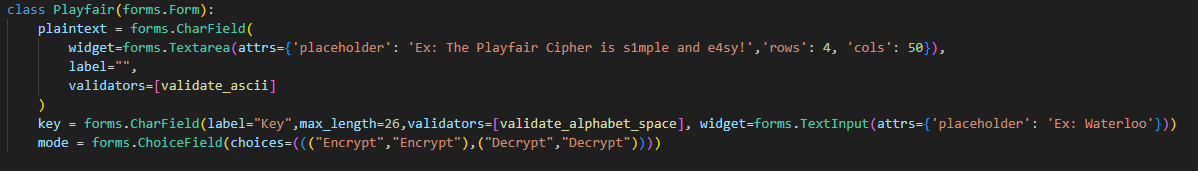


Figure : forms.py Section of the Playfair Cipher

#### Views.py

The views file connects the forms, python functions, and HTML page all together. The form will be displayed on the HTML page. If there is a request, all necessary inputs will be stored in variables and then passed to the python function of Playfair as parameters. Then the returned outputs are also stored in variables. To be able to display the key matrix, as a table on the HTML page, the key outputted as a string is transformed into a list. Then the request and the necessary arguments are rendered to the html page of the Playfair cipher. 

Figure : views.py Section of the Playfair Cipher

## Frontend

Moving onto the frontend side, we will discuss the components of the HTML page.

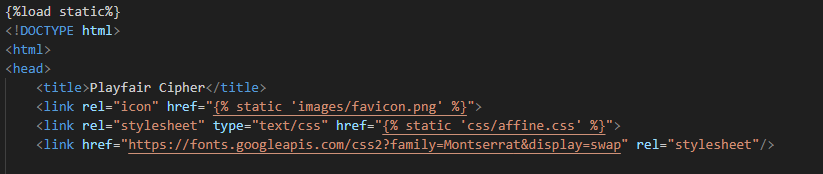
First the style of the page (background, Home link) is inherited from the styles.css file found in the static folder.

Figure : Inheriting the Base Style

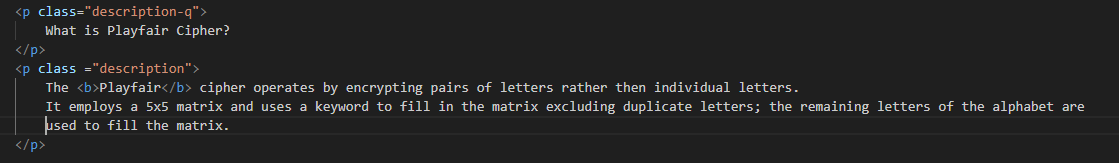
We then added a basic description of the Playfair cipher.

Figure : Code for Description

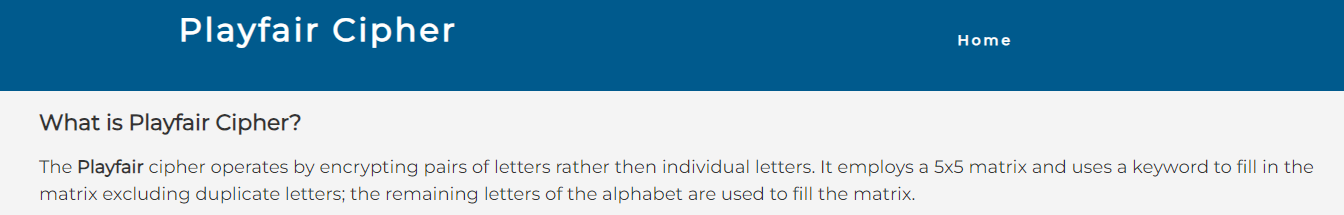
****

Figure 34: Description and Title on the Webpage

The forms are displayed by using the “form” that is requested from the view.py file. The csrf\_token ensures that no malicious attack happens when rendering the page.

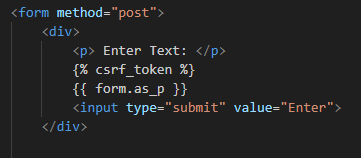


Figure : Code to Display the Forms

Figure : Forms on the Webpage

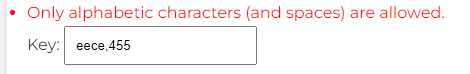
The interface is very basic. The text area could include anything that is considered an ASCII character. As for the key, it shouldn’t contain a non-letter character (a warning message will appear). The dropdown menu allows the users to choose the mode they want. And when the inputs are ready and upon clicking the Enter button, the inputs will be sent to be processed.

Figure : Error when Key Contains Non-letters

After processing the input, the result will be shown. In addition to that, the key matrix will be displayed as a 5x5 table.

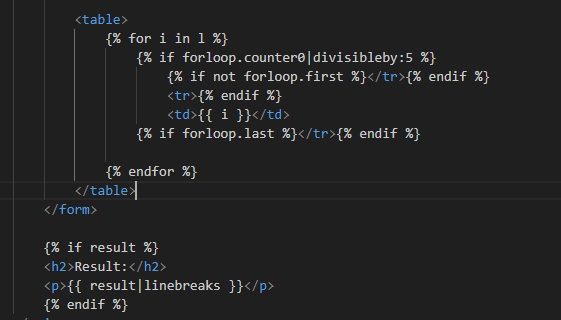


Figure : Results and Table Code

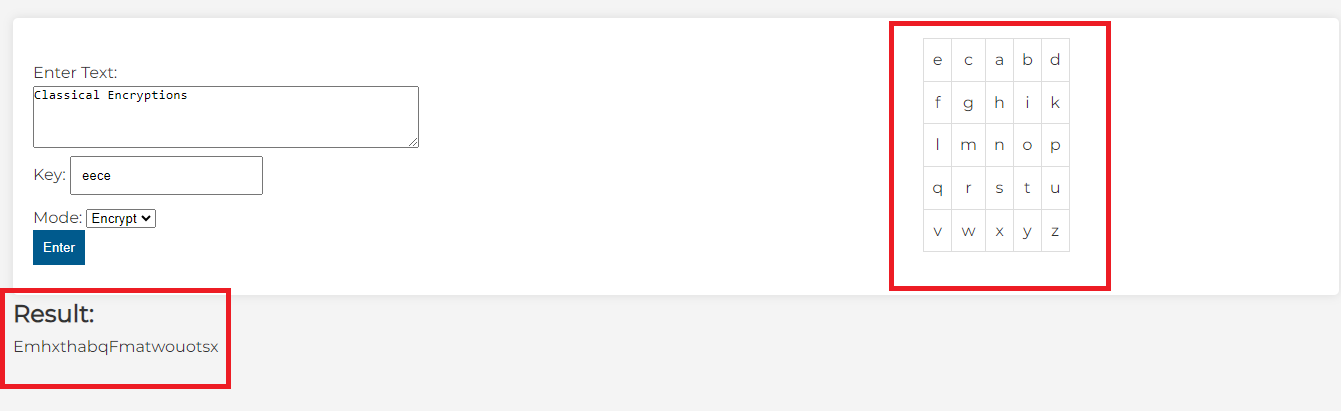
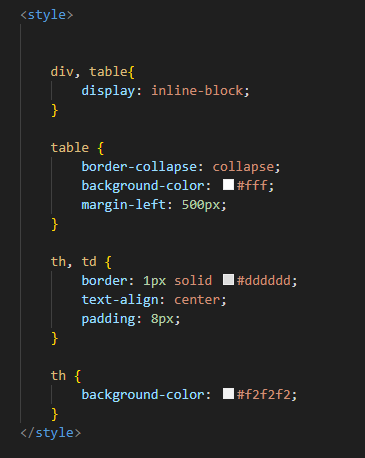


Figure : Styles for the table

Figure : Display of Result and Key Matrix on Webpage

## Hill Cipher:

The Hill Cipher implementation within the project encompasses various components, each contributing distinct functionalities to facilitate encryption and decryption processes.

### HTML/CSS Interface:

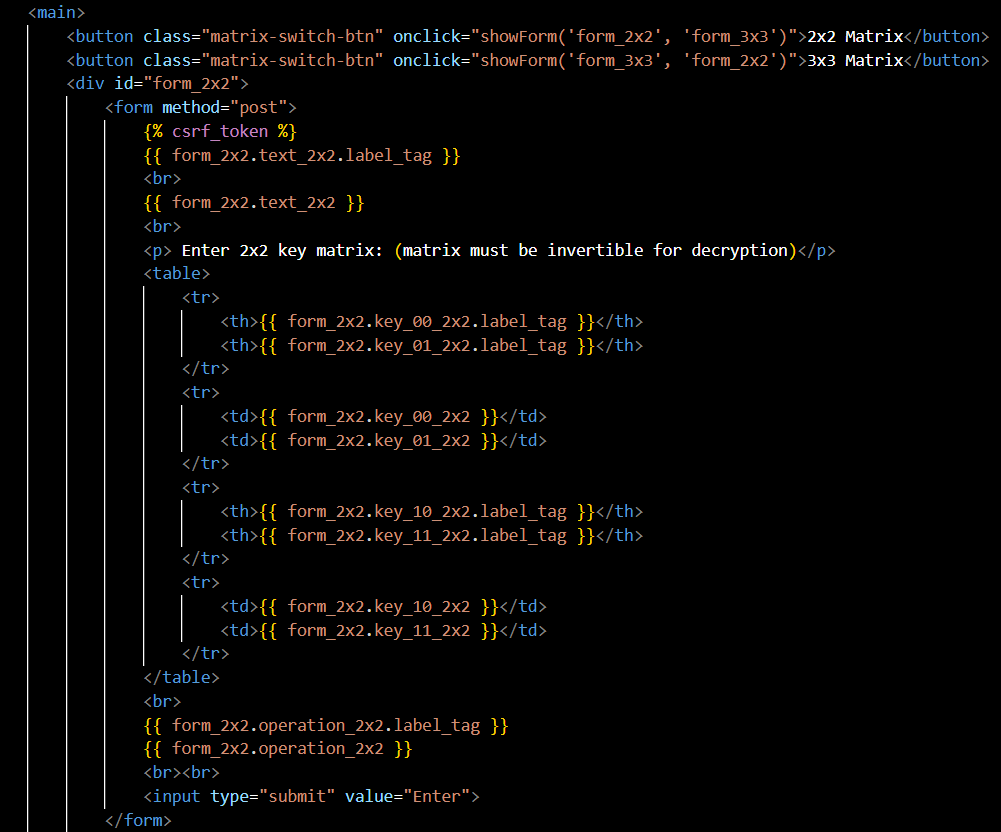


Figure 41: hillcipher.html Snippet for General Encryption/Decryption for Hill Cipher

The HTML file serves as the user interface, providing a platform for user interaction. It incorporates elements like matrix selection buttons, input forms for text and key matrices, and a mechanism to display results dynamically. These elements collectively enable users to input text, select matrix sizes, and execute encryption or decryption operations seamlessly.

### Django Forms:

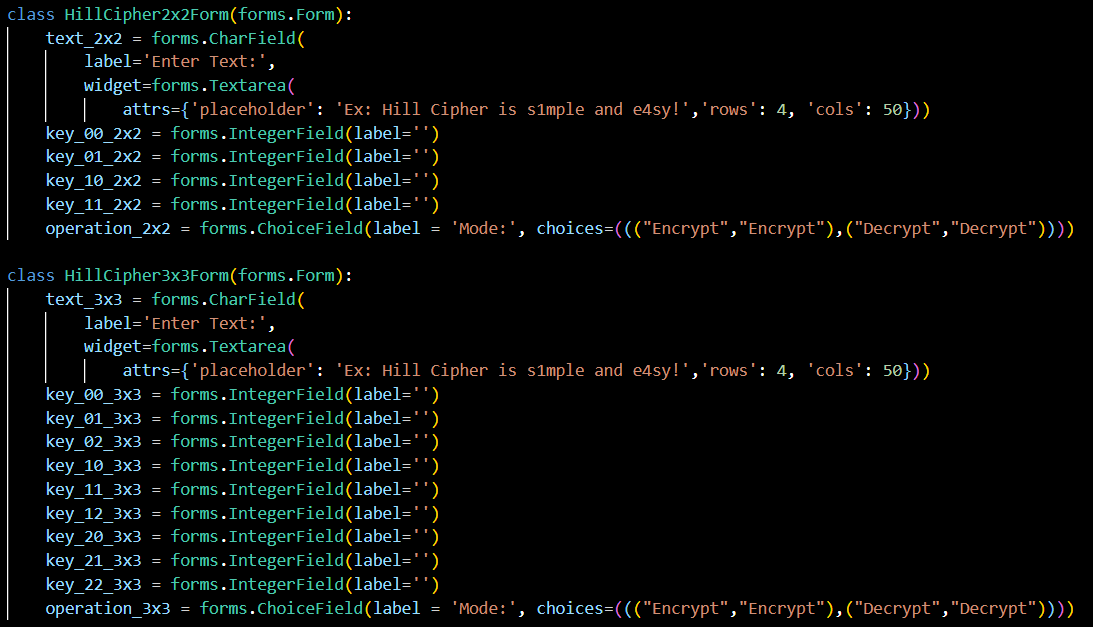


Figure 42: forms.py section for General Encryption/Decryption for Hill Cipher

The **‘HillCipher2x2Form’** and **‘HillCipher3x3Form’** are designed to capture user input for the Hill Cipher encryption and decryption process. They collect text input for encryption/decryption and numerical values to represent the key matrix. The forms also include a choice field to select between encryption and decryption modes. These forms ensure that users provide the necessary data required for the Hill Cipher algorithm to perform accurately.

Views:



Figure 43: views.py section for General Encryption/Decryption for Hill Cipher

Within the Django views, the **‘hillcipher’** function orchestrates the processing of user inputs and coordinates the encryption and decryption procedures. This function interacts with the forms, extracting validated data, including text, key matrices, and selected operations. Subsequently, it triggers the **‘hill’** function, passing the necessary parameters for encryption or decryption based on the user's choice. The function then handles the obtained results, preparing them for display back to the HTML template.

Python backend function:

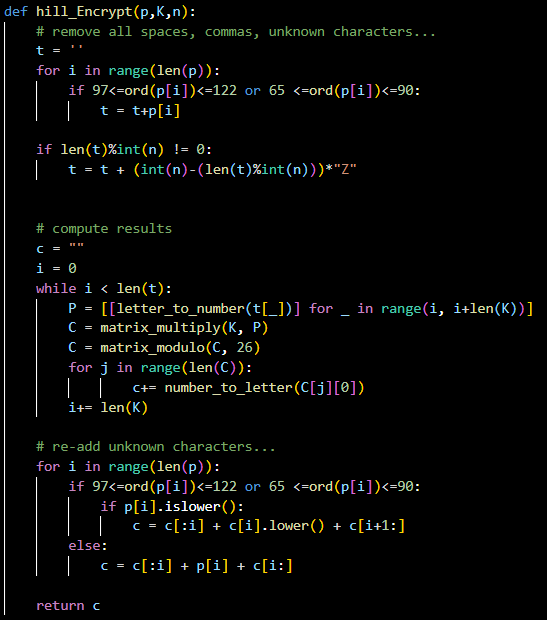
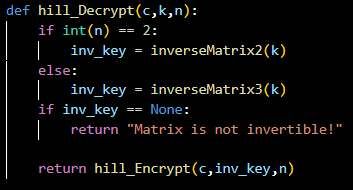


Figure : Decryption function of Hill

Figure 45: Encryption function of Hill

The core functionalities of the Hill cipher are encapsulated within the hillcipher.py file. This module includes several key functions, which collectively manage matrix operations, modular arithmetic, character conversions, and the actual encryption and decryption processes based on the Hill cipher algorithm. They enable the conversion of user inputs into encrypted or decrypted text, adhering to the principles of the Hill cipher.

### Main Page:

Figure : Webpage for General Encryption/Decryption for Hill

The main page offers users an initial glimpse into the Hill cipher and its application within classical encryption. Through a clean and intuitive interface, it presents key elements such as navigation links for seamless exploration, a descriptive section elucidating the essence of the Hill cipher, and interactive components like matrix selection buttons and input forms for text and key matrices. This amalgamation of educational content and user-friendly design aims to engage users while imparting fundamental knowledge about the Hill cipher's significance and functionality within the realm of encryption.

## Extended Euclidean Algorithm:

### HTML Interface (extendedeuclid.html):

A computer screen shot of a program

Description automatically generated

Figure 47: extendedeuclid.html Snippet for Extended Euclid

The HTML file for the Extended Euclidean Algorithm page furnishes a user-friendly interface tailored for computing modular inverses. Its components include:

The page's design encompasses a navigational section, mirroring the structure of other project pages, facilitating seamless navigation within the application. Additionally, it features a descriptive section outlining the purpose and functionality of the Extended Euclidean Algorithm. This section aims to elucidate how the algorithm computes the modular inverse of 'a' modulo 'm' when 'a' and 'm' are coprime.

The user interaction section is realized through a form soliciting 'a' and 'b' values. With placeholders and styling, the form enhances clarity and ease of input. Upon form submission, a POST request triggers backend processing.

### Backend Processing (views.py):

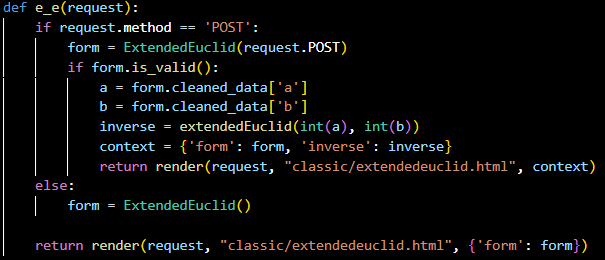


Figure 48: views.py section for Extended Euclid

The **‘e\_e’** view function within the backend manages the processing of user inputs. It functions to handle both GET and POST requests:

For POST requests, it validates the 'a' and 'b' values obtained from the user input form. Subsequently, it utilizes the Extended Euclidean Algorithm logic to compute the modular inverse. The result is prepared for display in the HTML template.

For GET requests, it renders the initial HTML template for the Extended Euclidean Algorithm page, presenting the user with the form to input 'a' and 'b' values.

### Form Design (forms.py):

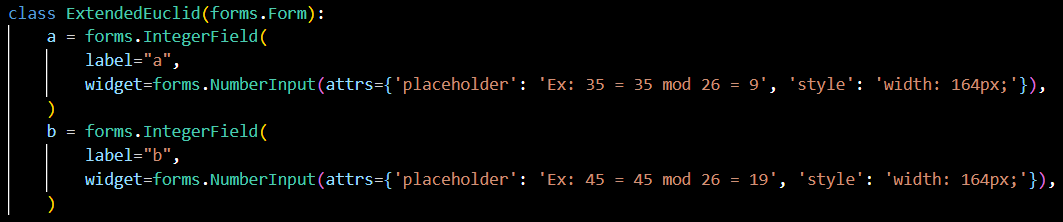


Figure 49: forms.py section for Extended Euclid

The **‘ExtendedEuclid’** form classifies and structures user input fields:

It includes fields for 'a' and 'b' inputs, restricting them to integer values. The form's design aligns with the HTML interface, offering placeholders and styling for clarity and user-friendliness.

# Testing & Results:

In this section dedicated to testing and results, we will rigorously assess the performance of our code by subjecting it to a diverse range of test cases. The subsequent presentation will showcase the corresponding outcomes, providing a comprehensive overview of the code's functionality and robustness across various scenarios.

## Affine Cipher:

### General Encryption and Decryption:

Figure 50: First Case using General Encryption and Decryption Affine

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, **a** and **b** input is modulo 26 (**a=9** and **b=19**) and finally any new lines are propagated into the result.

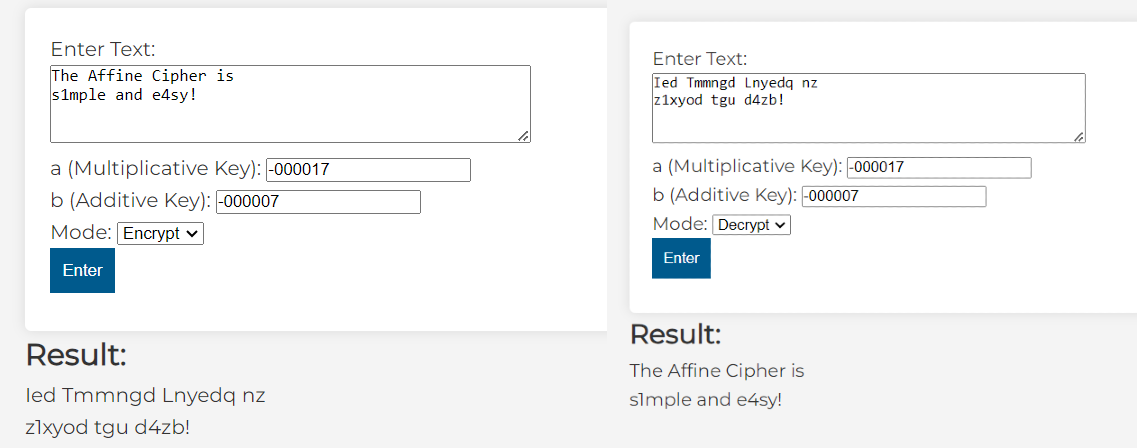


Figure 51: Second Case using General Encryption and Decryption Affine

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, **a** and **b** input is modulo 26, negative numbers work fine, inputting 000s in front don’t change anything (**a** = -000017 = **-17**mod26 = **9** and **b** = -000007=**-7**mod26 **= 19**), and finally any new lines are propagated into the result.

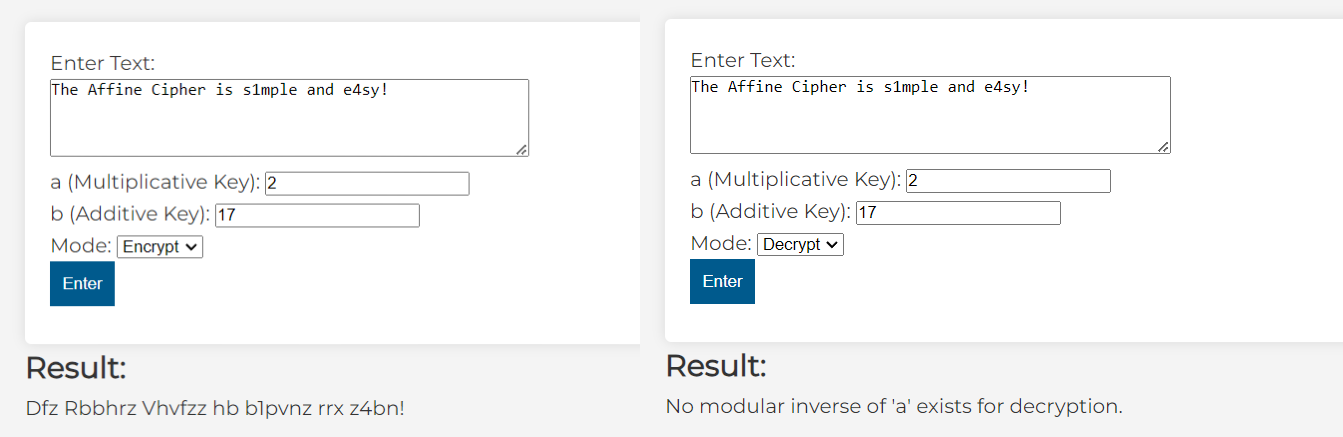


Figure 52: Third Case using General Encryption and Decryption Affine

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, **a** and **b** input is modulo 26 for the encryption part, however when attempting to decrypt the result mentions that no possible decryption is possible as **no modular inverse** of **a** exists, this is expected and stems from the attributes of the **Affine Cipher.**

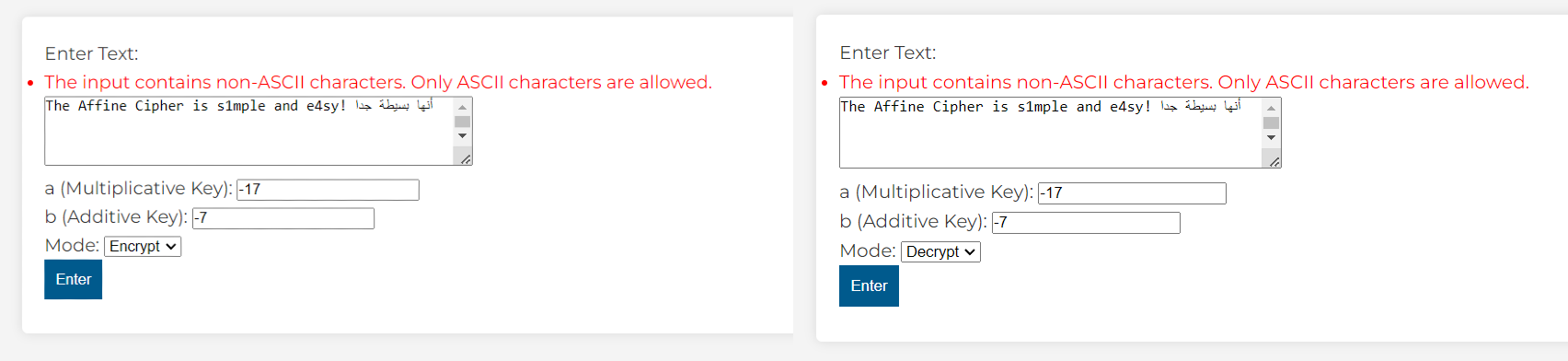


Figure 53: Fourth Case using General Encryption and Decryption Affine

The output is as we’d expect it, since the input contains **non-ASCII characters**, no result is outputted and a warning (generated by a **forms.py validator**) is given to use only **ASCII** **characters**.

#### Conclusion:

All possible cases are exhausted, the website can handle any integer value for **a** and **b** appropriately and any text input appropriately given that it is using only **ASCII** **characters** and that **a** has a modular inverse.

### Cracking using Letter Frequency:

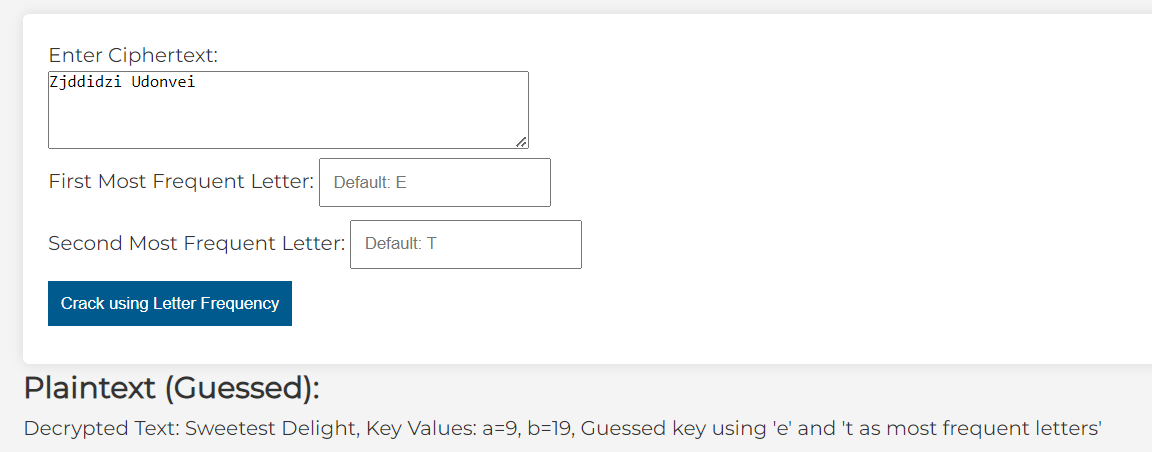
****

Figure 54: First Case using Crack using Letter Frequency

The output is as expected, though no values for the most frequent letters are given, the **forms.py** **value** allows the default inputs to be “e” and “t”.

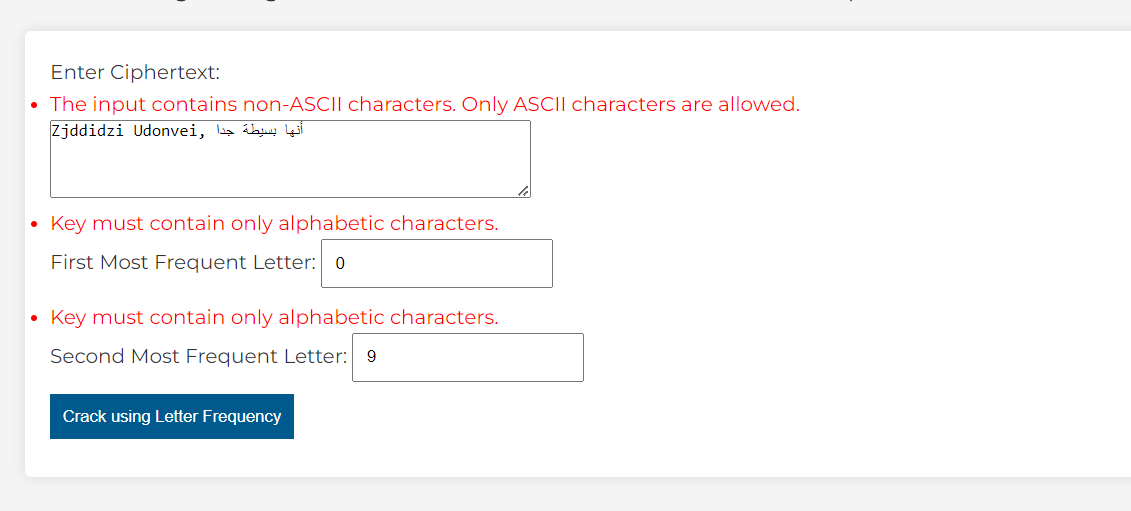
****

Figure 55: Second Case using Crack using Letter Frequency

The output is expected, no result is shown as the input for **firstletter** and **secondletter** must be a letter and the maximum length only **one** character (enforced by **form**), while for the **ciphertext** input the characters must be **ASCII.**

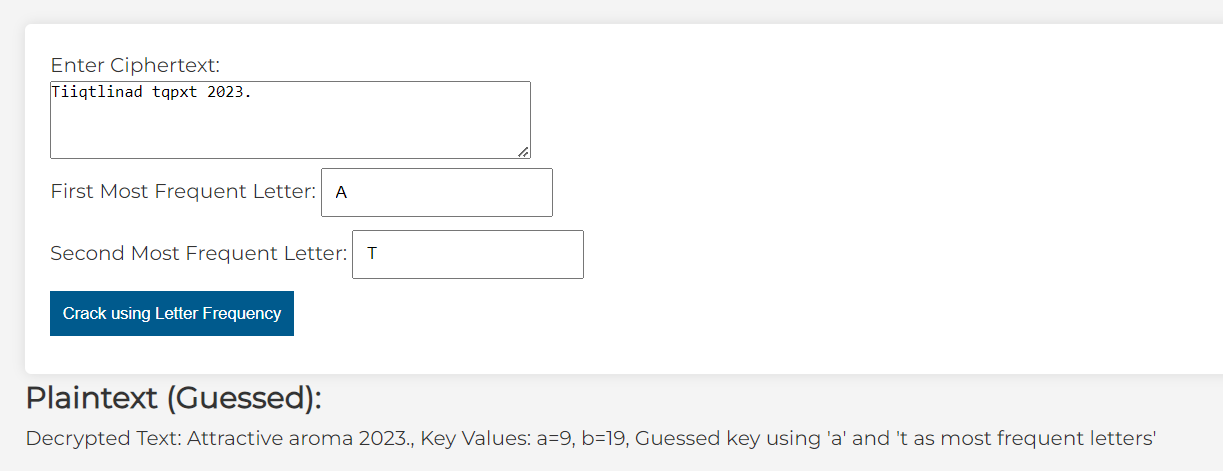
****

Figure 56: Third Case using Crack using Letter Frequency

The output is as expected, changing capitalization of **firstletter** and **secondletter** has no effect and a value other than “e” and “t” works correctly.

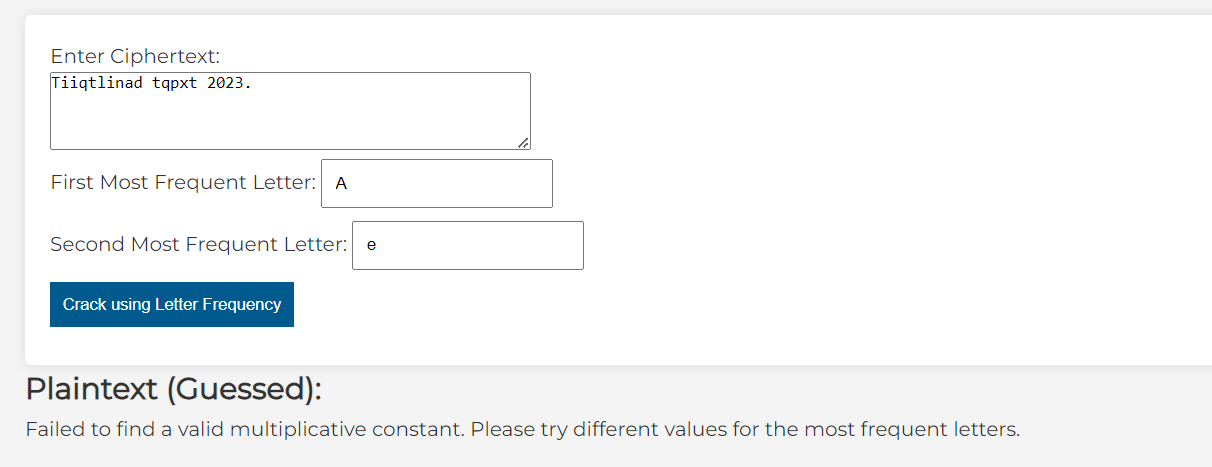


Figure 57: Fourth Case using Crack using Letter Frequency

The output is expected, the result mentions that no possible decryption is possible as **no modular inverse** of **a (multiplicative constant)** exists, this is expected and stems from the attributes of the **Affine Cipher.**

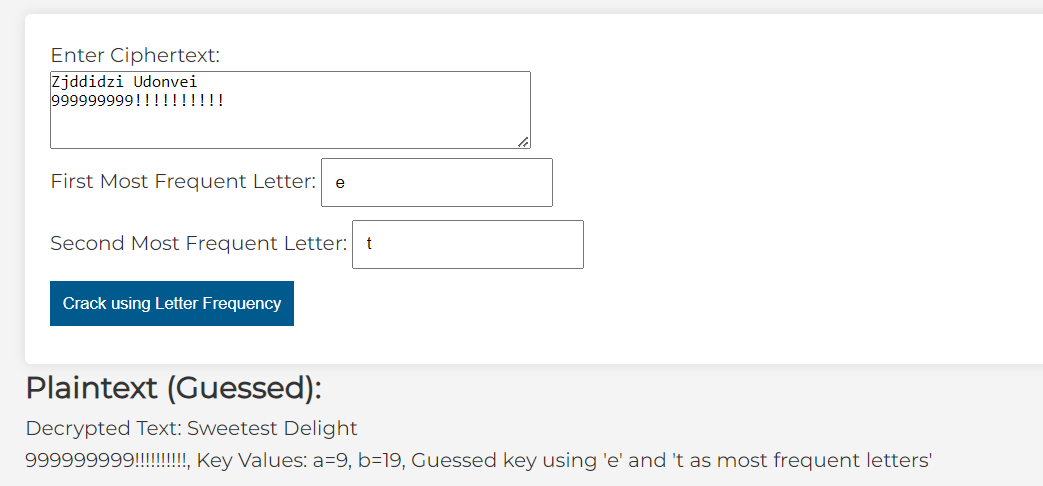
****

Figure 58: Fifth Case using Crack using Letter Frequency

The output is as expected, anything other than letters are **ignored** when **attempting to guess** and new lines are **considered** in the result.

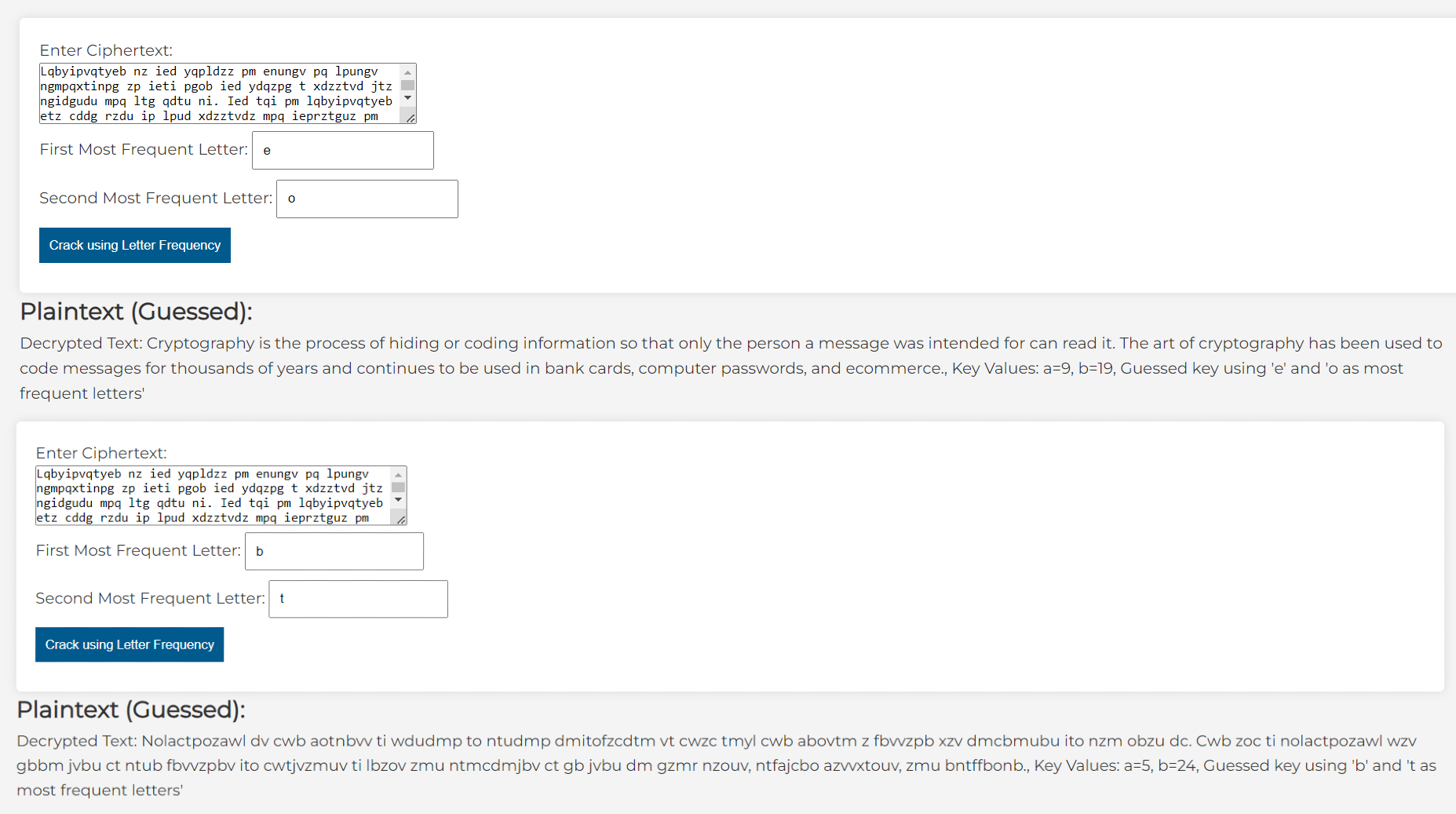
****

Figure 59: Sixth Case using Crack using Letter Frequency

Though the outputs depicted are expected, it is important to note that the cracking effectiveness is **extremely dependent on the user’s input**, hence a good guess for **firstletter** and **secondletter** are necessary for a **correct** result.

#### Conclusion:

All outputs are within expectations; however the cracking **accuracy** is **extremely dependent on the users input** due to the English language properties and letter frequency guesses (discussed in **detail in the literature review section**)

### Cracking using Top Brute Force:

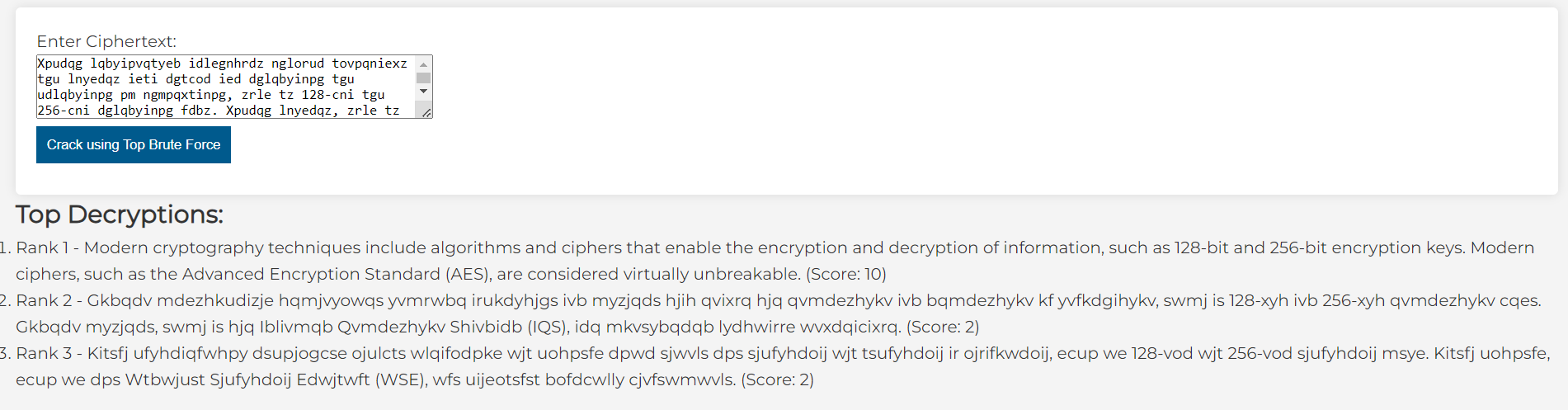
****

Figure 60: First Case using Crack using Top Brute Force

The output is expected, 3 results are outputted with respective **rank**, clearly the **first result** is the correct decryption as it has the **higher score** by a large difference, and is **clearly intelligible.**

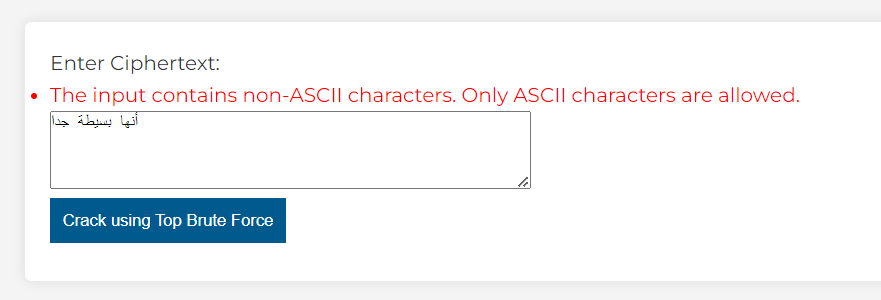
****

Figure 61: Second Case using Crack using Top Brute Force

The output is expected, like the rest of the website, only **ASCII** characters are **allowed** for the **ciphertext** input.

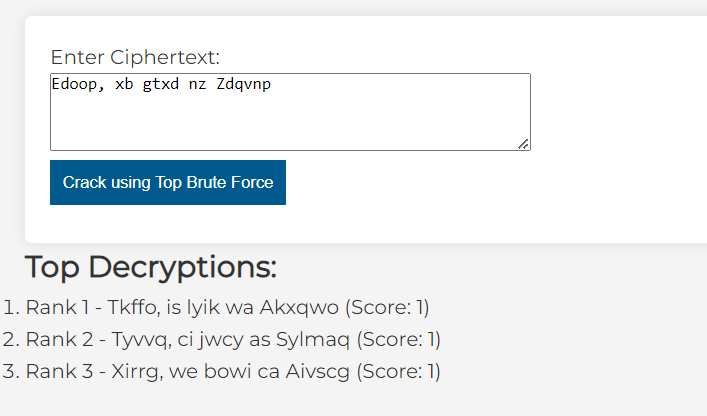


Figure 62: Third Case using Crack using Top Brute Force

The output is expected, a proper ranking **isn't shown** as the ciphertext is not long enough to give an accurate guess of which **brute force result** is ranked **higher**.

#### Conclusion:

All outputs are within expectations; however the fact remains that the **accuracy** of the result is **heavily dependent** on the **length** and **quality** of the **ciphertext** (discussed in **detail in literature review**).

## Vigenère:

### General Encryption and Decryption:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure : First Case using General Decryption Vigenère

Figure : First Case using General Encryption Vigenère

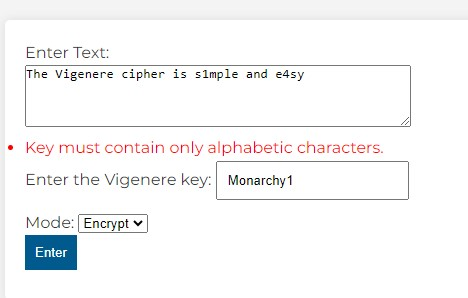
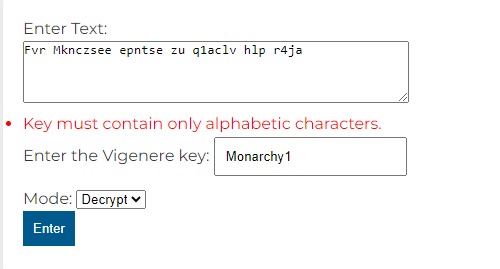
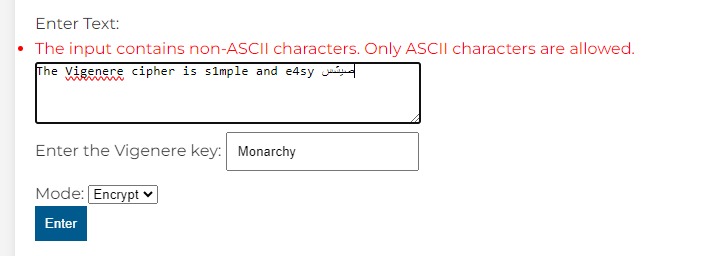
The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, **keyword** input is **monarchy added to the plaintext mod 26** and finally any new lines are propagated into the result.

Figure : Second Case using General Encryption Vigenère

The output is as we’d expect it, since the key contains **non-alphabetical characters**, no result is outputted.

Figure : Second Case using General Decryption Vignere

A screenshot of a computer

Description automatically generated

Figure : Third Case using General Encryption Vigenere

Figure : Third Case using General Decryption Vigenere

The output is as we’d expect it, since the input contains **non-ASCII characters**, no result is outputted and a warning (generated by a **forms.py validator**) is given to use only **ASCII** **characters**.

# 

# 

# 

## Monoalphabetic:

### General Encryption and Decryption:

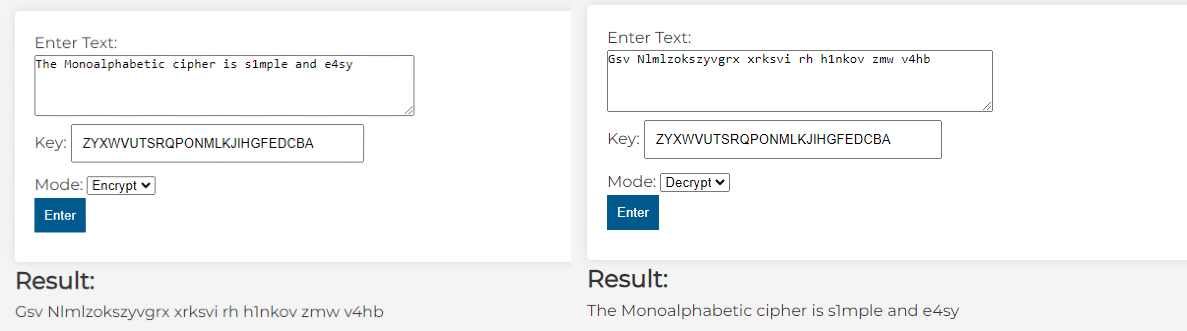


Figure 69: First Case using General Encryption and Decryption Monoalphabetic

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, **keyword** input is **26 letter long word added to the plaintext mod 26** and finally any new lines are propagated into the result.



Figure 70: Second Case using General Encryption and Decryption Monoalphabetic

The output is as we’d expect it, since the key contains **non-Unique-Alphabet characters**, no result is outputted and a warning (generated by a **forms.py validator**) is given to use only **Unique-Alphabet characters.**

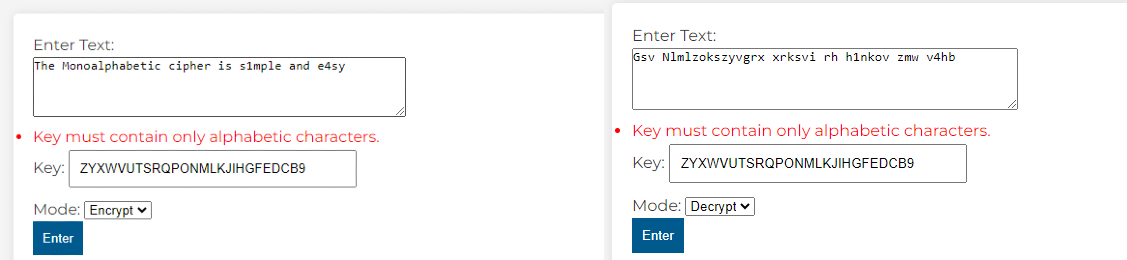


Figure 71: Third Case using General Encryption and Decryption Monoalphabetic

The output is as we’d expect it, since the key contains **a number,** no result is outputted and a warning (generated by a **forms.py validator**) is given to use only **Unique-Alphabet characters.**

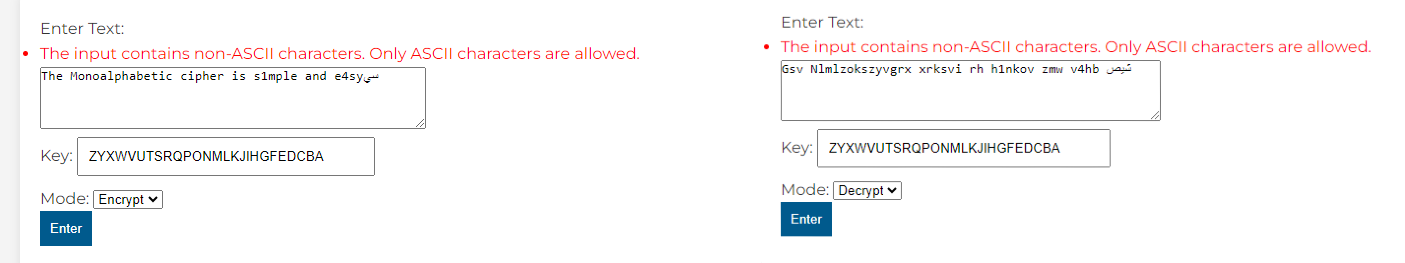
****

Figure 72: Fourth Case using General Encryption and Decryption Monoalphabetic

The output is as we’d expect it, since the input contains **non-ASCII characters**, no result is outputted and a warning (generated by a **forms.py validator**) is given to use only **ASCII** **characters**.

## Playfair Cipher

### General Encryption and Decryption:

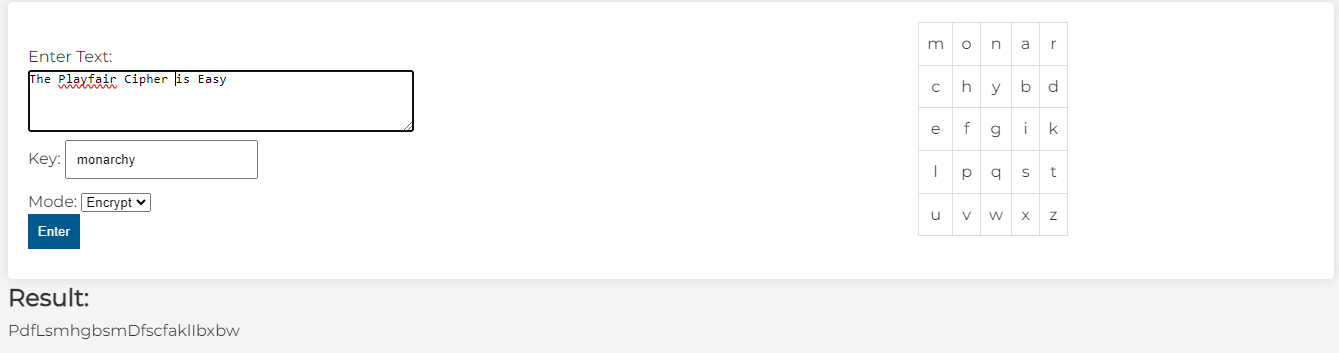
The result of this test is correct. As shown in the results section, the capital letters are preserved, however the spaces are not (discussed in the methodology section).

Figure : First test of the Playfair cipher

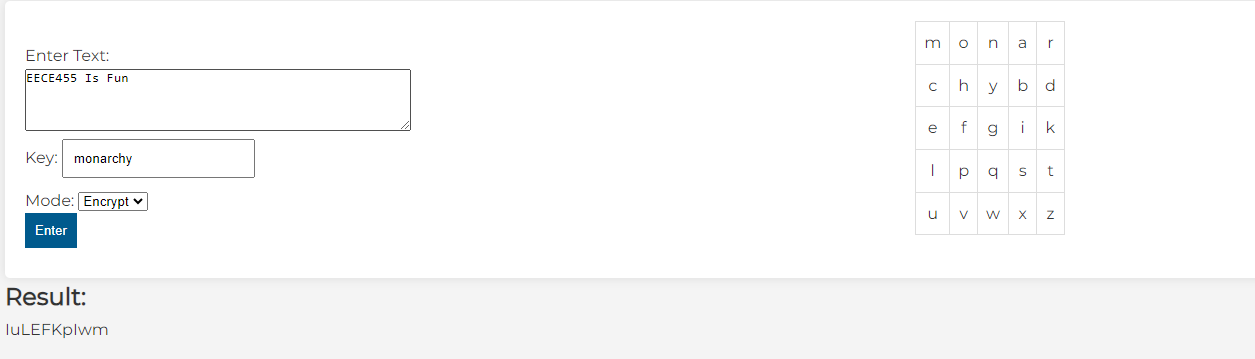
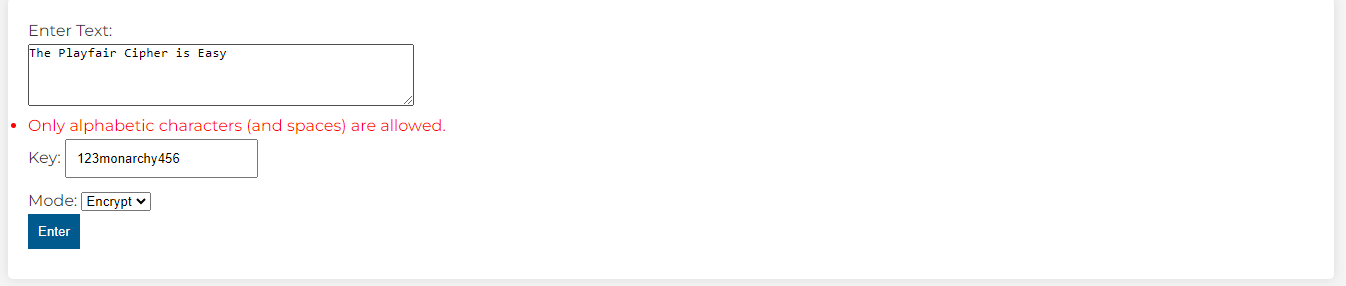
When a non-letter character (not a space also) is entered, it should return an error.

Figure : Second test of the Playfair Cipher

Figure : Third test of the Playfair Cipher

In this example, we can see that the numbers in the text are ignored, but the result for encrypting the remaining characters is correct. In the results, the second letter (u) is a padding since the first diagram (EE) consists of similar letters. As discussed in the methodology section, the padding is chosen to be small x, so with encryption it turned to u.

Figure : Fourth test of the Playfair Cipher

This decryption test returned the correct answer. We can see that there is x at the end, this represents that padding was needed to make the text of even length. Regarding the key, we notice that 3 out of the 4 letters are repeating, and it is evident from the table that the was only considered once.

## Hill Cipher:

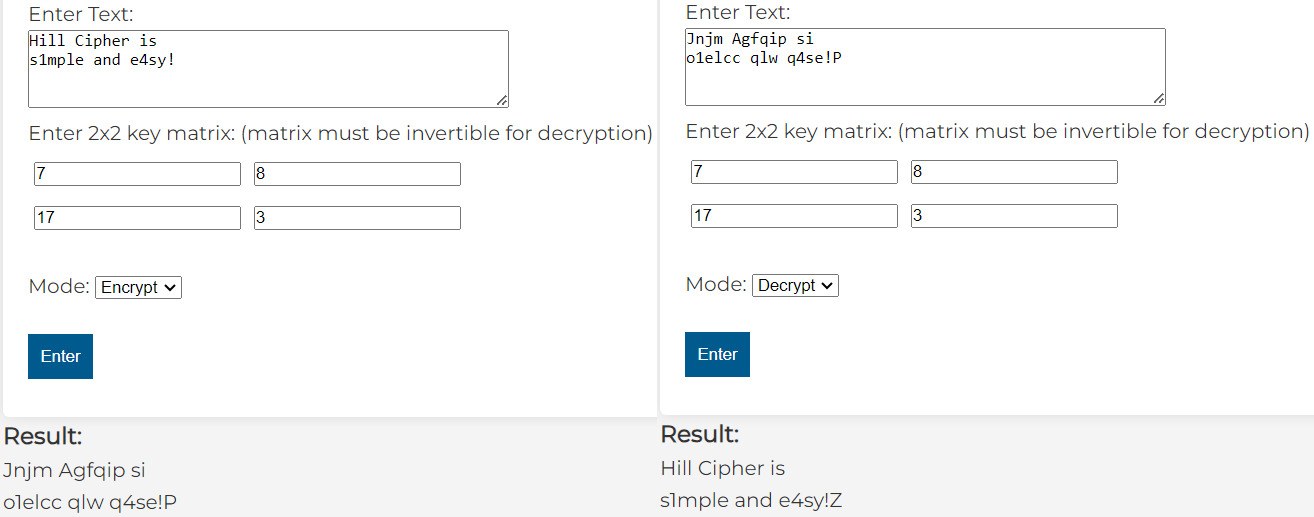
****

Figure : First Case using 3x3 key matrix for Hill Cipher

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, and any new lines are propagated into the result.

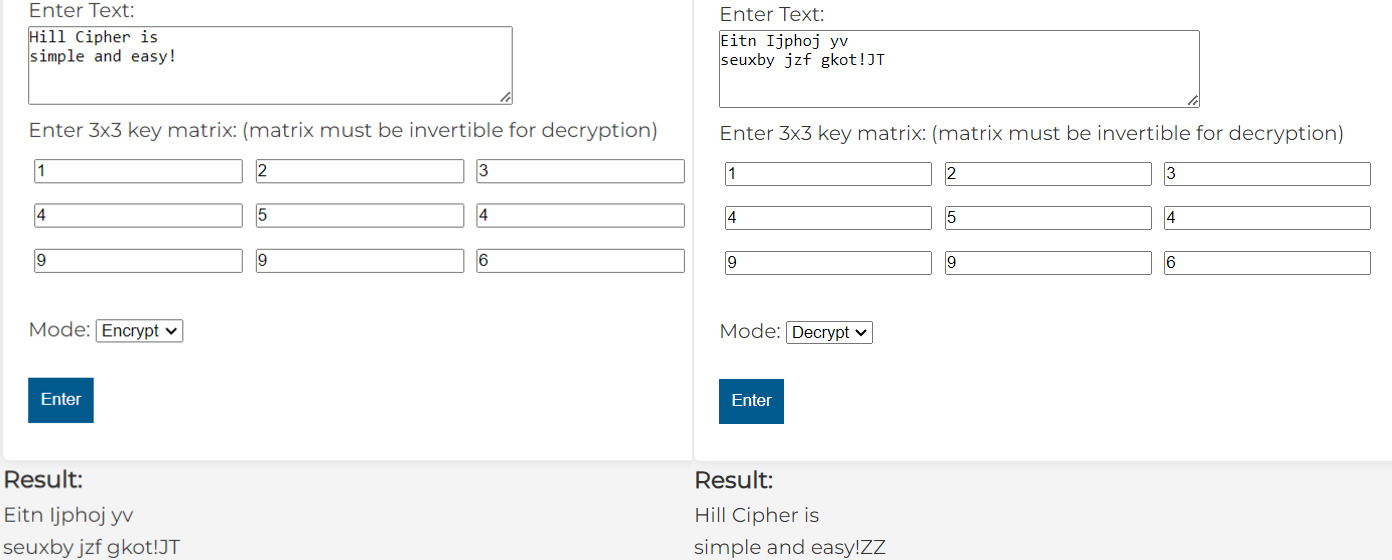


Figure : Second Case using 3x3 key matrix for Hill Cipher

The output is as we’d expect it, anything non-alphabetic is ignored and kept as is, letter capitalization is maintained, and any new lines are propagated into the result.

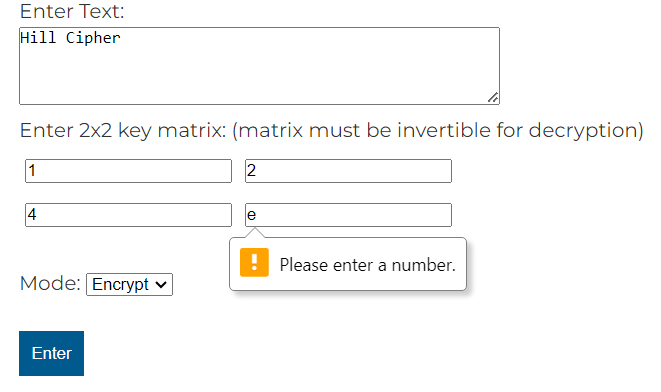


Figure : Third Case for Hill Cipher

If the user inputs a letter in the key matrix, an error is displayed, and the code doesn’t run.

## Extended Euclidean Algorithm:

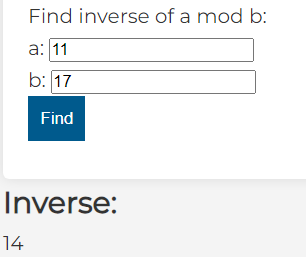
****

Figure : First Case for Extended Euclid

The output is as we’d expect it, 14 = 11^-1 mod 17

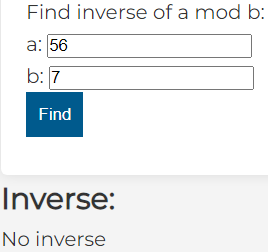
****

Figure : Second Case for Extended Euclid

The output is as we’d expect it, 56 mod 7 has no inverse.

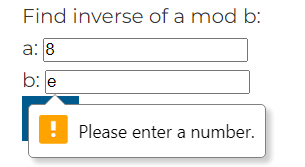
****

Figure : Third Case for Extended Euclid

If the user inputs a letter in a or b, an error is displayed, and the code doesn’t run.

# Conclusion:

Our journey through classical encryption techniques outlined in this report has been characterized by a strategic blend of effective time management, meticulous problem-solving, and an unwavering commitment to quality. Our team's deliberate planning and adherence to well-defined timelines ensured a systematic and thorough exploration of each encryption technique without compromising on depth.

Encountering challenges and errors is an inherent aspect of any endeavor, and our team demonstrated diligence and competence in addressing these issues. Collaborative problem-solving, transparent communication, and a shared dedication to excellence allowed us to navigate unforeseen obstacles seamlessly. The result was not only the resolution of errors but also continuous refinement and optimization of our work.

This project served as a rich learning experience for our team. Exploring classical encryption techniques deepened our understanding of cryptography and sharpened our practical problem-solving skills. The challenges we faced provided valuable opportunities for growth, fostering adaptability and resilience. Collaborating on this endeavor allowed us to apply theoretical knowledge in a real-world context, contributing not only to our understanding of encryption but also to our overall development as individuals and as a team.

Looking ahead, the potential for further enhancement and expansion of our project is evident. As the landscape of cryptography evolves, so too will our website. Future updates may involve incorporating newer encryption methodologies, exploring hybrid models, or delving into new algorithms. By staying attuned to emerging trends and maintaining an agile development approach, our platform will evolve as a dynamic resource for cryptography.

# References

1. Wiki, C. T. C. (n.d.). *Affine cipher*. Crypto Wiki. <https://cryptography.fandom.com/wiki/Affine_cipher>

2. *Affine cipher - weaknesses*. (n.d.).

<https://www.liquisearch.com/affine_cipher/weaknesses>

3. *Cryptanalysis*. (n.d.-b).

<https://www.uobabylon.edu.iq/eprints/paper_9_4879_1380.pdf>

4. Finkelstein, O. (2022, October 19). *Brute-force Attack - Secret Double Octopus*. Secret Double Octopus -. [https://doubleoctopus.com/security-wiki/threats-and-tools/brute-force-attack/#:~:text=Brute%2Dforce%20attack%20is%20an,enumerating%20and%20trying%20encryption%20keys.](https://doubleoctopus.com/security-wiki/threats-and-tools/brute-force-attack/%23:~:text=Brute%2Dforce%20attack%20is%20an,enumerating%20and%20trying%20encryption%20keys.)

5. Chowdary, C., & Chowdary, C. (2023, October 11). *Vigenère Cipher*. Intellipaat. [https://intellipaat.com/blog/vigenere-cipher/#:~:text=The%20autokey%20method%20for%20Vigenere,manually%20in%20a%20short%20time.](https://intellipaat.com/blog/vigenere-cipher/%23:~:text=The%20autokey%20method%20for%20Vigenere,manually%20in%20a%20short%20time.)

6. *What is Monoalphabetic Cipher in Information Security?* (n.d.). <https://www.tutorialspoint.com/what-is-monoalphabetic-cipher-in-information-security>

7. *PlayFair Cipher Implementation in Python - JavatPoint*. (n.d.). www.javatpoint.com. <https://www.javatpoint.com/playfair-cipher-implementation-in-python>