Number Theory and its Applications in Cryptography

Authors: Connor Kardokus & Cole Perry

1. Introduction

1.1. What is Modular Arithmetic and Congruence?

In simple terms modular arithmetic is how we find the remainder of an integer when it is divided by another integer. Modern modular arithmetic was developed by Carl Friedrich Gauss in 1801, and has a wide variety of applications. For example, the 12 hour clock uses modular arithmetic, just to name a simple example. Congruence is an application of modular arithmetic where two integers a and b are said to be congruent if their difference is divisible by some integer m (the modulus), and is written in the form: $a \equiv b \pmod{m}$ or $m|(a - b)$. The Chinese Remainder Theorem is another useful application of modular arithmetic that can be used to solve systems of linear congruence where $X \equiv ai \bmod mi$, $i = 1, 2, . . . , r$. The Chinese remainder theorem is also used in cryptography, one such example being its use in the RSA encryption algorithm. There are many properties and applications of modular arithmetic, but we will be focusing on the applications of modular arithmetic listed above and their usefulness in cryptography.

1.2. Importance of GCD and Extended Euclidean Algorithm

In order to solve linear congruence and systems of linear congruence it is necessary to find the greatest common divisor(GCD) of numbers. For example, when solving a linear congruence in the form $ax \equiv b \pmod{m}$, It is essential to check the GCD of a and m, as a only has a unique multiplicative inverse under modulo m if the GCD of a and m is equal to one. The extended Euclidean algorithm is also important for implementing the affine cipher. The extended Euclidean algorithm provides the multiplicative inverses of a and m, which is necessary for the decryption phase of the affine cipher. The affine cipher also relies on knowing the GCD of the x coefficient and size of the alphabet(26), when using the function for the affine cipher shift: $E(x) = (ax + b) \bmod m$, in the encryption phase. In order for the affine cipher to work properly the GCD of a and m must be equal to one, as there needs to be a unique multiplicative inverse of a in order for the cipher to work properly. This is due to the fact that if there is not a unique solution, the shift could be multiple different values, making it impossible to encrypt and decrypt the data correctly.

1.3. Applications of Modular Arithmetic in Caesar and Affine Ciphers

In cryptography modular arithmetic can be used to create simple ciphers such as the Caesar cipher or affine cipher. In each of these ciphers all letters a-z are given a numerical value 1-26, and the encrypted data is shifted by a certain amount, the key, before converting the numerical values back into letters to create an encrypted message that is unintelligible to someone without the key. In the case of both the Caesar and affine ciphers, the shift may cause the numerical value of a character in the string to fall out of range 1-26. Modular arithmetic is used in both ciphers to make sure all shifted values are correctly adjusted to be in range of the numeric values of the alphabet during both the encryption and decryption of data. This ensures that all the characters in the encrypted and decrypted data are properly stored, and returned so that the data stays intact through encryption and decryption.

1.4. Our Number Theory and Cipher Application

To apply our knowledge of number theory, modular arithmetic and their usefulness in creating ciphers, we have created an application in python that does the following: 1) Find the GCD of two numbers using the Euclidean algorithm, 2) Find the Bezout coefficients of two numbers using the extended Euclidean algorithm, 3) Solve systems of linear congruence using the Chinese Remainder theorem, 4) Encrypt strings using a Caesar cipher, 5) Decrypt strings using a Caesar cipher, 6) Encrypt strings using an affine cipher, and 7) Decrypt strings using and affine cipher.

2. Analysis

2.1. Analysis Intro

We decided to do our project using Python. This is the language we learned in 5001 this quarter so it made the most sense to combine the theories we learned in 5002 with our new python skills. Our goal was to implement as many theorems that we learned in class (mostly the NumberTheory section) with classes and functions in Python. Here is a summary of each file, including what we did, why, and the steps we took to get there.

2.2. Main.py

This file is used as our script. We created a “menu” with the following options: “0” - Terminate the program, “1” - Find greatest common divisor, “2” - Find bezout’s coefficients, “3” - Caesar cipher encryption, “4” - Caesar cipher decryption, “5” - Solve Chinese Remainder Theorem, “6” - Affine cipher encryption, and “7” - Affine cipher decryption. Within the main script, we started by creating an infinite loop to implement efficient error handling. If a user inputs the wrong text, we want them to come right back to the main menu to try again. If the user selects choice “0” - we simply exit the loop by using a break statement. Without going into every minute detail, here is an overview of the error handling we implemented:

- For gcd and bezout’s functions, we only allow users to enter an integer
- For bezout’s theorem, the GCD of x and y has to equal to 1 for it to run
- For the Caesar cipher encryption and decryption, the shift has to be entered as an integer
- For the Affine cipher, the user has to enter the x coefficient and constant as integers

In addition to the error handling, we also implemented user input for and classes/instances to make the code more efficient. For example, within the GCD menu selection, we gather x and y variables from the user, create an instance of the GCD class with x and y variables using self, and then call the gcd function on the instance to obtain the final result. Within the Bezout’s Theorem selection in the main script, we gather x and y variables from the user and create another GCD class instance to find the GCD. If the GCD is equal to 1, we proceed to using the instance we created to call the bezout’s function, printing out our s & t coefficients. We largely follow the same methods in main.py with the Caesar cipher encryption and decryption, Chinese remainder theorem, and Affine cipher encryption and decryption of:

- Gather user input as the arguments
- Create an instance of the class and passing the user input
- Call the appropriate method on the instance

2.3. GCD via the Euclidean Algorithm

The Gcd.py file is used to create the Class GCD which stores 3 variables: an integer x, an integer y, and a variable list to store variables for the bezout’s theorem. Within the GCD class we created the gcd function which outputs the greatest common divisor between the two numbers and only takes in one argument: self. This is because the x and y variables can be obtained directly from the instance we created in the main.py file using “self.x” and “self.y.” To begin solving this function, I grabbed my notes from class and started looking for patterns. I realized the first thing we did was divide y by x, and continue until the remainder was 0. I began the code with a while loop: “while y%x !=0.” I then created two more variables, “coefficient,” and “remainder.” During each loop, I calculated the coefficient as the floor of (y/x), the remainder as y%x, and then stored each variable y, x, coefficient, and remainder in the variable list as a tuple. To complete the function, I then updated the y variable to equal x, and the x variable to equal the remainder so the correct values will be used for the next loop. I then simply returned x, which is the GCD once y%x = 0. Overall, this function was quite simple but I had to come back to add the tuple functionality once I began working on Bezout's Theorem.

2.4. Bezout’s Coefficients via extended Euclidean

The Bezout function computes the coefficients of Bézout's identity for the given numbers. Bézout's identity states that for any integers a and b, there exist integers x and y such that: a * x + b * y = gcd(a, b). This function is within the GCD class. To begin, we obtain the last element of the tuple we created in the GCD function which contains the y, x, coefficient, and remainder for each loop. We begin at the end of the list because we have to work from the bottom up to find the coefficients. We first initialize the x coefficient as 1 to track the coefficient of the largest number, and the second coefficient as the negative of the coefficient on the last loo. We make the second coefficient negative so we can return the answer in the form s * x_coefficient + t * y_coefficient = gcd. Our loop begins from the second to last element or tuple in the list. This is because we need to use the “next” element to perform the updates to the coefficient. In each loop, we calculate the new coefficients s and t and then update the x and y coefficients for the next loop. At the end of the function, we return the variables s & t.

2.5. Systems of linear congruence via Chinese Remainder Theorem

The chinese_remainder.py file contains the necessary code to solve a system of linear congruences using the Chinese Remainder Theorem (CRT). In the main.py file if the user enters “5” choosing to solve a system of linear congruences they are prompted to enter the number of linear congruences in the system, and this variable is stored as “N.” Next an instance of the ChineseRemainder class is called, and then the method chinese_remainder is called with the variable N as its argument. Knowing how many linear congruences are in the system is key for the functionality of the chinese_remainder method, as N is used to set the range in all the for loops so that the program can work for a system that includes any number of linear congruences, including the first for loop that prompts the user to enter the “r” and “m” values for each linear congruence in order, and stores each of these values in a list, the “r_list” and “m_list” respectively. Additional empty lists were also created to store variables necessary for the CRT including “M”, “s”, and “x” titled “M_list”, “s_list” and “x_list” respectively.

Next the program checks that each m_i value is coprime with all other m_i values, as all m_i values must be coprime in order for the CRT to be implemented. It does this by using a nested for loop, calling the gcd function within the nested for loop to check the gcd of all m_i. Starting with m_1 the program checks it the GCD of m_1 and m_2 and all other m_i through m_N. Then depending on how many systems are in the congruence, it checks the GCD of m_2 and m_3 and all other m_i through m_N and so on until the GCD of every m_i pair has been calculated. This is done by setting the range of the first for loop to zero through N, and the second for loop range to i + 1, N, where “i” is the loop number of the first for loop. It checks each GCD as they are calculated, and if any of the GCD are not equal to one (the m_i are not coprime) the program prints a message to the user informing them that there is no unique solution and the program is ended.

If the user enters a valid set of linear congruences in which all m_i are coprime the program continues by calculating “M” using the formula *M* = Π *m_i* for all m_i in the “m_list”, initializing the variable “M” as 1, and then multiplying it by each successive m_i using a for loop. The program then finds each M_i by dividing M by m_i for each “m” in the “m_list” and adds those values to the “M_list” using a for loop to iterate through each “m” value. Next each s_i value is calculated using $s_i = M_i^{-1}$ mod m_i where M_i^{-1} is the multiplicative inverse of M_i. This is done by using a for loop to call the gcd and bezout method to find M_i^{-1}. Within the loop it checks if each s_i value is in range of zero through m_i and adjusts s_i using modular arithmetic if needed before adding each s_i value to the “s_list.” Finally each x_i is calculated using the formula $x_i = r_i \times s_i \times M_i$, using a for loop to access the proper variables from all the “r_list”, “s_list” and “M_list” respectively. Finally x_0 is calculated by summing all x_i in the “x_list” and the formula $X = x_0 mod M$ to find the solution to the system of linear congruences.

2.6. Caesar Cipher

In the caesar.py file the CaesarCipher class is initialized with a “shift” argument, the amount the numerical values will shift in the encryption and decryption phase. The initialization also includes a dictionary with the numerical values for each letter in the alphabet. The encrypt and decrypt method are essentially the same, except for how they alter the numerical value of each character. Both methods take a string argument and convert all letters in the string to upper case, as the letter keys in the dictionary are all upper case. Then a variable is created to store the encrypted or decrypted string respectively, and is initialized as an empty string. A for loop iterates through each character in the string, and if the character is a letter, the numeric value is found in the dictionary, stored in a variable “numeric_value” and then used to calculate the shifted value, stored in a variable “shifted_value.” The shifted values for the encryption uses the formula shifted_value = $(x + s - 1)\, mod\, (26 + 1)$ and decryption uses shifted_value = $(x - s - 1)\, mod\, (26 + 1)$ where “x” is the numeric value of the character and “s” is the shift value. In order to make the calculation work in python an adjustment of minus one on the left side, and plus one on the right side was needed. This is because the dictionary of numeric values starts at one and not zero, and therefore if shifted_value was equal to zero, the program would not work as no letter would be returned. Once the shifted value is calculated, the program finds the matching letter for that value and adds it to the encrypted or decrypted string respectively. In the initial for loop, if-else logic is used so that if the character is not a letter, it is simply added to the string as is. Once the for loop has iterated through each character in the string it returns the encrypted or decrypted string for the encrypted and decrypted methods respectively.

2.7. Affine Cipher

The affine.py file the program is identical to the Caesar cipher program in many areas, but with a few key changes, and an extra method. The affine cipher is also initialized with a constant for the shift, but it also takes a coefficient argument, as the shift formula for the affine cipher encryption is shifted_value = $(ax + b - 1)\, mod(26 + 1)$, and the formula for decryption is shifted_value = $(a^{-1}\, (x - b) - 1)\, mod(26 + 1)$ where “a” is the coefficient, “a^-1” is the multiplicative inverse, “x” is the numeric value of the letter, and “b” is the shift constant. Again including the minus one and plus one adjustments for the same reasons as in the Caesar cipher. The AffineCipher class also includes a method “check_key” to make sure the x-coefficient in the shift is coprime with 26 to ensure that there is a unique solution to the shift key formula, as is necessary for effective encryption and decryption. If-else logic is used to check the x-coefficient, and end the program and print a message to the user if they enter an invalid x-coefficient for the key. Along with this extra method, the function also differs from the Caesar cipher in that it returns a Boolean value along with a string. The Boolean value is used in the main.py file to decide which formatting the string will take, as the formatting for the encrypted and decrypted strings is different from the error message string for increased readability for the user.

3. Conclusion

3.1. Weaknesses

I believe this project was very successful and we were able to implement all of the functionality that we set out to from the beginning. We successfully obtained the correct results for all functions based on various testing as well as properly functioning error handling. The one weakness that we found in our code was in the application of bezout’s theorem. We originally solved this function only for the case that the gcd = 1, or if ax = 1modm. Bezout's theorem can also be solved for cases where the gcd is not equal to 1, or ax = bmodm, and this would result in multiple answers. I believe that if we had more time this would be the next thing we would solve. As for other weaknesses, we believe that the Caesar cipher is weak and can be easily decoded, but is still a good concept and exercise to understand. We believe this because it operates by shifting each letter in the plaintext by a fixed number of positions in the alphabet. This simplicity makes it vulnerable to brute-force attacks, where all possible shifts are tested until the original message is revealed. Another weakness is the time complexity of our algorithms. As we had limited time, we wanted to make sure all our algorithms worked, and did not have time to optimize them for reduced time complexity. For example the method for the CRT execution includes nested for loops, leading to time complexity of O(n^2), which could become problematic for large systems of linear congruence. Also, in the Caesar and affine ciphers there is probably a more efficient way to match the looping through the entire letter value dictionary to find a match. While this is fine for the amount of data we have been testing in these algorithms, the algorithms may not be very efficient for large data sets.

3.2. Areas of Further Research

Avenues for further research would include research on any other possible edge cases we may have missed as well as similar topics such as calculating the least common multiple of two numbers, finding the prime factorization of a number, and finding the congruence of two numbers mod m. In addition to that further research, optimizing our current programs for reduced time complexity would make them better able to handle large amounts of data. Another avenue for research would be to create more complex encryption and decryption methods, possibly using the CRT for encryption and/or decryption.

3.3. What I learned: Cole

This project was a great opportunity to tie in my python skills with concepts learned in our discrete structures course. I was able to practice my use of object oriented programming by building classes for different functions and using those classes in our main.py script to build out and operate a user menu. I also further honed in my skills and understanding on the the concepts of greatest common multiple, bezout's coefficient, chinese remainder theorem, and caesar and affine cipher so if any of these concepts come up throughout my career I will have a great foundation to build from.

3.4. What I learned: Connor

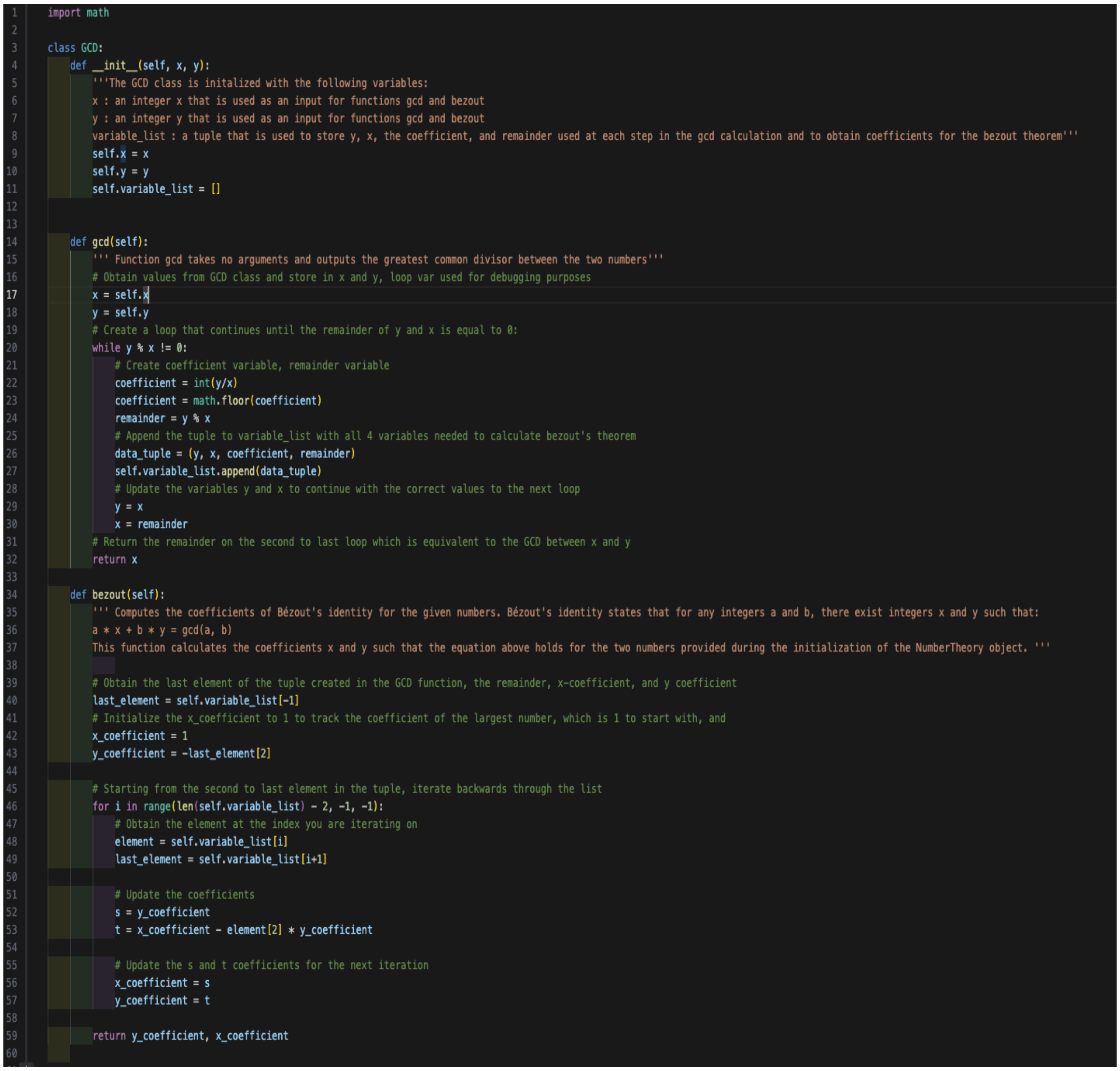
First, this project helped me to think deeper about the concepts covered in the number theory unit. Finding the GCD of two numbers or solving a system of linear congruences using the CRT by hand is much different than writing a program to solve these problems. Math is very intuitive to me, but working on the chinese_remainder.py file, for example, I really had to think about what each variable was doing, how it is calculated, and how it affects the final solution, etc. in order to write an algorithm to implement the CRT. Also, Because of this I also felt that my understanding of how to choose the right data structures for solving a math problem in python improved as well. For example, implementing lists to store variables in the chinese_remainder.py file vs. using a dictionary to store letters and their numeric value in the cipher files. Additionally, I got a lot of practice using loops to iterate through data to store, manipulate, or check each individual element. Next, I am a total beginner with coding, so getting to build more object oriented programs from scratch was a great way to reinforce and improve my understanding of the fundamentals of object oriented programming. I feel this was especially true for writing constructor methods from scratch and figuring out how to determine which arguments I need to pass to the constructor and other methods, which is something that I have gotten stuck on in the past. Finally, I learned how much I enjoy combining math and coding. I like both individually, but combining the two can be even more fun and challenging.

4. Appendix

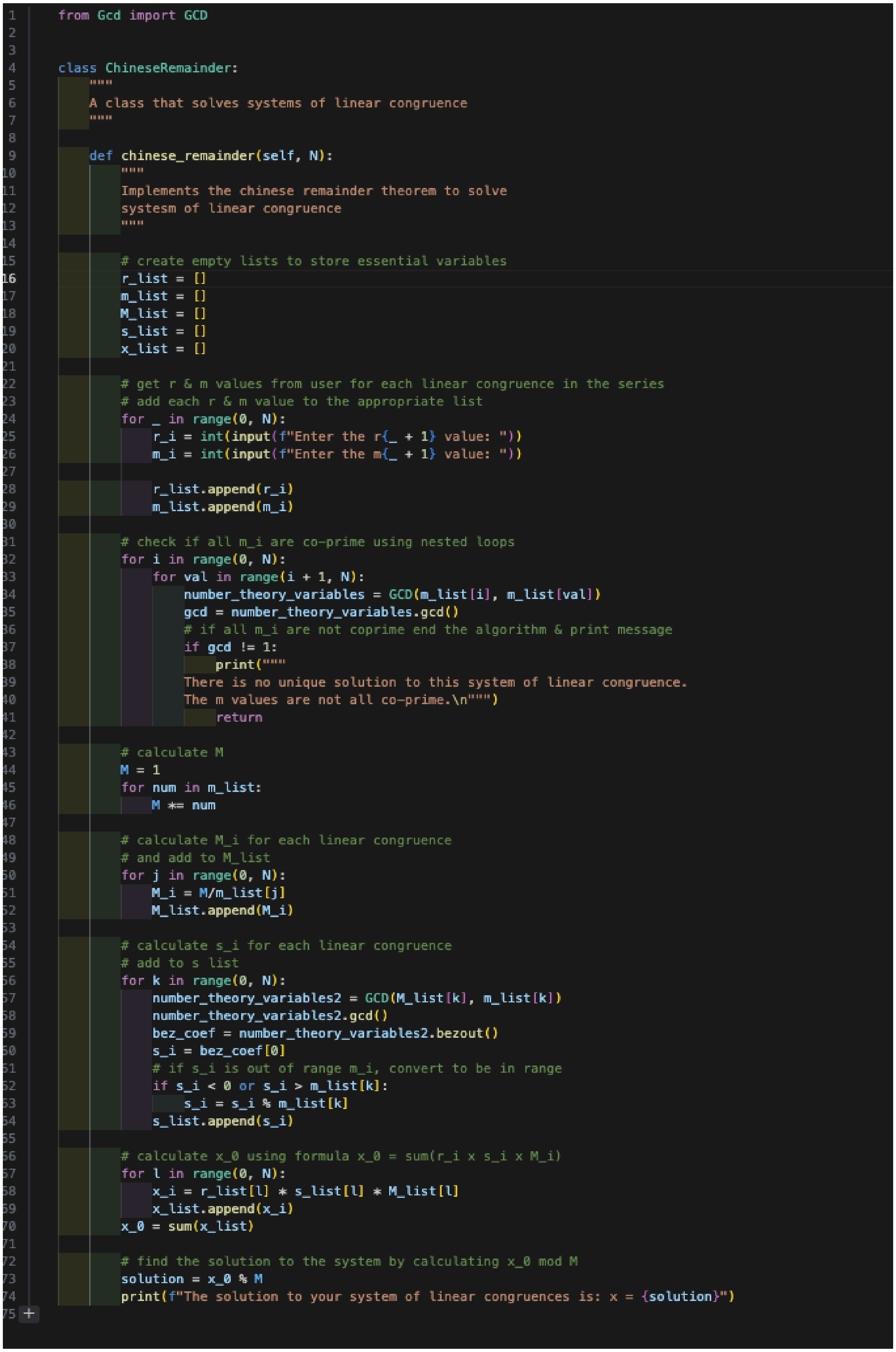
4.1. Sources

1. Lecture notes
2. https://en.wikipedia.org/wiki/Modular_arithmetic
3. https://en.wikipedia.org/wiki/Chinese_remainder_theorem#

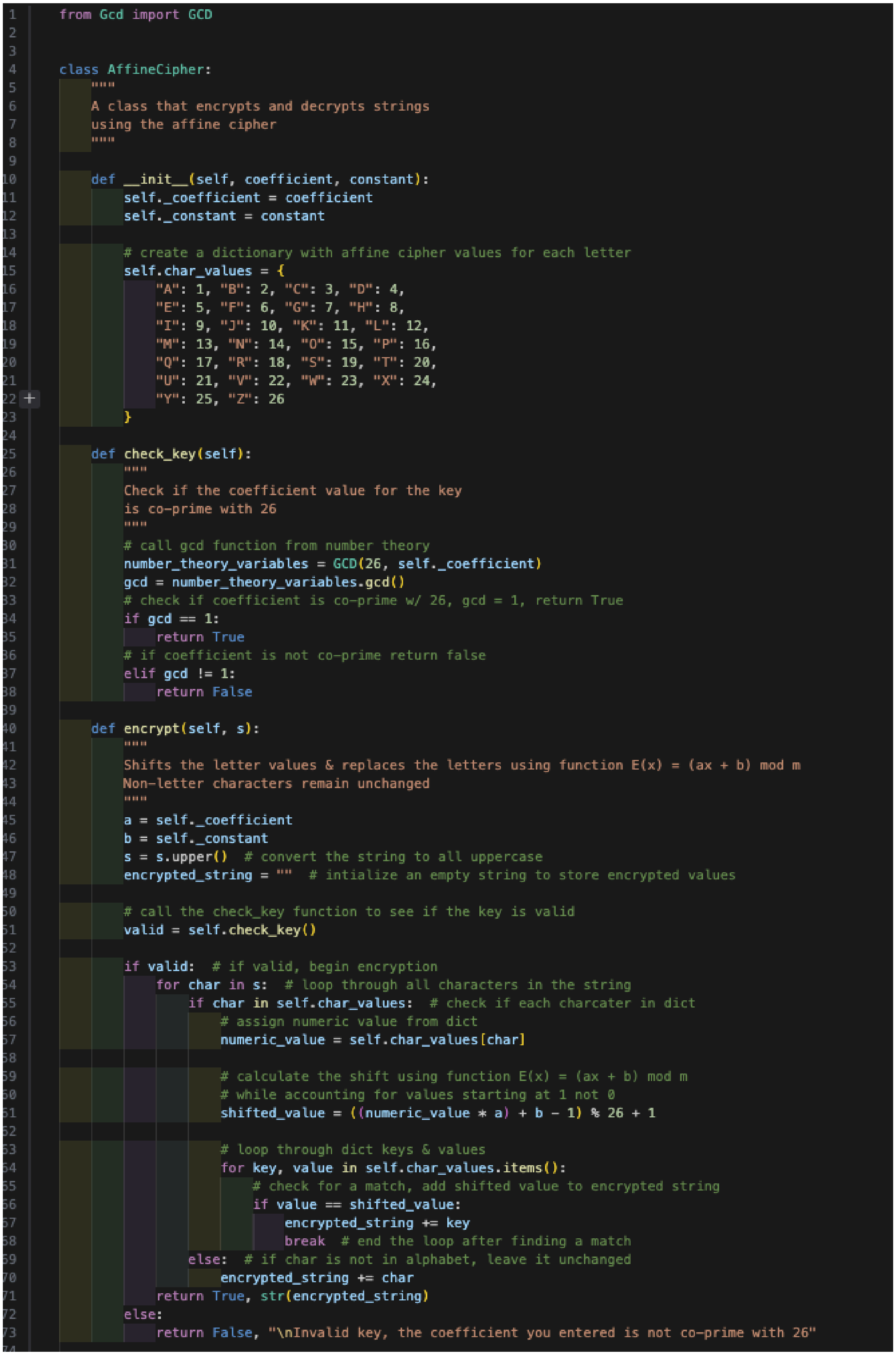
4.2. Source:



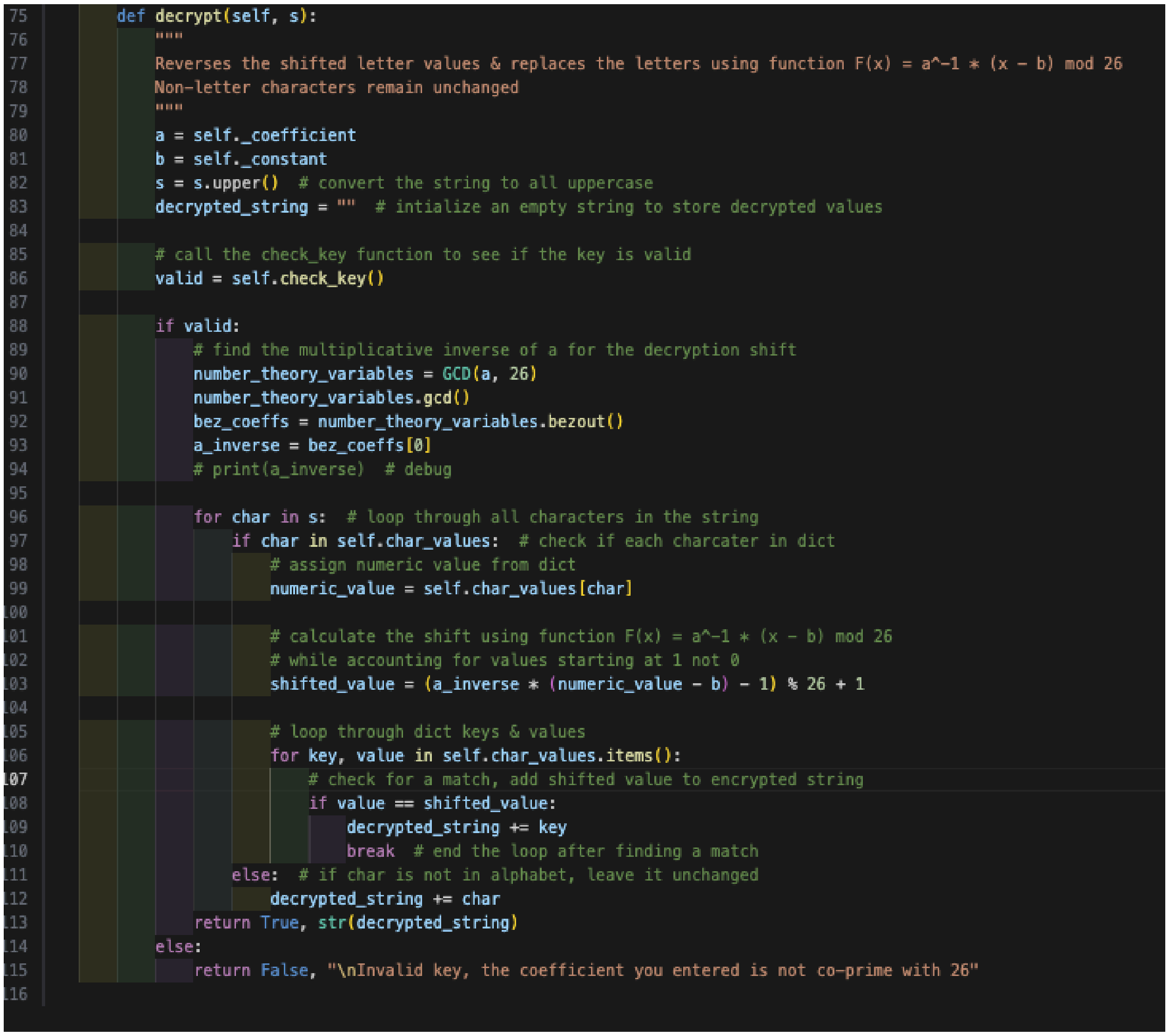
```python
import math

class GCD:
    def __init__(self, x, y):
        '''The GCD class is initalized with the following variables:
        x : an integer x that is used as an input for functions gcd and bezout
        y : an integer y that is used as an input for functions gcd and bezout
        variable_list : a tuple that is used to store y, x, the coefficient, and remainder used at each step in the gcd calculation and to obtain coefficients for the bezout theorem'''
        self.x = x
        self.y = y
        self.variable_list = []


    def gcd(self):
        ''' Function gcd takes no arguments and outputs the greatest common divisor between the two numbers'''
        # Obtain values from GCD class and store in x and y, loop var used for debugging purposes
        x = self.x
        y = self.y
        # Create a loop that continues until the remainder of y and x is equal to 0:
        while y % x != 0:
            # Create coefficient variable, remainder variable
            coefficient = int(y/x)
            coefficient = math.floor(coefficient)
            remainder = y % x
            # Append the tuple to variable_list with all 4 variables needed to calculate bezout's theorem
            data_tuple = (y, x, coefficient, remainder)
            self.variable_list.append(data_tuple)
            # Update the variables y and x to continue with the correct values to the next loop
            y = x
            x = remainder
        # Return the remainder on the second to last loop which is equivalent to the GCD between x and y
        return x

    def bezout(self):
        ''' Computes the coefficients of Bézout's identity for the given numbers. Bézout's identity states that for any integers a and b, there exist integers x and y such that:
        a * x + b * y = gcd(a, b)
        This function calculates the coefficients x and y such that the equation above holds for the two numbers provided during the initialization of the NumberTheory object. '''

        # Obtain the last element of the tuple created in the GCD function, the remainder, x-coefficient, and y coefficient
        last_element = self.variable_list[-1]
        # Initialize the x_coefficient to 1 to track the coefficient of the largest number, which is 1 to start with, and
        x_coefficient = 1
        y_coefficient = -last_element[2]

        # Starting from the second to last element in the tuple, iterate backwards through the list
        for i in range(len(self.variable_list) - 2, -1, -1):
            # Obtain the element at the index you are iterating on
            element = self.variable_list[i]
            last_element = self.variable_list[i+1]

            # Update the coefficients
            s = y_coefficient
            t = x_coefficient - element[2] * y_coefficient

            # Update the s and t coefficients for the next iteration
            x_coefficient = s
            y_coefficient = t

        return y_coefficient, x_coefficient

```



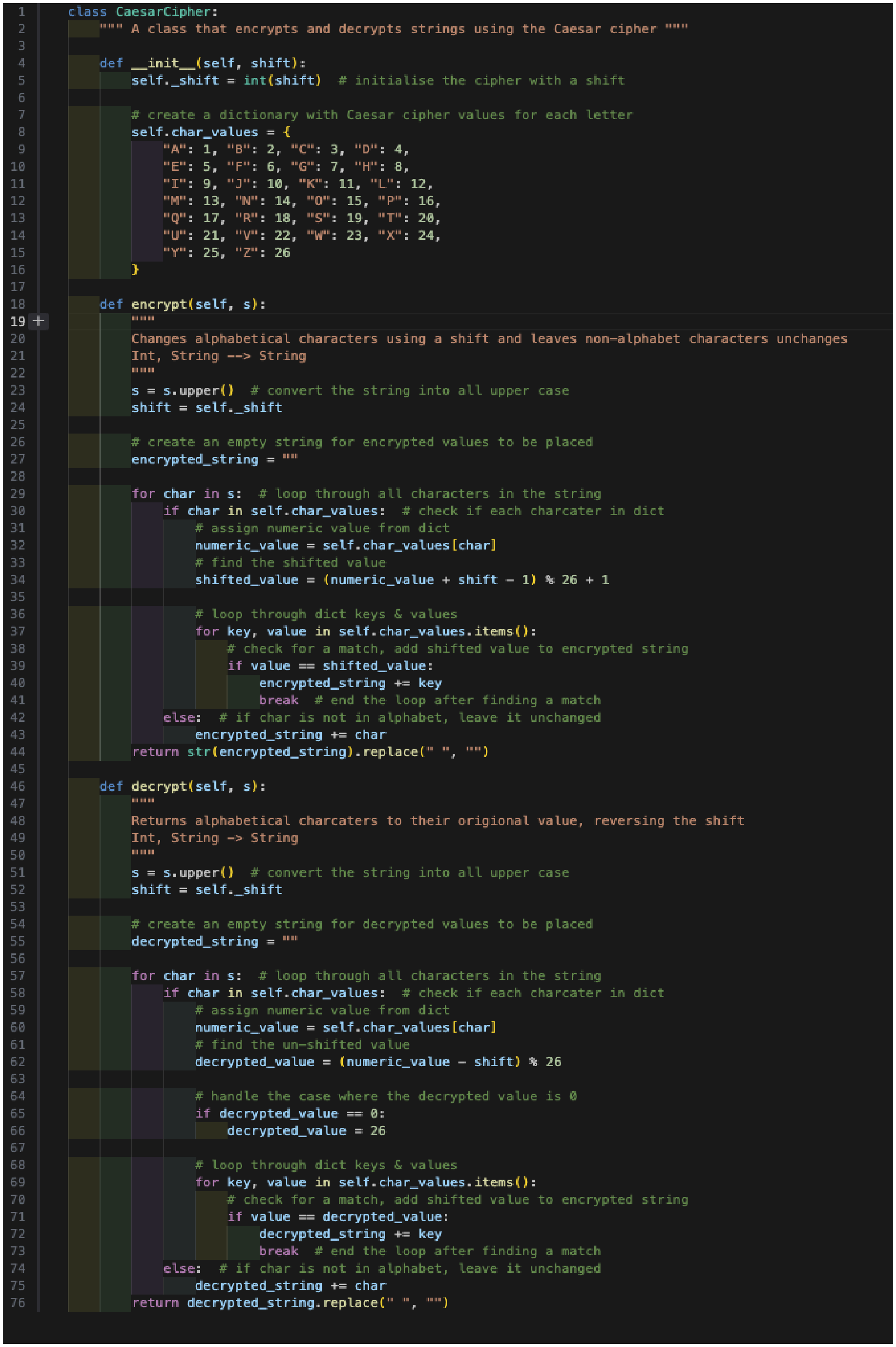
```python
from Gcd import GCD


class ChineseRemainder:
    """
    A class that solves systems of linear congruence
    """

    def chinese_remainder(self, N):
        """
        Implements the chinese remainder theorem to solve
        systesm of linear congruence
        """

        # create empty lists to store essential variables
        r_list = []
        m_list = []
        M_list = []
        s_list = []
        x_list = []

        # get r & m values from user for each linear congruence in the series
        # add each r & m value to the appropriate list
        for _ in range(0, N):
            r_i = int(input(f"Enter the r{_ + 1} value: "))
            m_i = int(input(f"Enter the m{_ + 1} value: "))

            r_list.append(r_i)
            m_list.append(m_i)

        # check if all m_i are co-prime using nested loops
        for i in range(0, N):
            for val in range(i + 1, N):
                number_theory_variables = GCD(m_list[i], m_list[val])
                gcd = number_theory_variables.gcd()
                # if all m_i are not coprime end the algorithm & print message
                if gcd != 1:
                    print("""
                There is no unique solution to this system of linear congruence.
                The m values are not all co-prime.\n""")
                    return

        # calculate M
        M = 1
        for num in m_list:
            M *= num

        # calculate M_i for each linear congruence
        # and add to M_list
        for j in range(0, N):
            M_i = M/m_list[j]
            M_list.append(M_i)

        # calculate s_i for each linear congruence
        # add to s list
        for k in range(0, N):
            number_theory_variables2 = GCD(M_list[k], m_list[k])
            number_theory_variables2.gcd()
            bez_coef = number_theory_variables2.bezout()
            s_i = bez_coef[0]
            # if s_i is out of range m_i, convert to be in range
            if s_i < 0 or s_i > m_list[k]:
                s_i = s_i % m_list[k]
            s_list.append(s_i)

        # calculate x_0 using formula x_0 = sum(r_i x s_i x M_i)
        for l in range(0, N):
            x_i = r_list[l] * s_list[l] * M_list[l]
            x_list.append(x_i)
        x_0 = sum(x_list)

        # find the solution to the system by calculating x_0 mod M
        solution = x_0 % M
        print(f"The solution to your system of linear congruences is: x = {solution}")
```



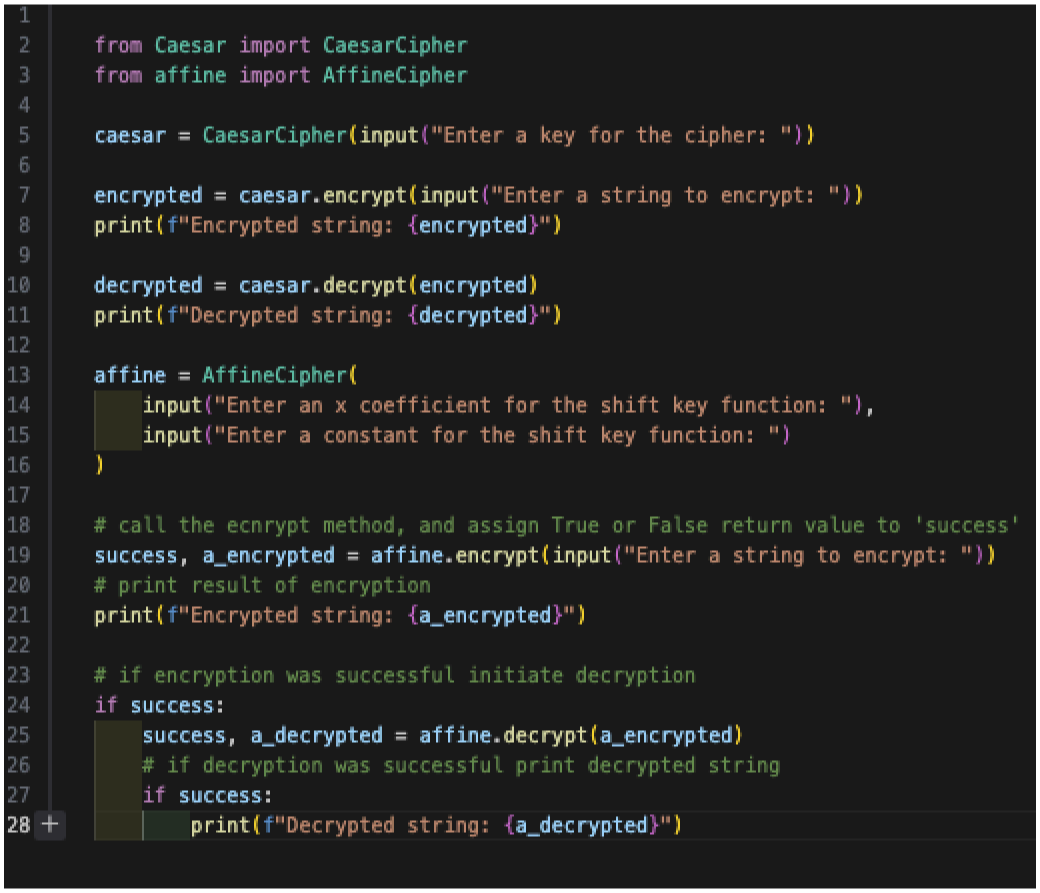
```python
from Gcd import GCD


class AffineCipher:
    """
    A class that encrypts and decrypts strings
    using the affine cipher
    """

    def __init__(self, coefficient, constant):
        self._coefficient = coefficient
        self._constant = constant

        # create a dictionary with affine cipher values for each letter
        self.char_values = {
            "A": 1, "B": 2, "C": 3, "D": 4,
            "E": 5, "F": 6, "G": 7, "H": 8,
            "I": 9, "J": 10, "K": 11, "L": 12,
            "M": 13, "N": 14, "O": 15, "P": 16,
            "Q": 17, "R": 18, "S": 19, "T": 20,
            "U": 21, "V": 22, "W": 23, "X": 24,
            "Y": 25, "Z": 26
        }

    def check_key(self):
        """
        Check if the coefficient value for the key
        is co-prime with 26
        """
        # call gcd function from number theory
        number_theory_variables = GCD(26, self._coefficient)
        gcd = number_theory_variables.gcd()
        # check if coefficient is co-prime w/ 26, gcd = 1, return True
        if gcd == 1:
            return True
        # if coefficient is not co-prime return false
        elif gcd != 1:
            return False

    def encrypt(self, s):
        """
        Shifts the letter values & replaces the letters using function E(x) = (ax + b) mod m
        Non-letter characters remain unchanged
        """
        a = self._coefficient
        b = self._constant
        s = s.upper()  # convert the string to all uppercase
        encrypted_string = ""  # intialize an empty string to store encrypted values

        # call the check_key function to see if the key is valid
        valid = self.check_key()

        if valid:  # if valid, begin encryption
            for char in s:  # loop through all characters in the string
                if char in self.char_values:  # check if each charcater in dict
                    # assign numeric value from dict
                    numeric_value = self.char_values[char]

                    # calculate the shift using function E(x) = (ax + b) mod m
                    # while accounting for values starting at 1 not 0
                    shifted_value = ((numeric_value * a) + b - 1) % 26 + 1

                    # loop through dict keys & values
                    for key, value in self.char_values.items():
                        # check for a match, add shifted value to encrypted string
                        if value == shifted_value:
                            encrypted_string += key
                            break  # end the loop after finding a match
                else:  # if char is not in alphabet, leave it unchanged
                    encrypted_string += char
            return True, str(encrypted_string)
        else:
            return False, "\nInvalid key, the coefficient you entered is not co-prime with 26"
```



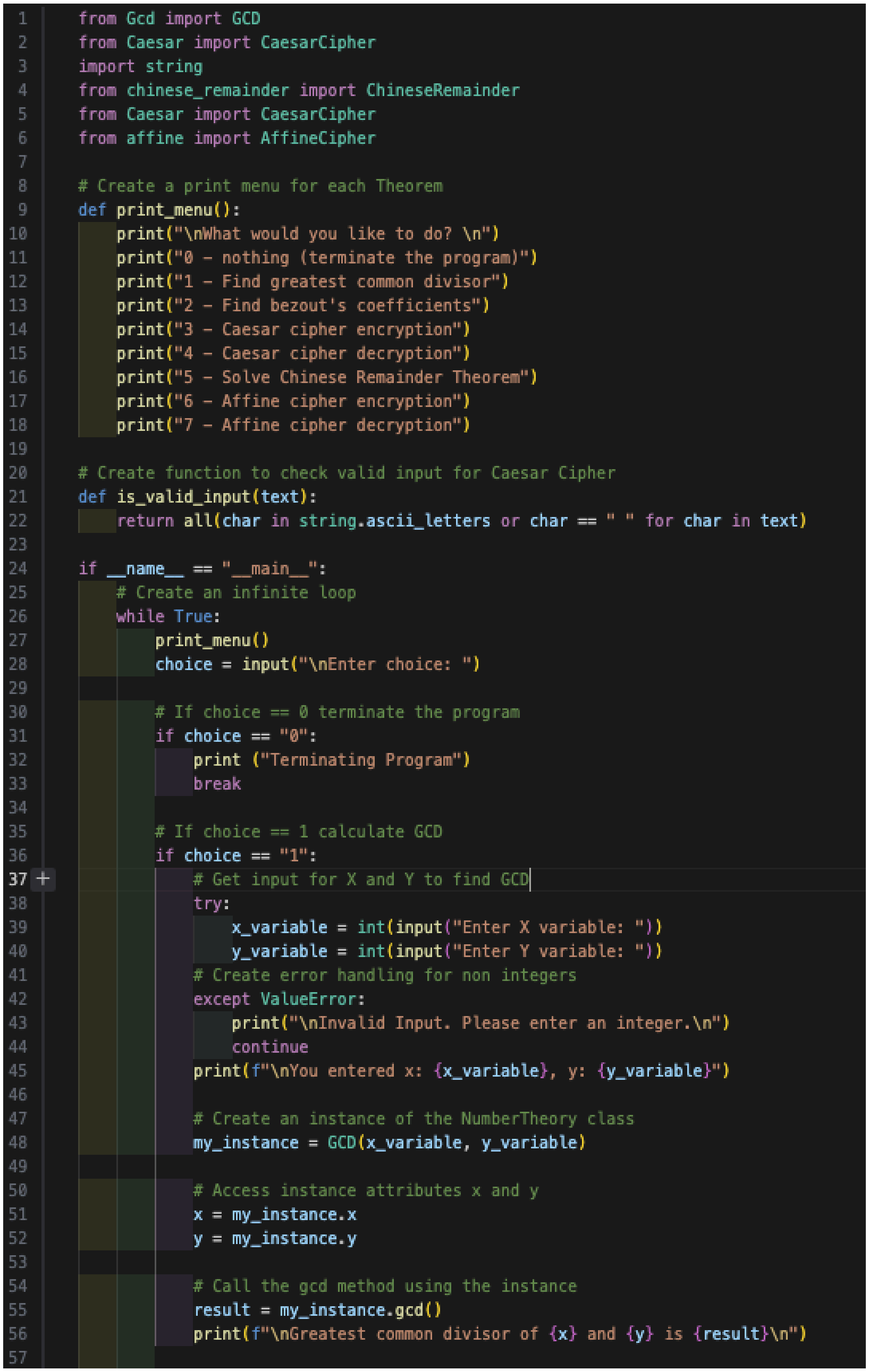
```python
    def decrypt(self, s):
        """
        Reverses the shifted letter values & replaces the letters using function F(x) = a^-1 * (x - b) mod 26
        Non-letter characters remain unchanged
        """
        a = self._coefficient
        b = self._constant
        s = s.upper()  # convert the string to all uppercase
        decrypted_string = ""  # intialize an empty string to store decrypted values

        # call the check_key function to see if the key is valid
        valid = self.check_key()

        if valid:
            # find the multiplicative inverse of a for the decryption shift
            number_theory_variables = GCD(a, 26)
            number_theory_variables.gcd()
            bez_coeffs = number_theory_variables.bezout()
            a_inverse = bez_coeffs[0]
            # print(a_inverse)  # debug

            for char in s:  # loop through all characters in the string
                if char in self.char_values:  # check if each charcater in dict
                    # assign numeric value from dict
                    numeric_value = self.char_values[char]

                    # calculate the shift using function F(x) = a^-1 * (x - b) mod 26
                    # while accounting for values starting at 1 not 0
                    shifted_value = (a_inverse * (numeric_value - b) - 1) % 26 + 1

                    # loop through dict keys & values
                    for key, value in self.char_values.items():
                        # check for a match, add shifted value to encrypted string
                        if value == shifted_value:
                            decrypted_string += key
                            break  # end the loop after finding a match
                else:  # if char is not in alphabet, leave it unchanged
                    decrypted_string += char
            return True, str(decrypted_string)
        else:
            return False, "\nInvalid key, the coefficient you entered is not co-prime with 26"

```



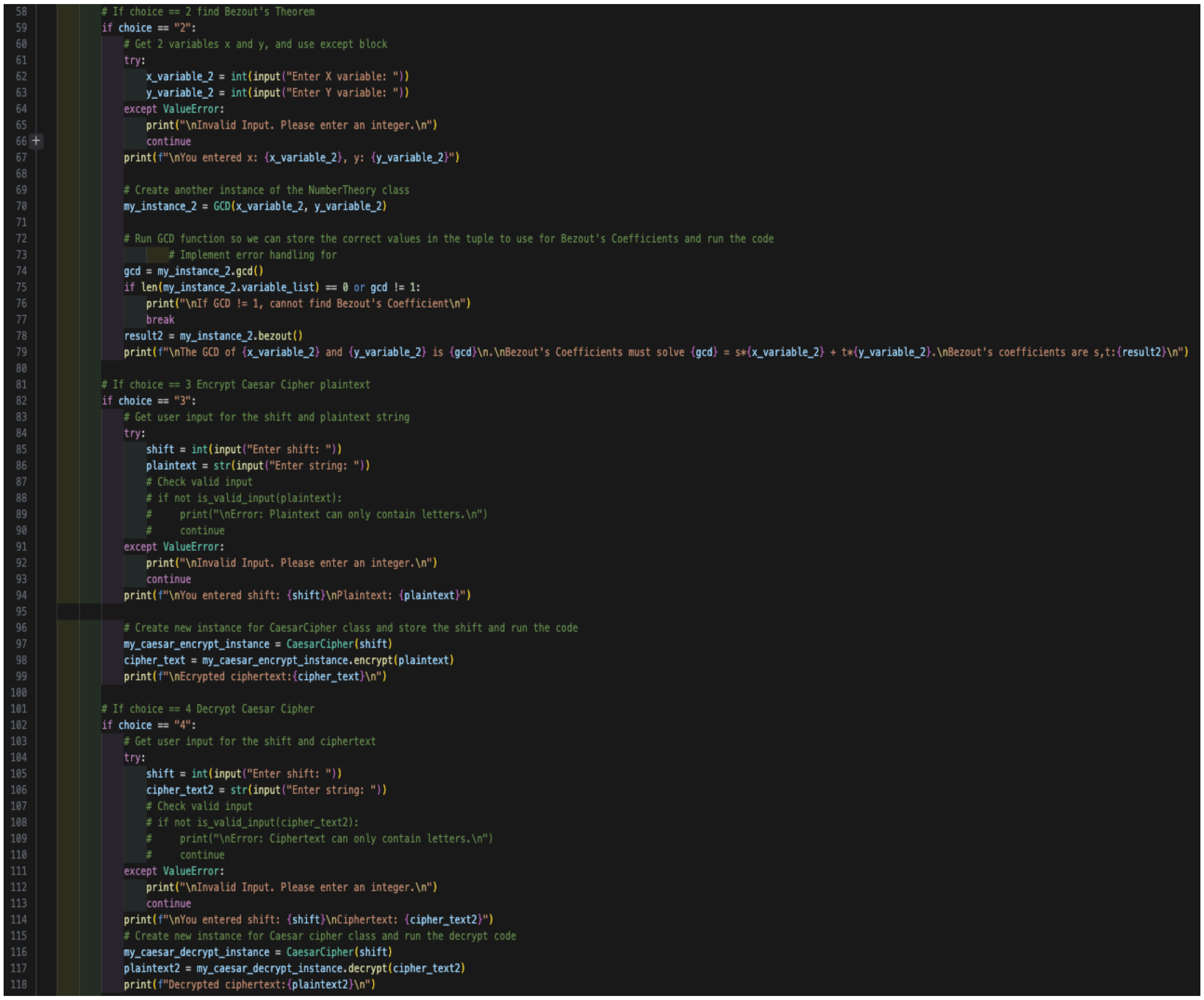
```python
class CaesarCipher:
    """ A class that encrypts and decrypts strings using the Caesar cipher """

    def __init__(self, shift):
        self._shift = int(shift)  # initialise the cipher with a shift

        # create a dictionary with Caesar cipher values for each letter
        self.char_values = {
            "A": 1, "B": 2, "C": 3, "D": 4,
            "E": 5, "F": 6, "G": 7, "H": 8,
            "I": 9, "J": 10, "K": 11, "L": 12,
            "M": 13, "N": 14, "O": 15, "P": 16,
            "Q": 17, "R": 18, "S": 19, "T": 20,
            "U": 21, "V": 22, "W": 23, "X": 24,
            "Y": 25, "Z": 26
        }

    def encrypt(self, s):
        """
        Changes alphabetical characters using a shift and leaves non-alphabet characters unchanges
        Int, String --> String
        """
        s = s.upper()  # convert the string into all upper case
        shift = self._shift

        # create an empty string for encrypted values to be placed
        encrypted_string = ""

        for char in s:  # loop through all characters in the string
            if char in self.char_values:  # check if each charcater in dict
                # assign numeric value from dict
                numeric_value = self.char_values[char]
                # find the shifted value
                shifted_value = (numeric_value + shift - 1) % 26 + 1

                # loop through dict keys & values
                for key, value in self.char_values.items():
                    # check for a match, add shifted value to encrypted string
                    if value == shifted_value:
                        encrypted_string += key
                        break  # end the loop after finding a match
            else:  # if char is not in alphabet, leave it unchanged
                encrypted_string += char
        return str(encrypted_string).replace(" ", "")

    def decrypt(self, s):
        """
        Returns alphabetical charcaters to their origional value, reversing the shift
        Int, String -> String
        """
        s = s.upper()  # convert the string into all upper case
        shift = self._shift

        # create an empty string for decrypted values to be placed
        decrypted_string = ""

        for char in s:  # loop through all characters in the string
            if char in self.char_values:  # check if each charcater in dict
                # assign numeric value from dict
                numeric_value = self.char_values[char]
                # find the un-shifted value
                decrypted_value = (numeric_value - shift) % 26

                # handle the case where the decrypted value is 0
                if decrypted_value == 0:
                    decrypted_value = 26

                # loop through dict keys & values
                for key, value in self.char_values.items():
                    # check for a match, add shifted value to encrypted string
                    if value == decrypted_value:
                        decrypted_string += key
                        break  # end the loop after finding a match
            else:  # if char is not in alphabet, leave it unchanged
                decrypted_string += char
        return decrypted_string.replace(" ", "")
```



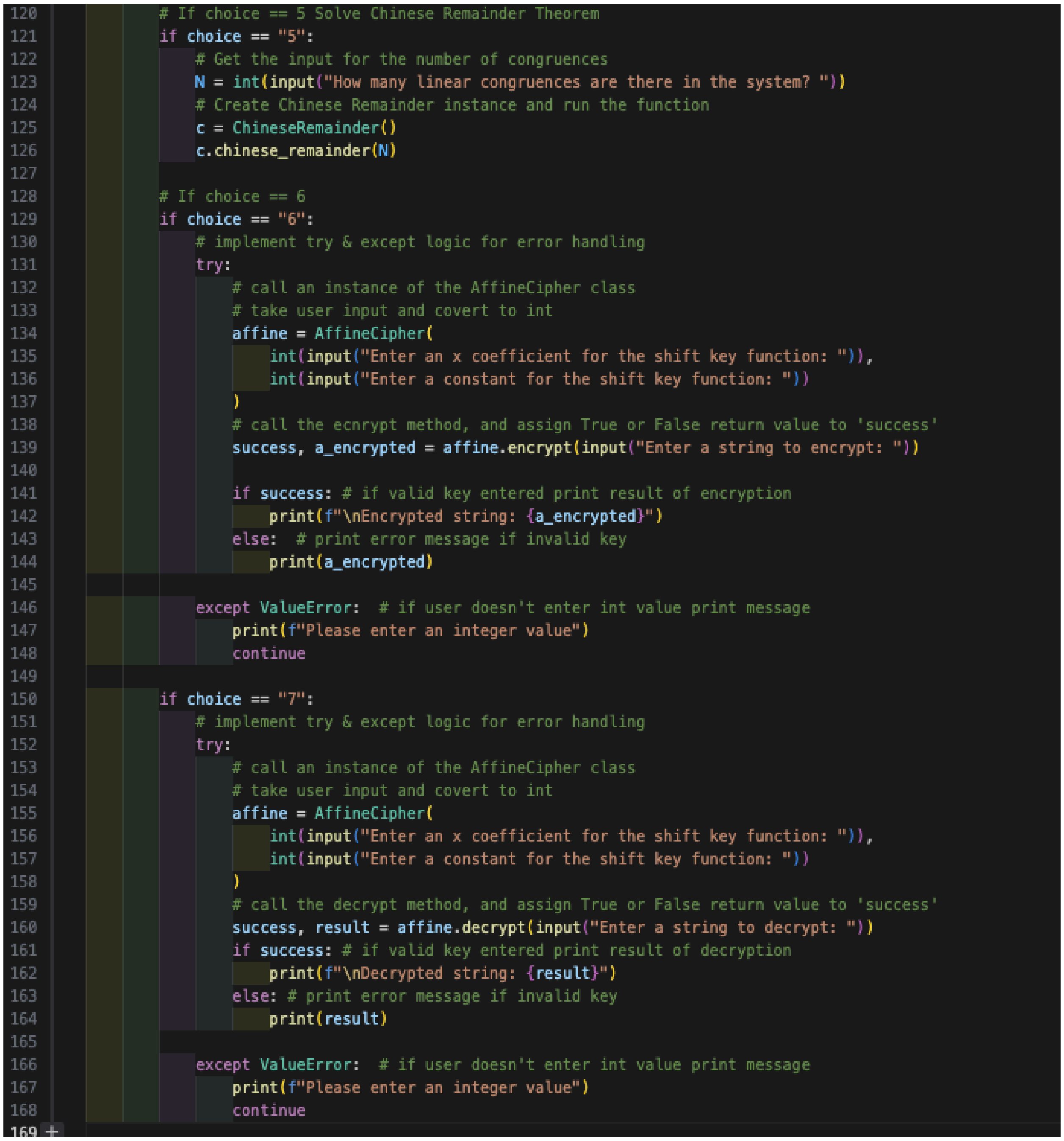
```python

from Caesar import CaesarCipher
from affine import AffineCipher

caesar = CaesarCipher(input("Enter a key for the cipher: "))

encrypted = caesar.encrypt(input("Enter a string to encrypt: "))
print(f"Encrypted string: {encrypted}")

decrypted = caesar.decrypt(encrypted)
print(f"Decrypted string: {decrypted}")

affine = AffineCipher(
    input("Enter an x coefficient for the shift key function: "),
    input("Enter a constant for the shift key function: ")
)

# call the ecnrypt method, and assign True or False return value to 'success'
success, a_encrypted = affine.encrypt(input("Enter a string to encrypt: "))
# print result of encryption
print(f"Encrypted string: {a_encrypted}")

# if encryption was successful initiate decryption
if success:
    success, a_decrypted = affine.decrypt(a_encrypted)
    # if decryption was successful print decrypted string
    if success:
        print(f"Decrypted string: {a_decrypted}")
```



```python
from Gcd import GCD
from Caesar import CaesarCipher
import string
from chinese_remainder import ChineseRemainder
from Caesar import CaesarCipher
from affine import AffineCipher

# Create a print menu for each Theorem
def print_menu():
    print("\nWhat would you like to do? \n")
    print("0 - nothing (terminate the program)")
    print("1 - Find greatest common divisor")
    print("2 - Find bezout's coefficients")
    print("3 - Caesar cipher encryption")
    print("4 - Caesar cipher decryption")
    print("5 - Solve Chinese Remainder Theorem")
    print("6 - Affine cipher encryption")
    print("7 - Affine cipher decryption")

# Create function to check valid input for Caesar Cipher
def is_valid_input(text):
    return all(char in string.ascii_letters or char == " " for char in text)

if __name__ == "__main__":
    # Create an infinite loop
    while True:
        print_menu()
        choice = input("\nEnter choice: ")

        # If choice == 0 terminate the program
        if choice == "0":
            print ("Terminating Program")
            break

        # If choice == 1 calculate GCD
        if choice == "1":
            # Get input for X and Y to find GCD
            try:
                x_variable = int(input("Enter X variable: "))
                y_variable = int(input("Enter Y variable: "))
            # Create error handling for non integers
            except ValueError:
                print("\nInvalid Input. Please enter an integer.\n")
                continue
            print(f"\nYou entered x: {x_variable}, y: {y_variable}")

            # Create an instance of the NumberTheory class
            my_instance = GCD(x_variable, y_variable)

            # Access instance attributes x and y
            x = my_instance.x
            y = my_instance.y

            # Call the gcd method using the instance
            result = my_instance.gcd()
            print(f"\nGreatest common divisor of {x} and {y} is {result}\n")

```



```
        # If choice == 2 find Bezout's Theorem
        if choice == "2":
            # Get 2 variables x and y, and use except block
            try:
                x_variable_2 = int(input("Enter X variable: "))
                y_variable_2 = int(input("Enter Y variable: "))
            except ValueError:
                print("\nInvalid Input. Please enter an integer.\n")
                continue
            print(f"\nYou entered x: {x_variable_2}, y: {y_variable_2}")

            # Create another instance of the NumberTheory class
            my_instance_2 = GCD(x_variable_2, y_variable_2)

            # Run GCD function so we can store the correct values in the tuple to use for Bezout's Coefficients and run the code
                    # Implement error handling for
            gcd = my_instance_2.gcd()
            if len(my_instance_2.variable_list) == 0 or gcd != 1:
                print("\nIf GCD != 1, cannot find Bezout's Coefficient\n")
                break
            result2 = my_instance_2.bezout()
            print(f"\nThe GCD of {x_variable_2} and {y_variable_2} is {gcd}\n.\nBezout's Coefficients must solve {gcd} = s*{x_variable_2} + t*{y_variable_2}.\nBezout's coefficients are s,t:{result2}\n")

        # If choice == 3 Encrypt Caesar Cipher plaintext
        if choice == "3":
            # Get user input for the shift and plaintext string
            try:
                shift = int(input("Enter shift: "))
                plaintext = str(input("Enter string: "))
                # Check valid input
                # if not is_valid_input(plaintext):
                #     print("\nError: Plaintext can only contain letters.\n")
                #     continue
            except ValueError:
                print("\nInvalid Input. Please enter an integer.\n")
                continue
            print(f"\nYou entered shift: {shift}\nPlaintext: {plaintext}")

            # Create new instance for CaesarCipher class and store the shift and run the code
            my_caesar_encrypt_instance = CaesarCipher(shift)
            cipher_text = my_caesar_encrypt_instance.encrypt(plaintext)
            print(f"\nEcrypted ciphertext:{cipher_text}\n")

        # If choice == 4 Decrypt Caesar Cipher
        if choice == "4":
            # Get user input for the shift and ciphertext
            try:
                shift = int(input("Enter shift: "))
                cipher_text2 = str(input("Enter string: "))
                # Check valid input
                # if not is_valid_input(cipher_text2):
                #     print("\nError: Ciphertext can only contain letters.\n")
                #     continue
            except ValueError:
                print("\nInvalid Input. Please enter an integer.\n")
                continue
            print(f"\nYou entered shift: {shift}\nCiphertext: {cipher_text2}")
            # Create new instance for Caesar cipher class and run the decrypt code
            my_caesar_decrypt_instance = CaesarCipher(shift)
            plaintext2 = my_caesar_decrypt_instance.decrypt(cipher_text2)
            print(f"Decrypted ciphertext:{plaintext2}\n")
```



```python
        # If choice == 5 Solve Chinese Remainder Theorem
        if choice == "5":
            # Get the input for the number of congruences
            N = int(input("How many linear congruences are there in the system? "))
            # Create Chinese Remainder instance and run the function
            c = ChineseRemainder()
            c.chinese_remainder(N)

        # If choice == 6
        if choice == "6":
            # implement try & except logic for error handling
            try:
                # call an instance of the AffineCipher class
                # take user input and covert to int
                affine = AffineCipher(
                    int(input("Enter an x coefficient for the shift key function: ")),
                    int(input("Enter a constant for the shift key function: "))
                )
                # call the ecnrypt method, and assign True or False return value to 'success'
                success, a_encrypted = affine.encrypt(input("Enter a string to encrypt: "))

                if success: # if valid key entered print result of encryption
                    print(f"\nEncrypted string: {a_encrypted}")
                else:  # print error message if invalid key
                    print(a_encrypted)

            except ValueError:  # if user doesn't enter int value print message
                print(f"Please enter an integer value")
                continue

        if choice == "7":
            # implement try & except logic for error handling
            try:
                # call an instance of the AffineCipher class
                # take user input and covert to int
                affine = AffineCipher(
                    int(input("Enter an x coefficient for the shift key function: ")),
                    int(input("Enter a constant for the shift key function: "))
                )
                # call the decrypt method, and assign True or False return value to 'success'
                success, result = affine.decrypt(input("Enter a string to decrypt: "))
                if success: # if valid key entered print result of decryption
                    print(f"\nDecrypted string: {result}")
                else: # print error message if invalid key
                    print(result)

            except ValueError:  # if user doesn't enter int value print message
                print(f"Please enter an integer value")
                continue
```