# Computer Networks Report – Assignment 1

**Name –** Aritra Dutta

**Roll –** 002010501054

**Class –** BCSE 3rd year

**Group –** A2

**Assignment – 1**

**DEADLINE: 7 th August, 2022**

**Problem Statement – Design and implement an error detection module.**

Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. Please note that you may need to use these schemes separately for other applications (assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame (decide the size of the frame) from the input. Based on the schemes, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases.

(a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given in next page).
(b) Error is detected by checksum but not by CRC.
(c) Error is detected by VRC but not by CRC.

**Date of Submission: 7th August, 2022**

## DESIGN

A bit, on travel, is subjected to electromagnetic (optical) interference due to noise signals (light sources). Thus, the data transmitted may be prone to errors. In particular, a bit '0' (bit '1') sent by the sender may be delivered as a bit '1' (bit '0') at the receiver. The receiver has to detect the error and ask the sender to transmit the data packet or must have the ability to detect and correct the errors.

In this assignment, we shall discuss the following error detection techniques in detail.

1. Vertical Redundancy check
2. Longitudinal Redundancy check
3. Checksum
4. Cyclic Redundancy check

## Implementation of Error Detection Module :

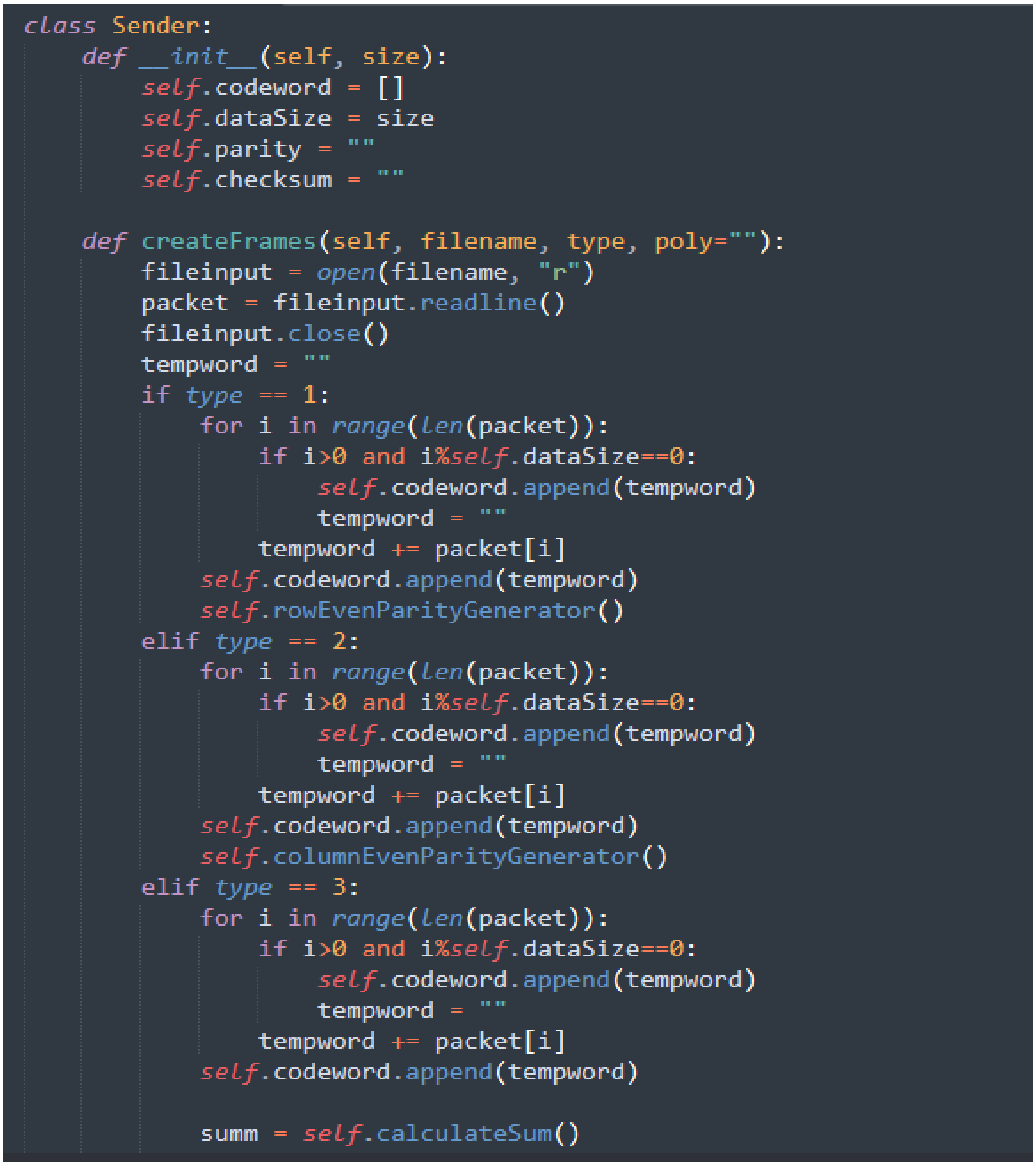
- **Sender.py** (Sender Module)
- **Receiver.py** (Receiver Module)
- **Run.py** (Interface to interact with Sender and Receiver)

## Details of the Above Files that achieve error detection for data frames

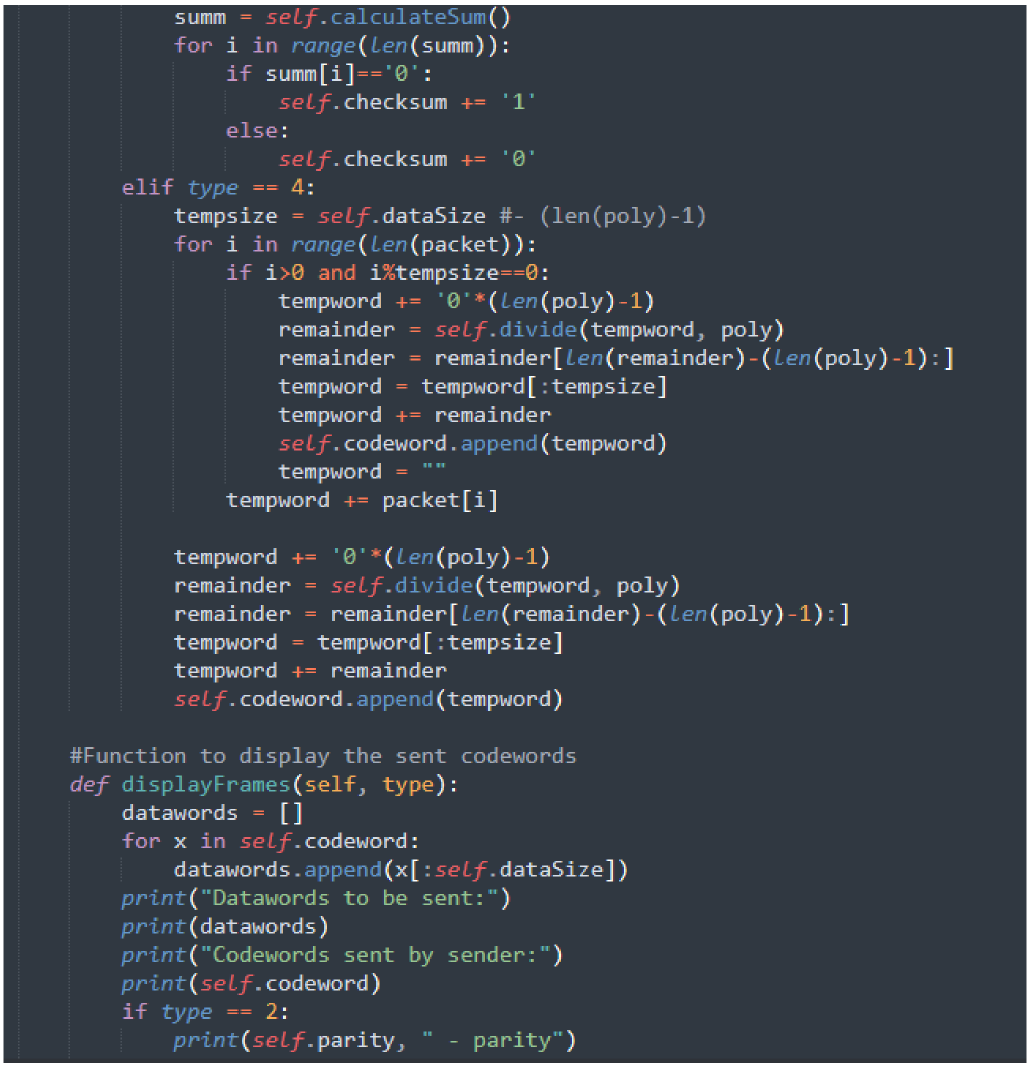
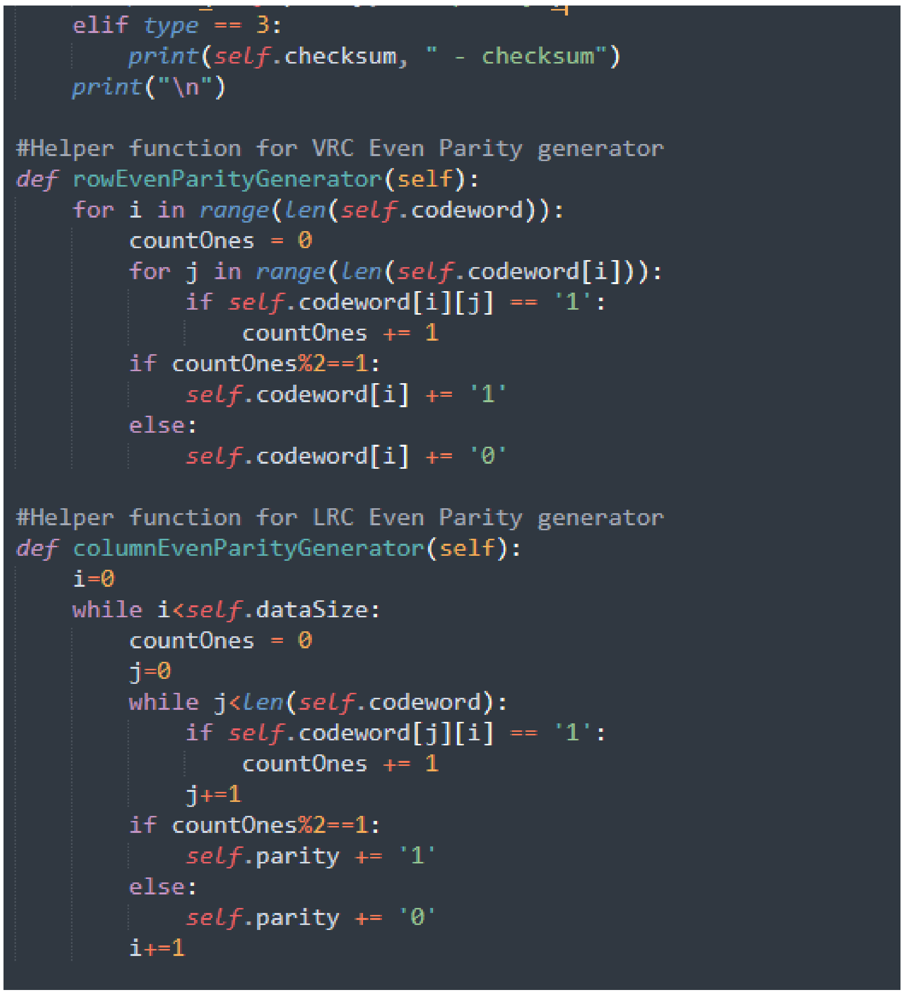
1. **Sender.py** :
    - The input file is read, which contains a sequence of 0 and 1.
    - The message sequence is divided into datawords on the basis of frame size taken as the input from the user.
    - According to the four schemes namely VRC, LRC, Checksum and CRC, redundant bits/dataword are added along with the datawords to form codewords.
    - The datawords and codewords which are to be sent are displayed.
    - These encoded codewords are then sent to the receiver.
2. **Receiver.py** :
    - The codewords are received from the sender.
    - The received codewords are then decoded according to the four schemes namely VRC, LRC, Checksum and CRC.
    - The result is checked and shown if there is any error detected.
    - The codewords and the datawords extracted from the codewords are also displayed.
3. **Run.py** :
    - This program acts as an interface to the above programs – sender.py and receiver.py
    - In this program, there are functions for injecting errors are included.
    - There are two separate functions for injecting errors on random positions of codeword, and for injecting errors on specific positions taken as input.
    - A filename is taken as input from command line while executing the program.
    - This is then used as the input file which has a sequence of 0 and 1 stored in it.
    - The function invocations of Sender class stored in sender.py and Receiver class stored in receiver.py are done in this file to send and receive the data.
    - The sent data are injected with errors using the above mentioned functions.
    - For all the following three cases, three different functions have been created to execute a specific case at a moment.
    - Error is detected by all four schemes.

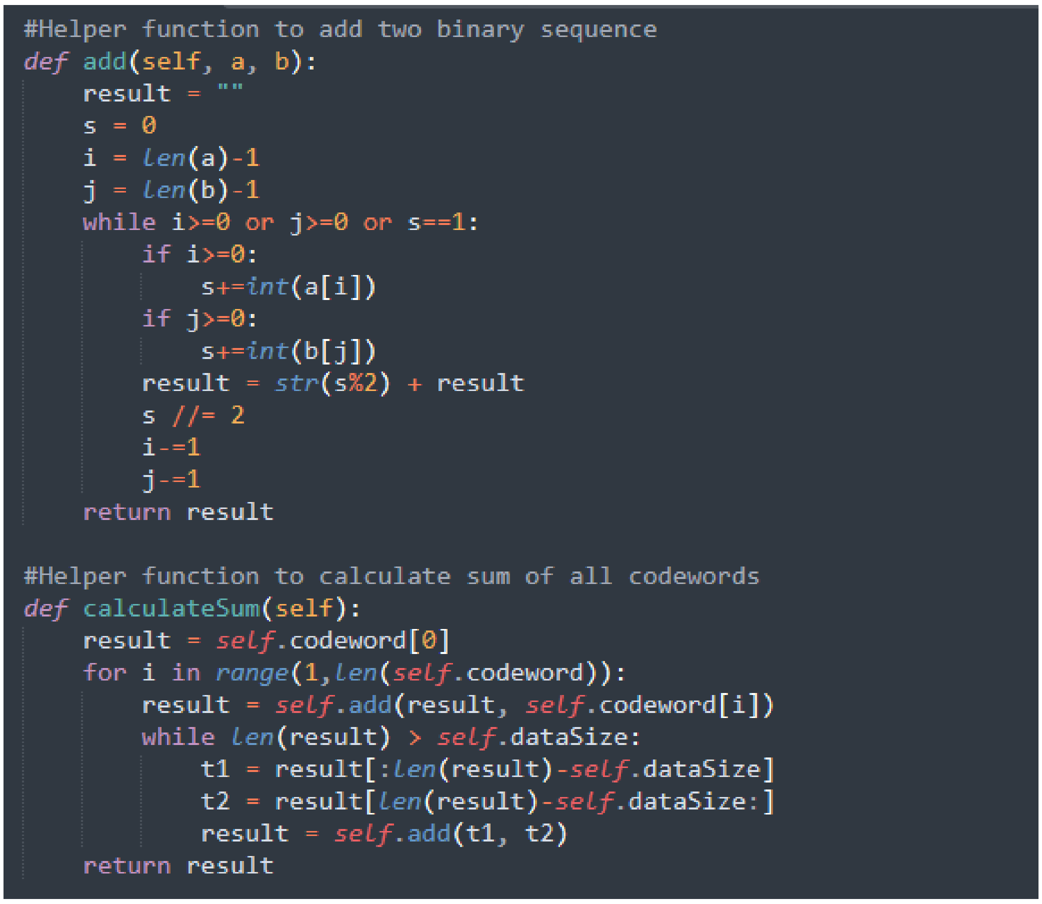
- Error is detected by checksum but not CRC.

**Code Snippet of sender.py:**

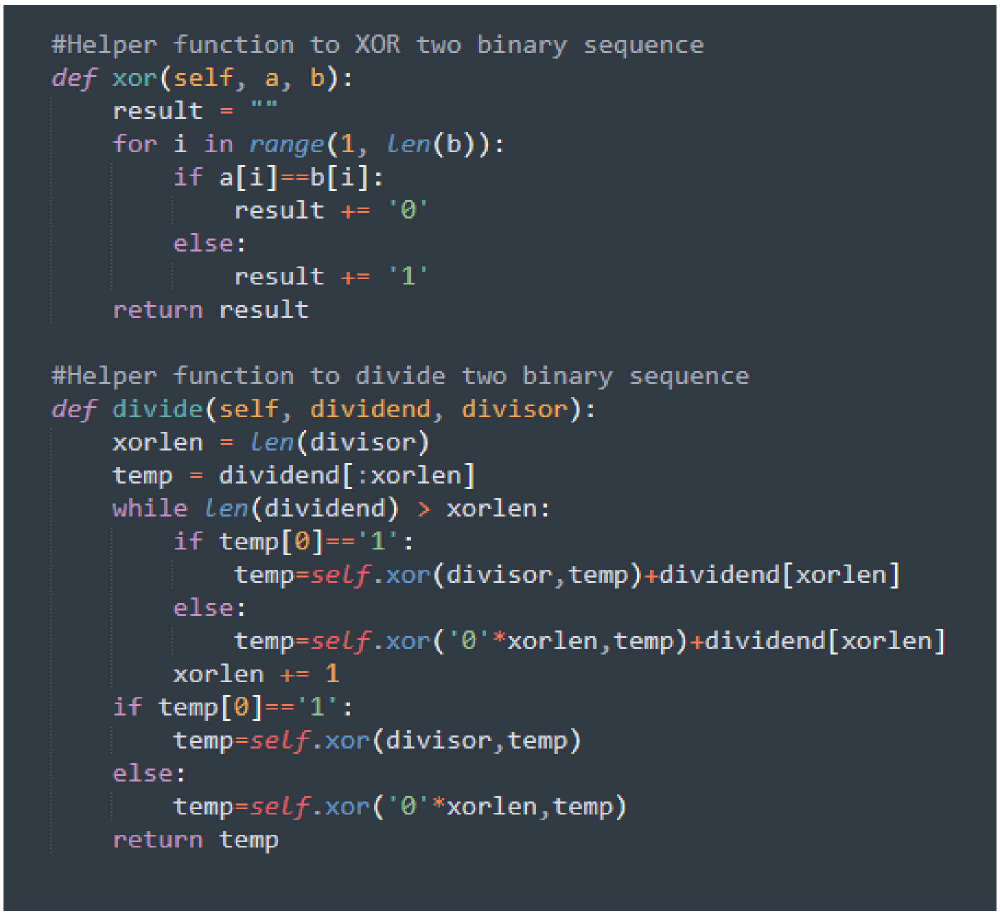


```python
class Sender:
    def __init__(self, size):
        self.codeword = []
        self.dataSize = size
        self.parity = ""
        self.checksum = ""

    def createFrames(self, filename, type, poly=""):
        fileinput = open(filename, "r")
        packet = fileinput.readline()
        fileinput.close()
        tempword = ""
        if type == 1:
            for i in range(len(packet)):
                if i>0 and i%self.dataSize==0:
                    self.codeword.append(tempword)
                    tempword = ""
                tempword += packet[i]
            self.codeword.append(tempword)
            self.rowEvenParityGenerator()
        elif type == 2:
            for i in range(len(packet)):
                if i>0 and i%self.dataSize==0:
                    self.codeword.append(tempword)
                    tempword = ""
                tempword += packet[i]
            self.codeword.append(tempword)
            self.columnEvenParityGenerator()
        elif type == 3:
            for i in range(len(packet)):
                if i>0 and i%self.dataSize==0:
                    self.codeword.append(tempword)
                    tempword = ""
                tempword += packet[i]
            self.codeword.append(tempword)

            summ = self.calculateSum()
```


```
            summ = self.calculateSum()
            for i in range(len(summ)):
                if summ[i]=='0':
                    self.checksum += '1'
                else:
                    self.checksum += '0'
        elif type == 4:
            tempsize = self.dataSize #- (len(poly)-1)
            for i in range(len(packet)):
                if i>0 and i%tempsize==0:
                    tempword += '0'*(len(poly)-1)
                    remainder = self.divide(tempword, poly)
                    remainder = remainder[len(remainder)-(len(poly)-1):]
                    tempword = tempword[:tempsize]
                    tempword += remainder
                    self.codeword.append(tempword)
                    tempword = ""
                tempword += packet[i]

            tempword += '0'*(len(poly)-1)
            remainder = self.divide(tempword, poly)
            remainder = remainder[len(remainder)-(len(poly)-1):]
            tempword = tempword[:tempsize]
            tempword += remainder
            self.codeword.append(tempword)

    #Function to display the sent codewords
    def displayFrames(self, type):
        datawords = []
        for x in self.codeword:
            datawords.append(x[:self.dataSize])
        print("Datawords to be sent:")
        print(datawords)
        print("Codewords sent by sender:")
        print(self.codeword)
        if type == 2:
            print(self.parity, " - parity")
```

```
        elif type == 3:
            print(self.checksum, " - checksum")
        print("\n")

    #Helper function for VRC Even Parity generator
    def rowEvenParityGenerator(self):
        for i in range(len(self.codeword)):
            countOnes = 0
            for j in range(len(self.codeword[i])):
                if self.codeword[i][j] == '1':
                    countOnes += 1
            if countOnes%2==1:
                self.codeword[i] += '1'
            else:
                self.codeword[i] += '0'

    #Helper function for LRC Even Parity generator
    def columnEvenParityGenerator(self):
        i=0
        while i<self.dataSize:
            countOnes = 0
            j=0
            while j<len(self.codeword):
                if self.codeword[j][i] == '1':
                    countOnes += 1
                j+=1
            if countOnes%2==1:
                self.parity += '1'
            else:
                self.parity += '0'
            i+=1
```



```
#Helper function to add two binary sequence
def add(self, a, b):
    result = ""
    s = 0
    i = len(a)-1
    j = len(b)-1
    while i>=0 or j>=0 or s==1:
        if i>=0:
            s+=int(a[i])
        if j>=0:
            s+=int(b[j])
        result = str(s%2) + result
        s //= 2
        i-=1
        j-=1
    return result

#Helper function to calculate sum of all codewords
def calculateSum(self):
    result = self.codeword[0]
    for i in range(1,len(self.codeword)):
        result = self.add(result, self.codeword[i])
        while len(result) > self.dataSize:
            t1 = result[:len(result)-self.dataSize]
            t2 = result[len(result)-self.dataSize:]
            result = self.add(t1, t2)
    return result
```



```
#Helper function to XOR two binary sequence
def xor(self, a, b):
    result = ""
    for i in range(1, len(b)):
        if a[i]==b[i]:
            result += '0'
        else:
            result += '1'
    return result

#Helper function to divide two binary sequence
def divide(self, dividend, divisor):
    xorlen = len(divisor)
    temp = dividend[:xorlen]
    while len(dividend) > xorlen:
        if temp[0]=='1':
            temp=self.xor(divisor,temp)+dividend[xorlen]
        else:
            temp=self.xor('0'*xorlen,temp)+dividend[xorlen]
        xorlen += 1
    if temp[0]=='1':
        temp=self.xor(divisor,temp)
    else:
        temp=self.xor('0'*xorlen,temp)
    return temp
```

## Method descriptions of sender.py:

- **createFrames(self, filename, type, poly=””)** : This method is used to take input of sequence of 0,1 from a given file, accessing it using the filename passed as argument. Then the packet is divided into datawords of fixed size. According to the type (scheme) of error detection, namely VRC, LRC, Checksum, CRC, the datawords are appended with appropriate redundant bits

- **displayframes(self,type)** : This method has been used for displaying the datawords and codewords which are to be sent.

- **rowEvenParityGenerator(self)** : This method has been used as a helper function for VRC. In this method, for each dataword, a parity bit is generated such that the parity of the word is even. This parity bit is then appended to dataword to form a codeword

- **columnEvenParityGenerator(self)** : This method has been used as a helper function for LRC. In this method, for every column of all dataword, corresponding parity bit is generator such that the parity becomes even in that column. As a result, a parity sequence is appended along with codewords

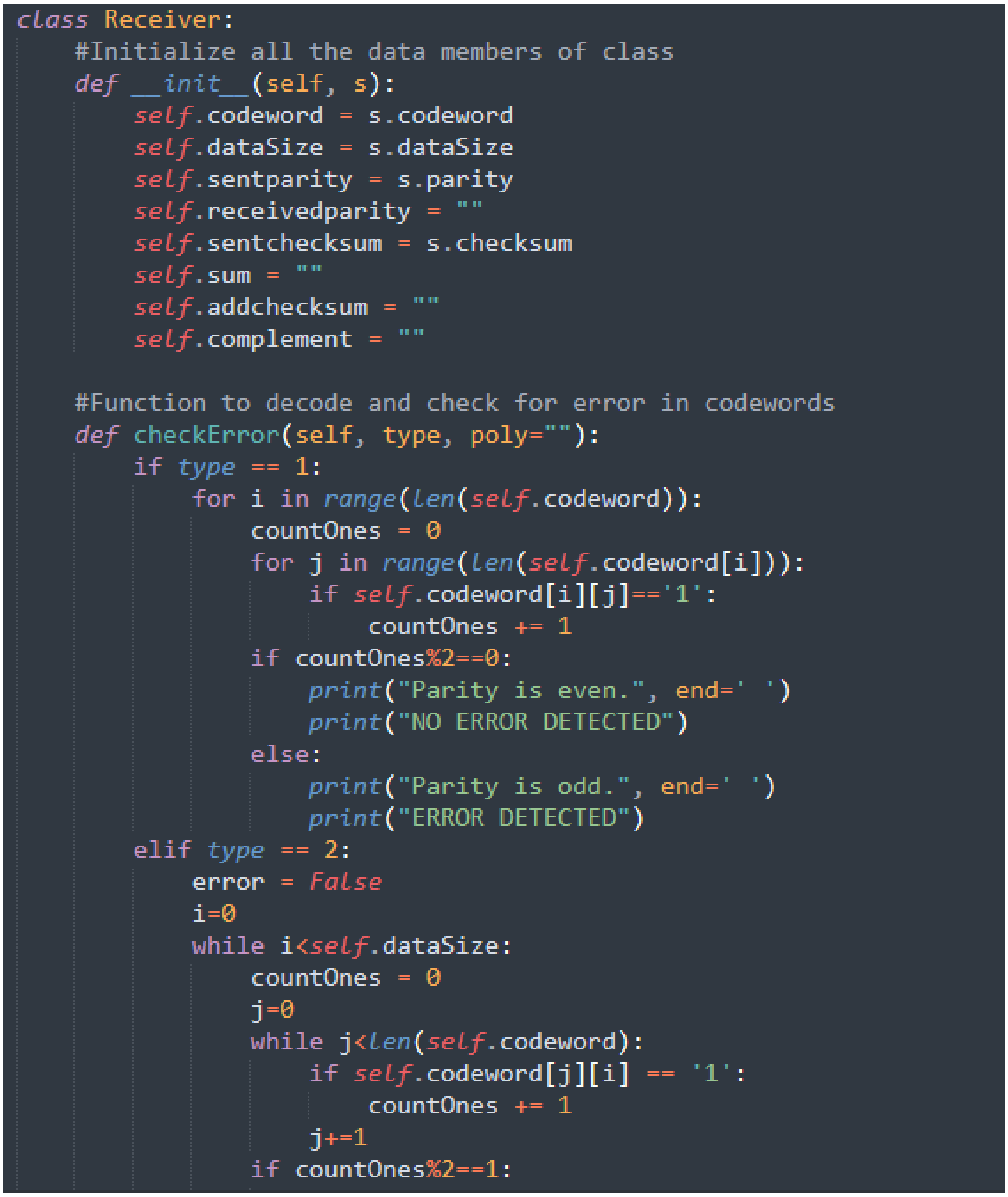
- **add(self,a,b)** : This method has been used as a helper function for implementing Checksum. It takes two binary sequence as parameters, and returns the result of wrapped addition of them

- **calculateSum(self)** : This method has been used as a helper function for implementing Checksum. It calculates the sum of all the datawords (using the above mentioned function) and returns the result

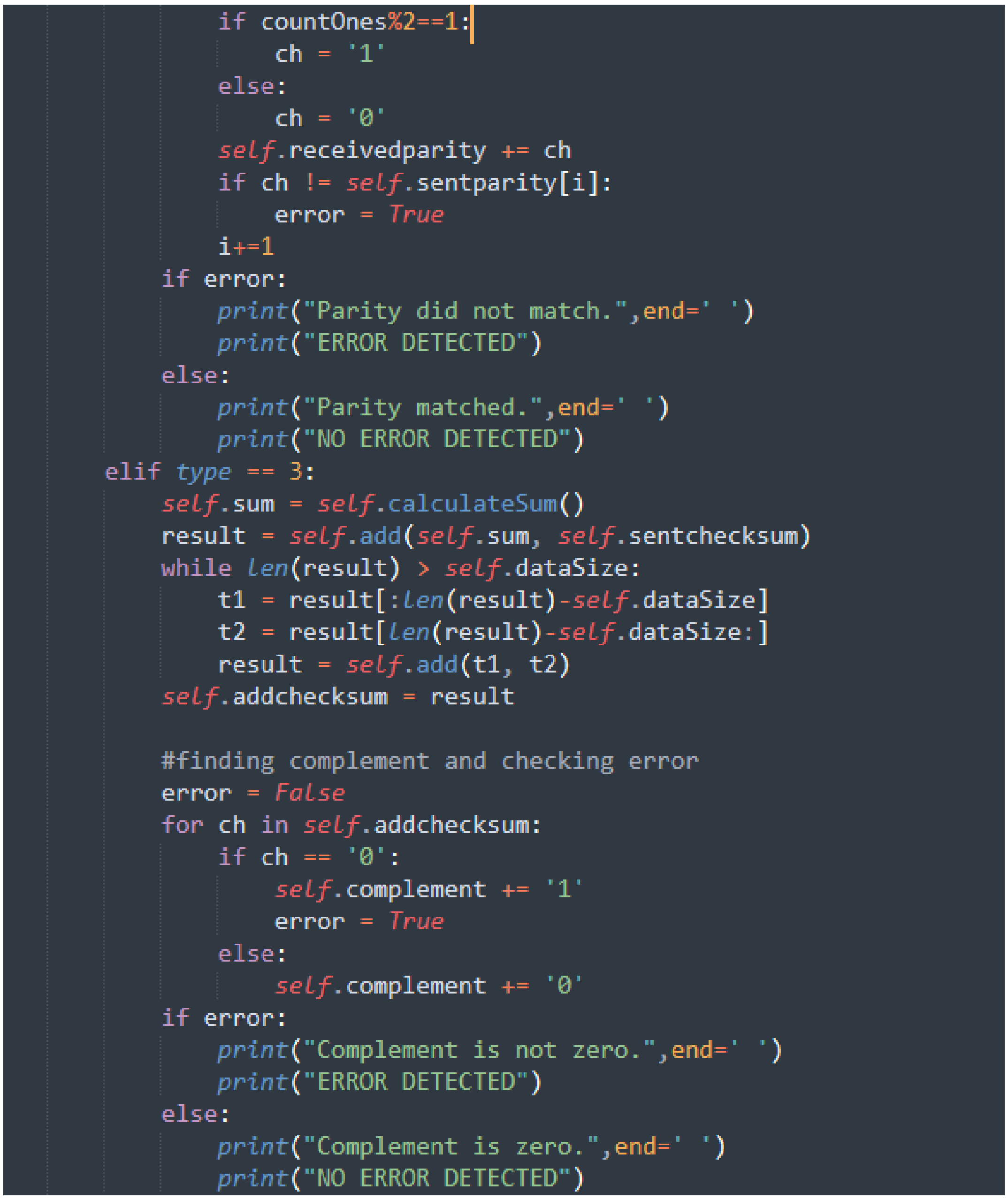
- **xor(self,a,b)** : This method has been used as a helper function for implementing CRC. In this method, two binary sequence is taken as input from its arguments, and return the xor of them.

- **divide(self,dividend,divisor) :** This method has been used as a helper function for implementing CRC. In this method, dividend and divisor is taken as input from its arguments. The division is performed using mod 2 arithmetic (exclusice-OR) on the message using divisor polynomial, and the remainder is returned

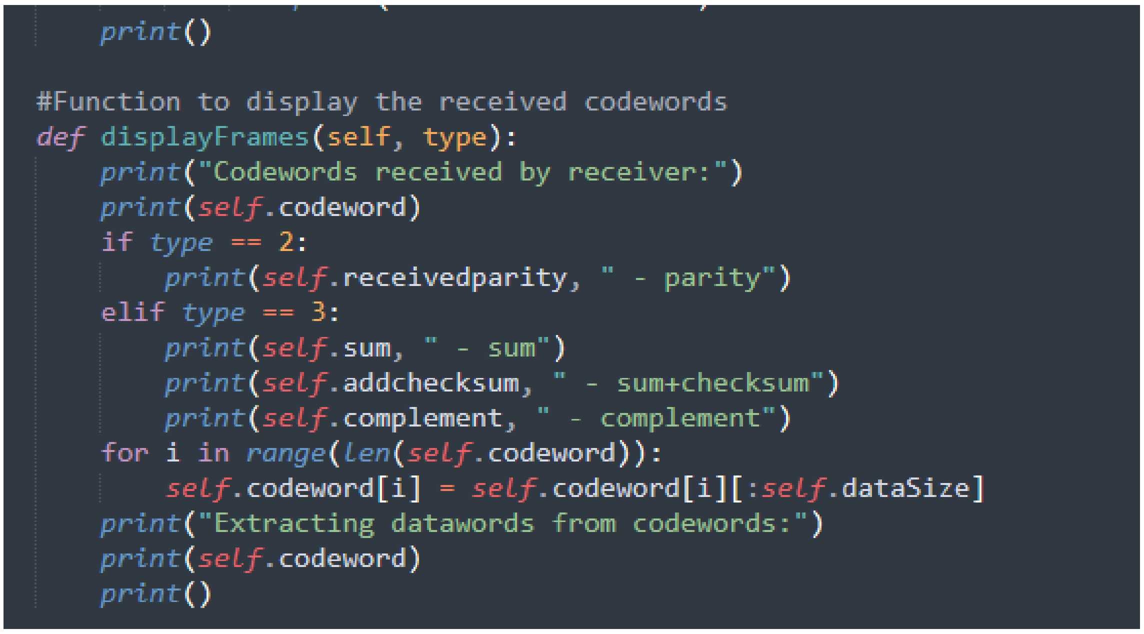
## Code Snippet of receiver.py:



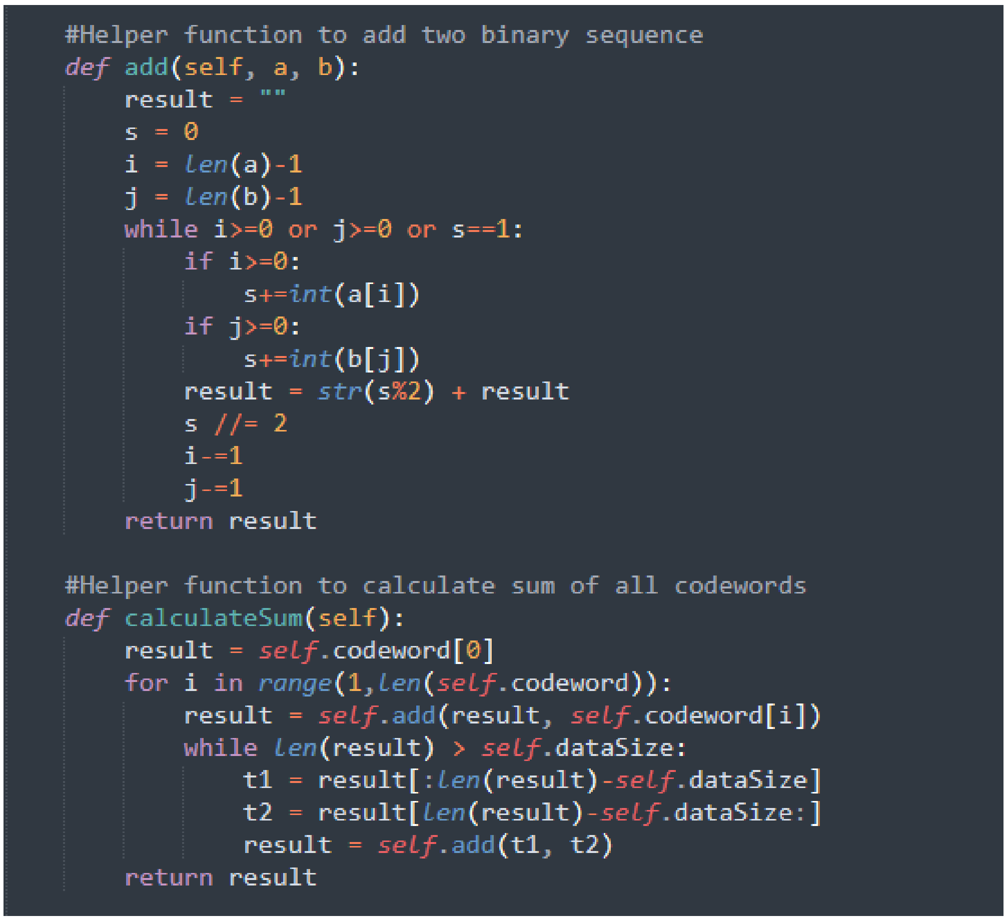
```python
class Receiver:
    #Initialize all the data members of class
    def __init__(self, s):
        self.codeword = s.codeword
        self.dataSize = s.dataSize
        self.sentparity = s.parity
        self.receivedparity = ""
        self.sentchecksum = s.checksum
        self.sum = ""
        self.addchecksum = ""
        self.complement = ""

    #Function to decode and check for error in codewords
    def checkError(self, type, poly=""):
        if type == 1:
            for i in range(len(self.codeword)):
                countOnes = 0
                for j in range(len(self.codeword[i])):
                    if self.codeword[i][j]=='1':
                        countOnes += 1
                if countOnes%2==0:
                    print("Parity is even.", end=' ')
                    print("NO ERROR DETECTED")
                else:
                    print("Parity is odd.", end=' ')
                    print("ERROR DETECTED")
        elif type == 2:
            error = False
            i=0
            while i<self.dataSize:
                countOnes = 0
                j=0
                while j<len(self.codeword):
                    if self.codeword[j][i] == '1':
                        countOnes += 1
                    j+=1
                if countOnes%2==1:
```



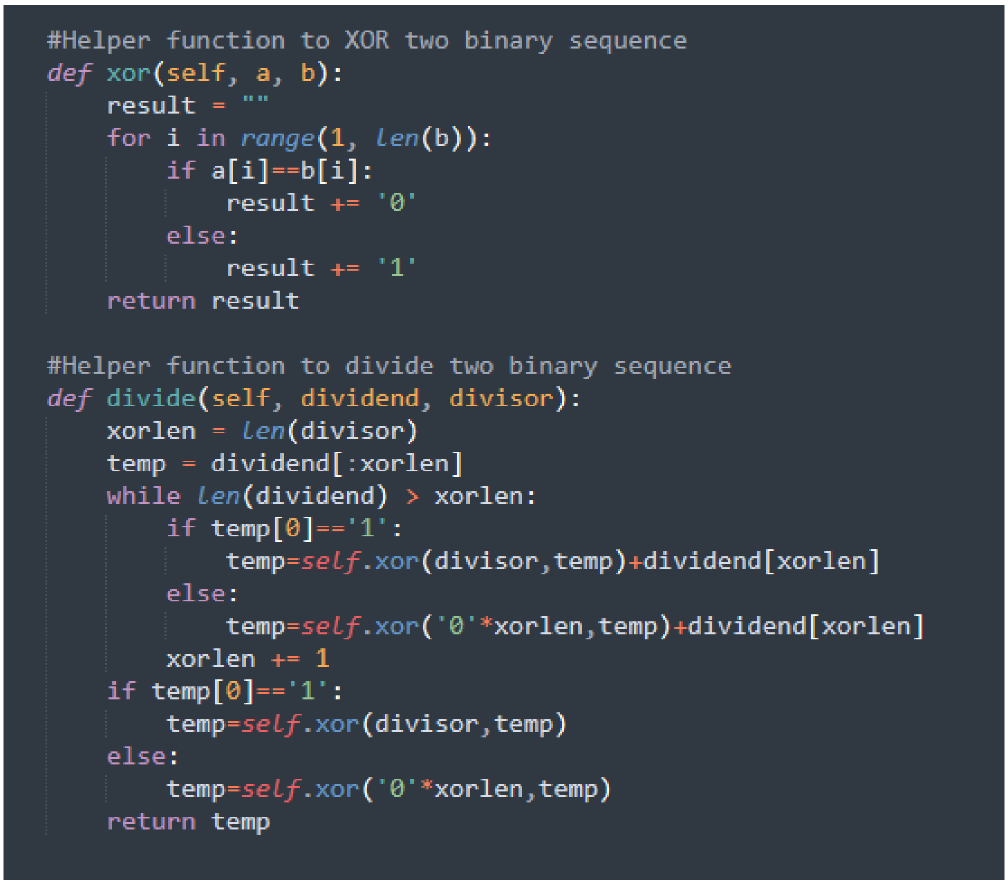
```python
                    if countOnes%2==1:
                        ch = '1'
                    else:
                        ch = '0'
                    self.receivedparity += ch
                    if ch != self.sentparity[i]:
                        error = True
                    i+=1
                if error:
                    print("Parity did not match.",end=' ')
                    print("ERROR DETECTED")
                else:
                    print("Parity matched.",end=' ')
                    print("NO ERROR DETECTED")
            elif type == 3:
                self.sum = self.calculateSum()
                result = self.add(self.sum, self.sentchecksum)
                while len(result) > self.dataSize:
                    t1 = result[:len(result)-self.dataSize]
                    t2 = result[len(result)-self.dataSize:]
                    result = self.add(t1, t2)
                self.addchecksum = result

                #finding complement and checking error
                error = False
                for ch in self.addchecksum:
                    if ch == '0':
                        self.complement += '1'
                        error = True
                    else:
                        self.complement += '0'
                if error:
                    print("Complement is not zero.",end=' ')
                    print("ERROR DETECTED")
                else:
                    print("Complement is zero.",end=' ')
                    print("NO ERROR DETECTED")
```



```python
        print()

    #Function to display the received codewords
    def displayFrames(self, type):
        print("Codewords received by receiver:")
        print(self.codeword)
        if type == 2:
            print(self.receivedparity, " - parity")
        elif type == 3:
            print(self.sum, " - sum")
            print(self.addchecksum, " - sum+checksum")
            print(self.complement, " - complement")
        for i in range(len(self.codeword)):
            self.codeword[i] = self.codeword[i][:self.dataSize]
        print("Extracting datawords from codewords:")
        print(self.codeword)
        print()
```



```python
#Helper function to add two binary sequence
def add(self, a, b):
    result = ""
    s = 0
    i = len(a)-1
    j = len(b)-1
    while i>=0 or j>=0 or s==1:
        if i>=0:
            s+=int(a[i])
        if j>=0:
            s+=int(b[j])
        result = str(s%2) + result
        s //= 2
        i-=1
        j-=1
    return result

#Helper function to calculate sum of all codewords
def calculateSum(self):
    result = self.codeword[0]
    for i in range(1,len(self.codeword)):
        result = self.add(result, self.codeword[i])
        while len(result) > self.dataSize:
            t1 = result[:len(result)-self.dataSize]
            t2 = result[len(result)-self.dataSize:]
            result = self.add(t1, t2)
    return result
```

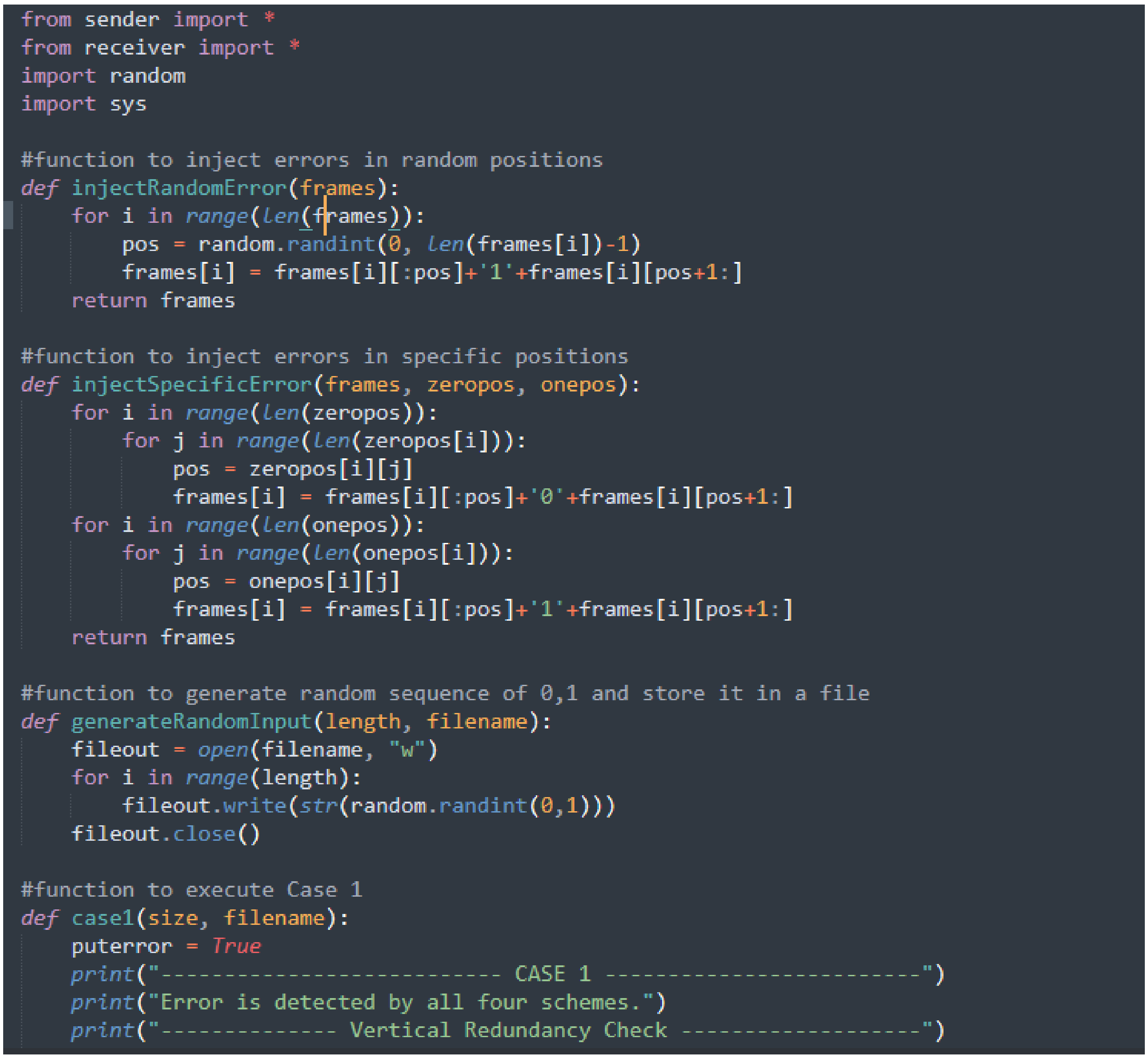


```python
#Helper function to XOR two binary sequence
def xor(self, a, b):
    result = ""
    for i in range(1, len(b)):
        if a[i]==b[i]:
            result += '0'
        else:
            result += '1'
    return result

#Helper function to divide two binary sequence
def divide(self, dividend, divisor):
    xorlen = len(divisor)
    temp = dividend[:xorlen]
    while len(dividend) > xorlen:
        if temp[0]=='1':
            temp=self.xor(divisor,temp)+dividend[xorlen]
        else:
            temp=self.xor('0'*xorlen,temp)+dividend[xorlen]
        xorlen += 1
    if temp[0]=='1':
        temp=self.xor(divisor,temp)
    else:
        temp=self.xor('0'*xorlen,temp)
    return temp
```

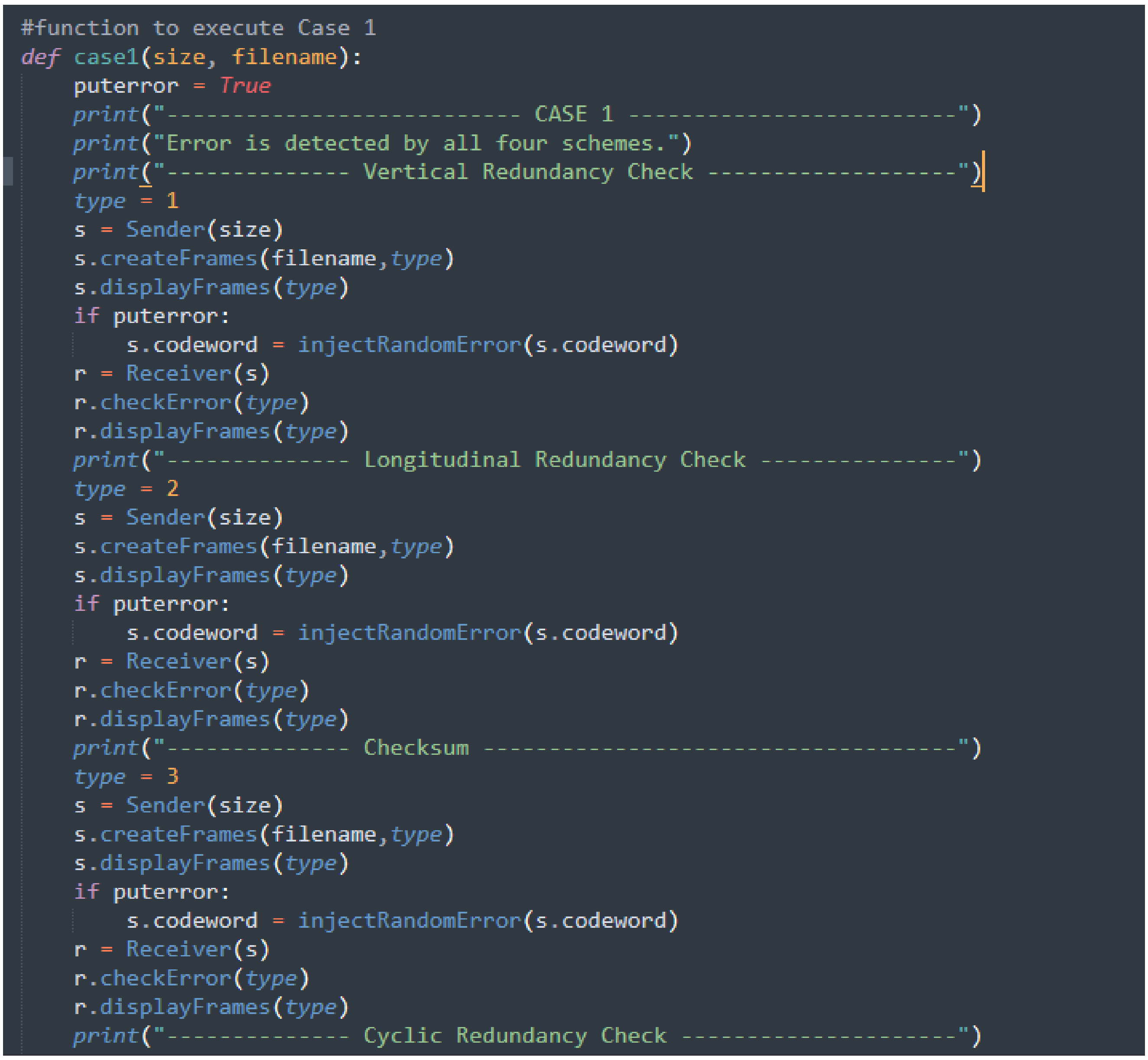
## Method descriptions of receiver.py:

- **checkError(self,type,poly=””)** : This method takes type (scheme) of error detection method as one of its argument. On the basis of the scheme the codewords received is checked for error. If there is an error, appropriate message is displayed on the screen
- **displayFrames(self,type**) : This method has been used to display the received codewords and datawords from the sender
- **add(self,a,b)** : This method has been used as a helper function for implementing Checksum. It takes two binary sequence as parameters, and returns the result of wrapped addition of them.
- **calculateSum(self)** : This method has been used as a helper function for implementing Checksum. It calculates the sum of all the datawords (using the above mentioned function) and returns the result.
- **xor(self,a,b)** : This method has been used as a helper function for implementing CRC. In this method, two binary sequence is taken as input from its arguments, and return the xor of them.
- **divide(self,dividend,divisor**) : This method has been used as a helper function for implementing CRC. In this method, dividend and divisor is taken as input from its arguments. The division is performed using mod 2 arithmetic (exclusice-OR) on the message using divisor polynomial, and the remainder is returned

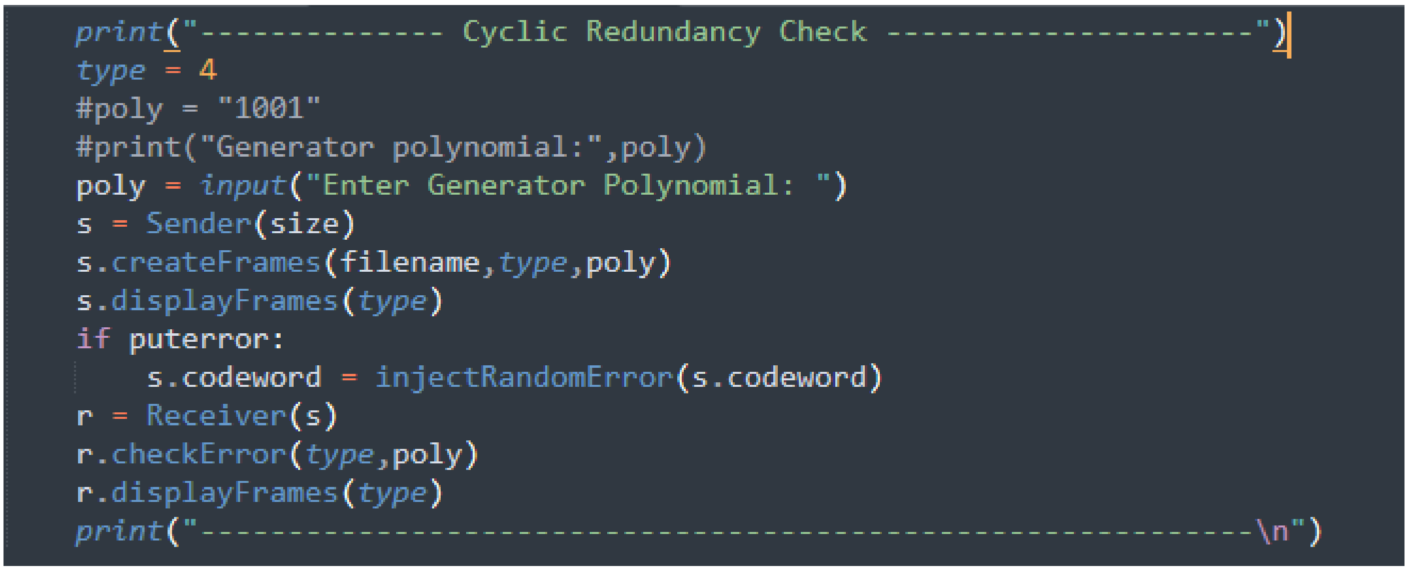
## Code Snippet of run.py:



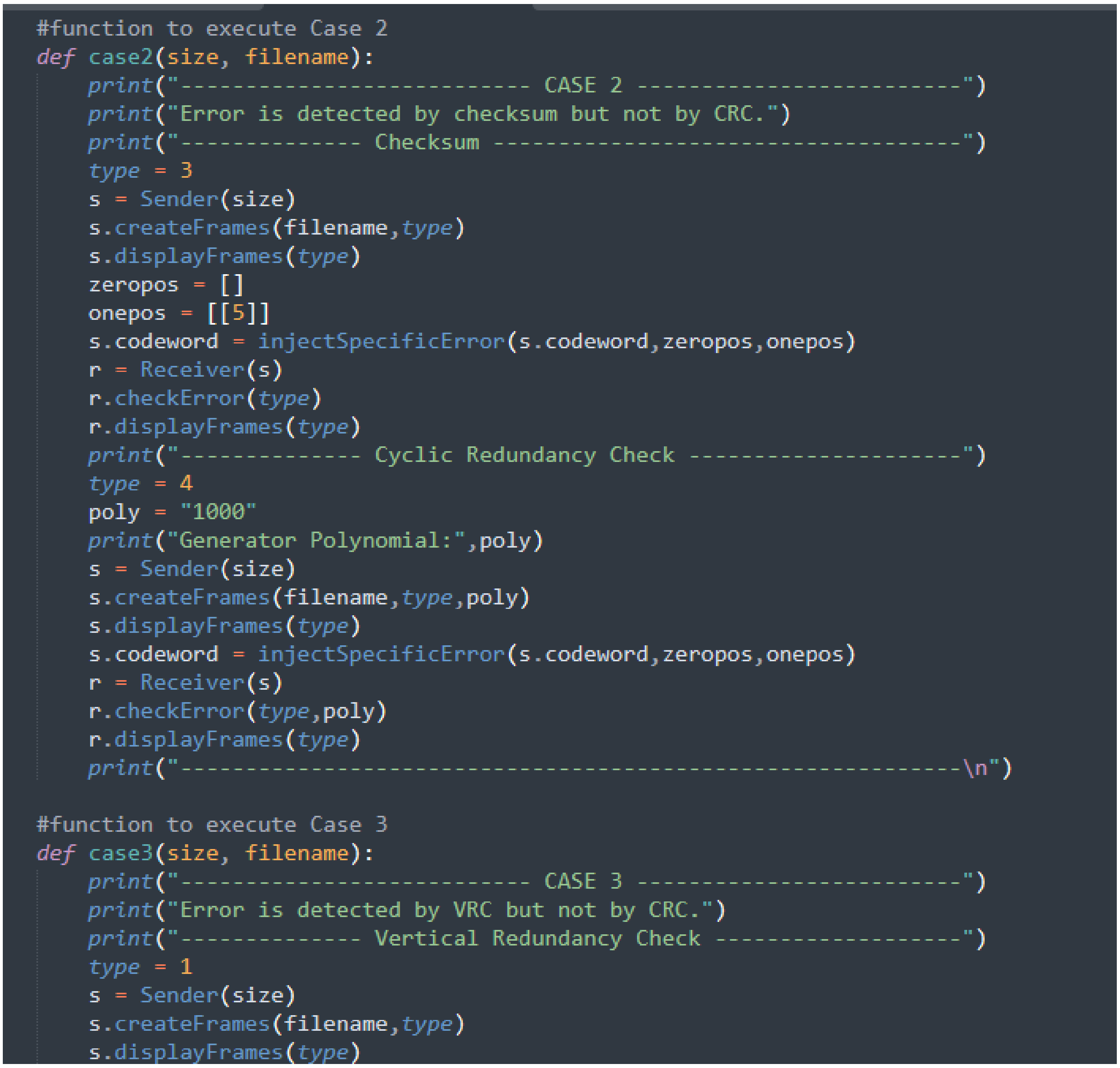
```
from sender import *
from receiver import *
import random
import sys

#function to inject errors in random positions
def injectRandomError(frames):
    for i in range(len(frames)):
        pos = random.randint(0, len(frames[i])-1)
        frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames

#function to inject errors in specific positions
def injectSpecificError(frames, zeropos, onepos):
    for i in range(len(zeropos)):
        for j in range(len(zeropos[i])):
            pos = zeropos[i][j]
            frames[i] = frames[i][:pos]+'0'+frames[i][pos+1:]
    for i in range(len(onepos)):
        for j in range(len(onepos[i])):
            pos = onepos[i][j]
            frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames

#function to generate random sequence of 0,1 and store it in a file
def generateRandomInput(length, filename):
    fileout = open(filename, "w")
    for i in range(length):
        fileout.write(str(random.randint(0,1)))
    fileout.close()

#function to execute Case 1
def case1(size, filename):
    puterror = True
    print("--------------------------- CASE 1 -------------------------")
    print("Error is detected by all four schemes.")
    print("-------------- Vertical Redundancy Check -------------------")
```



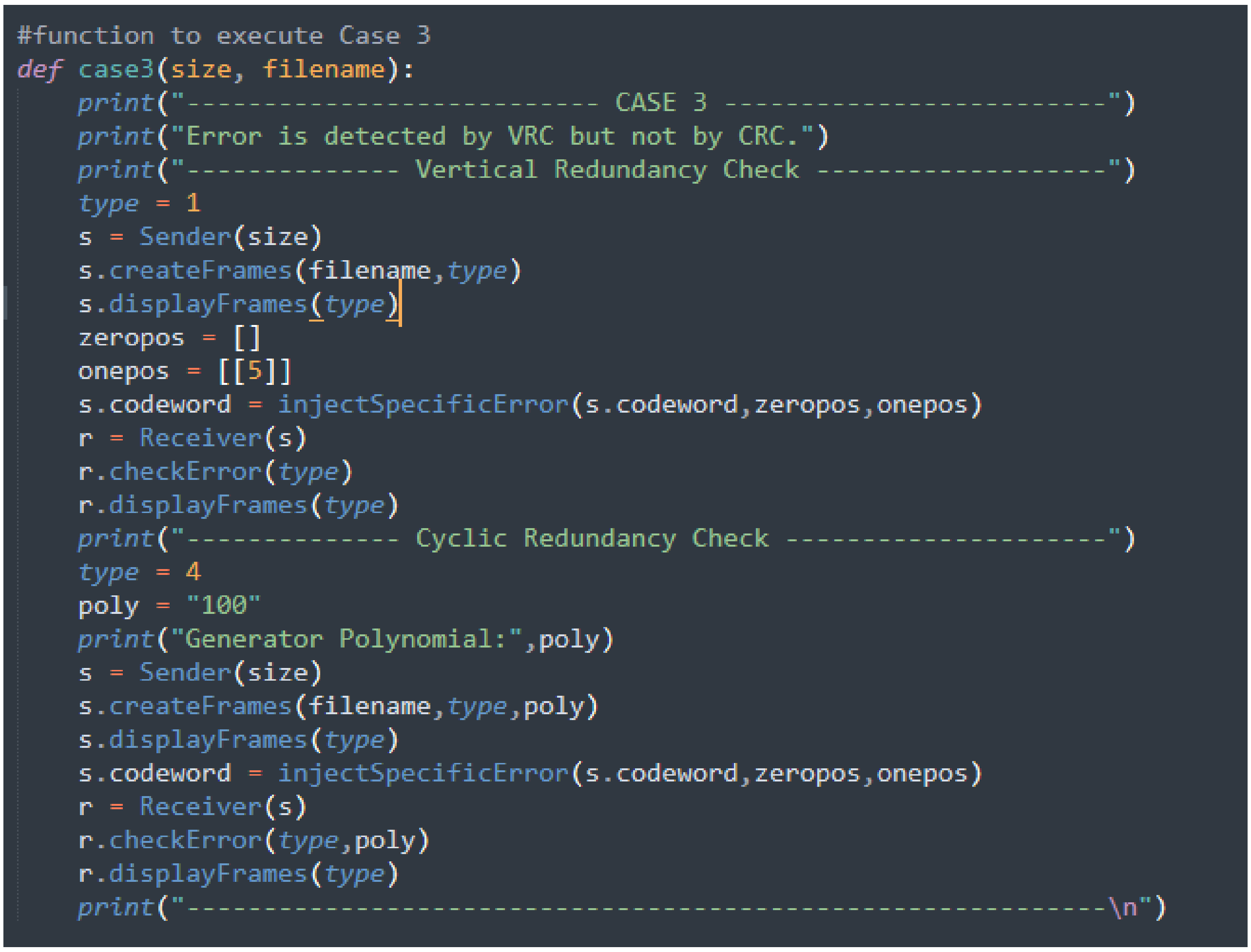
```python
#function to execute Case 1
def case1(size, filename):
    puterror = True
    print("--------------------------- CASE 1 -------------------------")
    print("Error is detected by all four schemes.")
    print("-------------- Vertical Redundancy Check --------------------")
    type = 1
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
    if puterror:
        s.codeword = injectRandomError(s.codeword)
    r = Receiver(s)
    r.checkError(type)
    r.displayFrames(type)
    print("-------------- Longitudinal Redundancy Check ---------------")
    type = 2
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
    if puterror:
        s.codeword = injectRandomError(s.codeword)
    r = Receiver(s)
    r.checkError(type)
    r.displayFrames(type)
    print("-------------- Checksum ------------------------------------")
    type = 3
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
    if puterror:
        s.codeword = injectRandomError(s.codeword)
    r = Receiver(s)
    r.checkError(type)
    r.displayFrames(type)
    print("-------------- Cyclic Redundancy Check ---------------------")
```



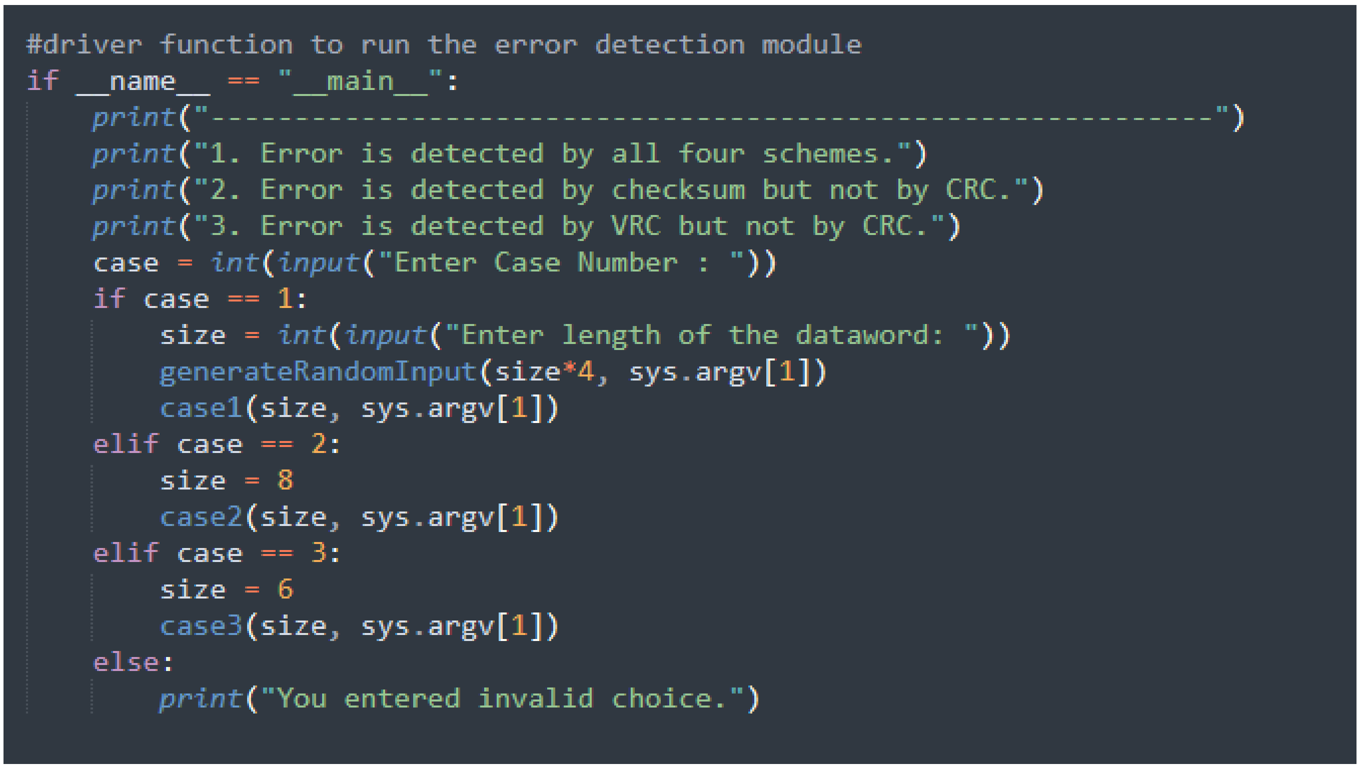
```python
    print("-------------- Cyclic Redundancy Check ---------------------")
    type = 4
    #poly = "1001"
    #print("Generator polynomial:",poly)
    poly = input("Enter Generator Polynomial: ")
    s = Sender(size)
    s.createFrames(filename,type,poly)
    s.displayFrames(type)
    if puterror:
        s.codeword = injectRandomError(s.codeword)
    r = Receiver(s)
    r.checkError(type,poly)
    r.displayFrames(type)
    print("-----------------------------------------------------------\n")
```



```python
#function to execute Case 2
def case2(size, filename):
    print("--------------------------- CASE 2 -------------------------")
    print("Error is detected by checksum but not by CRC.")
    print("-------------- Checksum ------------------------------------")
    type = 3
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
    zeropos = []
    onepos = [[5]]
    s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
    r = Receiver(s)
    r.checkError(type)
    r.displayFrames(type)
    print("-------------- Cyclic Redundancy Check ---------------------")
    type = 4
    poly = "1000"
    print("Generator Polynomial:",poly)
    s = Sender(size)
    s.createFrames(filename,type,poly)
    s.displayFrames(type)
    s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
    r = Receiver(s)
    r.checkError(type,poly)
    r.displayFrames(type)
    print("------------------------------------------------------------\n")

#function to execute Case 3
def case3(size, filename):
    print("--------------------------- CASE 3 -------------------------")
    print("Error is detected by VRC but not by CRC.")
    print("-------------- Vertical Redundancy Check -------------------")
    type = 1
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
```



```python
#function to execute Case 3
def case3(size, filename):
    print("--------------------------- CASE 3 -------------------------")
    print("Error is detected by VRC but not by CRC.")
    print("-------------- Vertical Redundancy Check -------------------")
    type = 1
    s = Sender(size)
    s.createFrames(filename,type)
    s.displayFrames(type)
    zeropos = []
    onepos = [[5]]
    s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
    r = Receiver(s)
    r.checkError(type)
    r.displayFrames(type)
    print("-------------- Cyclic Redundancy Check ---------------------")
    type = 4
    poly = "100"
    print("Generator Polynomial:",poly)
    s = Sender(size)
    s.createFrames(filename,type,poly)
    s.displayFrames(type)
    s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
    r = Receiver(s)
    r.checkError(type,poly)
    r.displayFrames(type)
    print("------------------------------------------------------------\n")
```

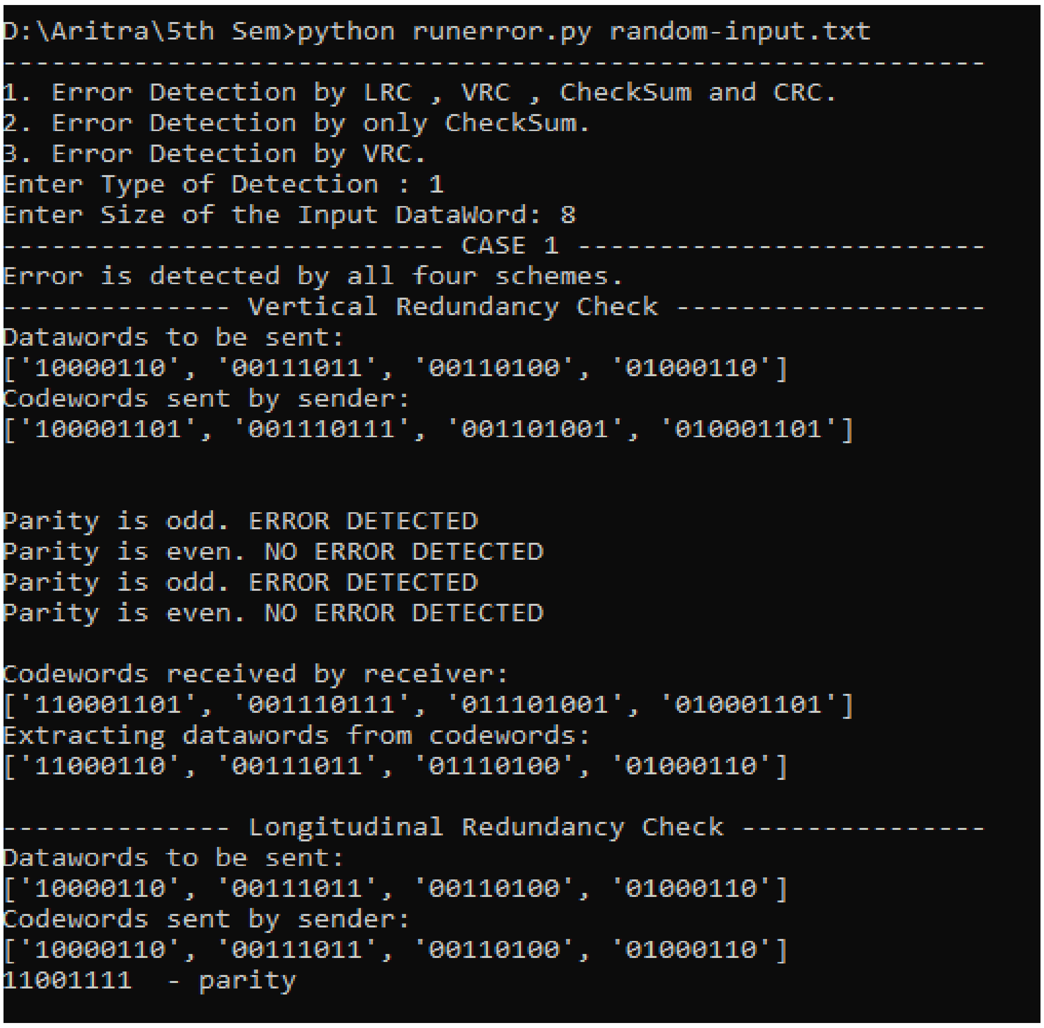


```python
#driver function to run the error detection module
if __name__ == "__main__":
    print("------------------------------------------------------------")
    print("1. Error is detected by all four schemes.")
    print("2. Error is detected by checksum but not by CRC.")
    print("3. Error is detected by VRC but not by CRC.")
    case = int(input("Enter Case Number : "))
    if case == 1:
        size = int(input("Enter length of the dataword: "))
        generateRandomInput(size*4, sys.argv[1])
        case1(size, sys.argv[1])
    elif case == 2:
        size = 8
        case2(size, sys.argv[1])
    elif case == 3:
        size = 6
        case3(size, sys.argv[1])
    else:
        print("You entered invalid choice.")
```

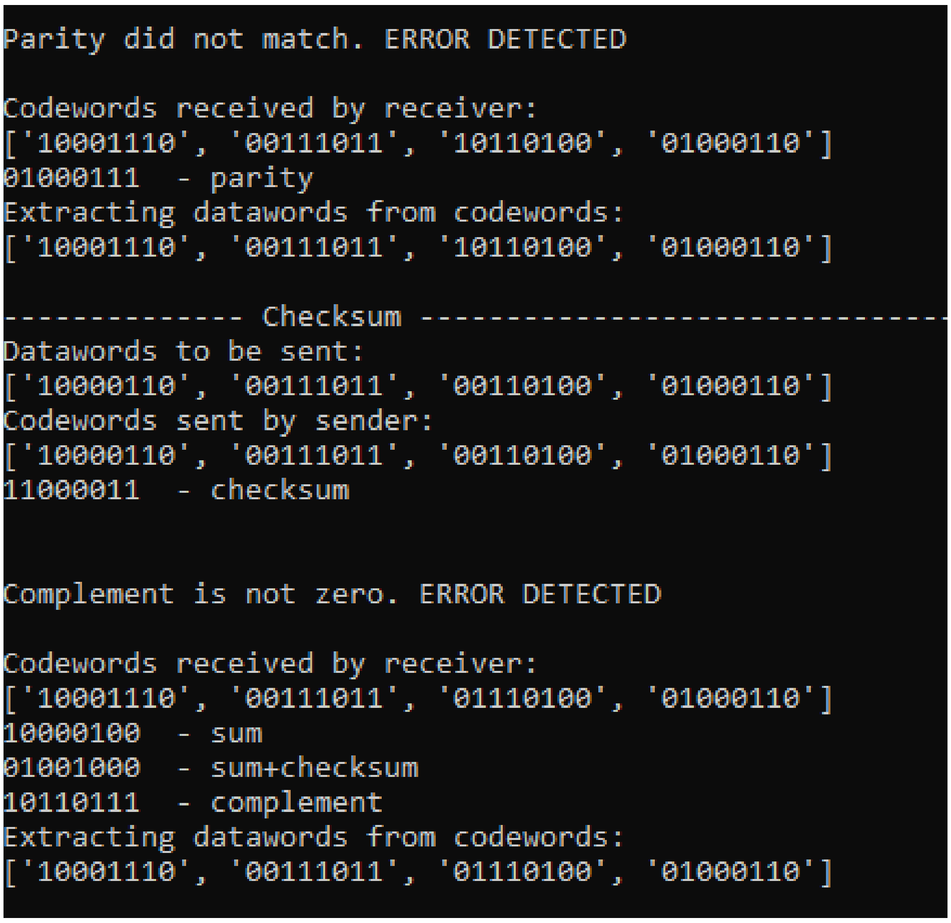
## Method descriptions of run.py :

- **injectRandomError(frames)** : This method is used to inject errors in random positions of the codewords which are sent by sender program. The randint() function of random python module has been used for generating random position within the size of the codewords
- **injectSpecificError(frames,zeropos,onepos) :** This method is used to inject errors in specific positions of the codewords. The positions in which the error is to be inserted is passed as arguments, as zeropos and onepos. Zeropos contains the list of positions in which the value of those positions will be made 0, whereas Onepos containes the list of positions in which the value of those positions will be made 1.
- **case1(size,filename) :** This method is implemented to make function invocations and displaying the results of all schemes, considering the case 1 of the problem statement (Error is detected by all four schemes). The dataword size and input file name is taken as function arguments
- **case2(size,filename) :** This method is implemented to make function invocations and displaying the results of Checksum and CRC, such that the case 2 of the problem statement satisfies (Error is detected by Checksum but not CRC). The dataword size and input file name is taken as function arguments
- **case3(size,filename) :** This method is implemented to make function invocations and displaying the results of VRC and CRC, such that the case 3 of the problem statement satisfies (Error is detected by VRC but not CRC). The dataword size and input file name is taken as function arguments

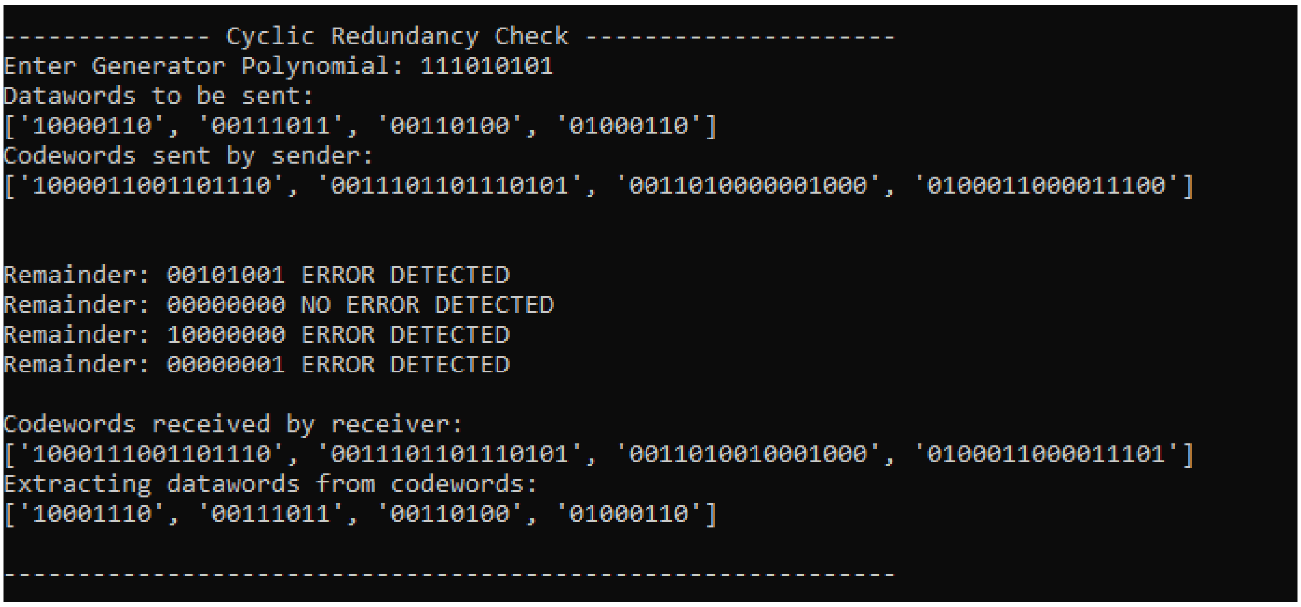
# <u>TEST CASES :</u>



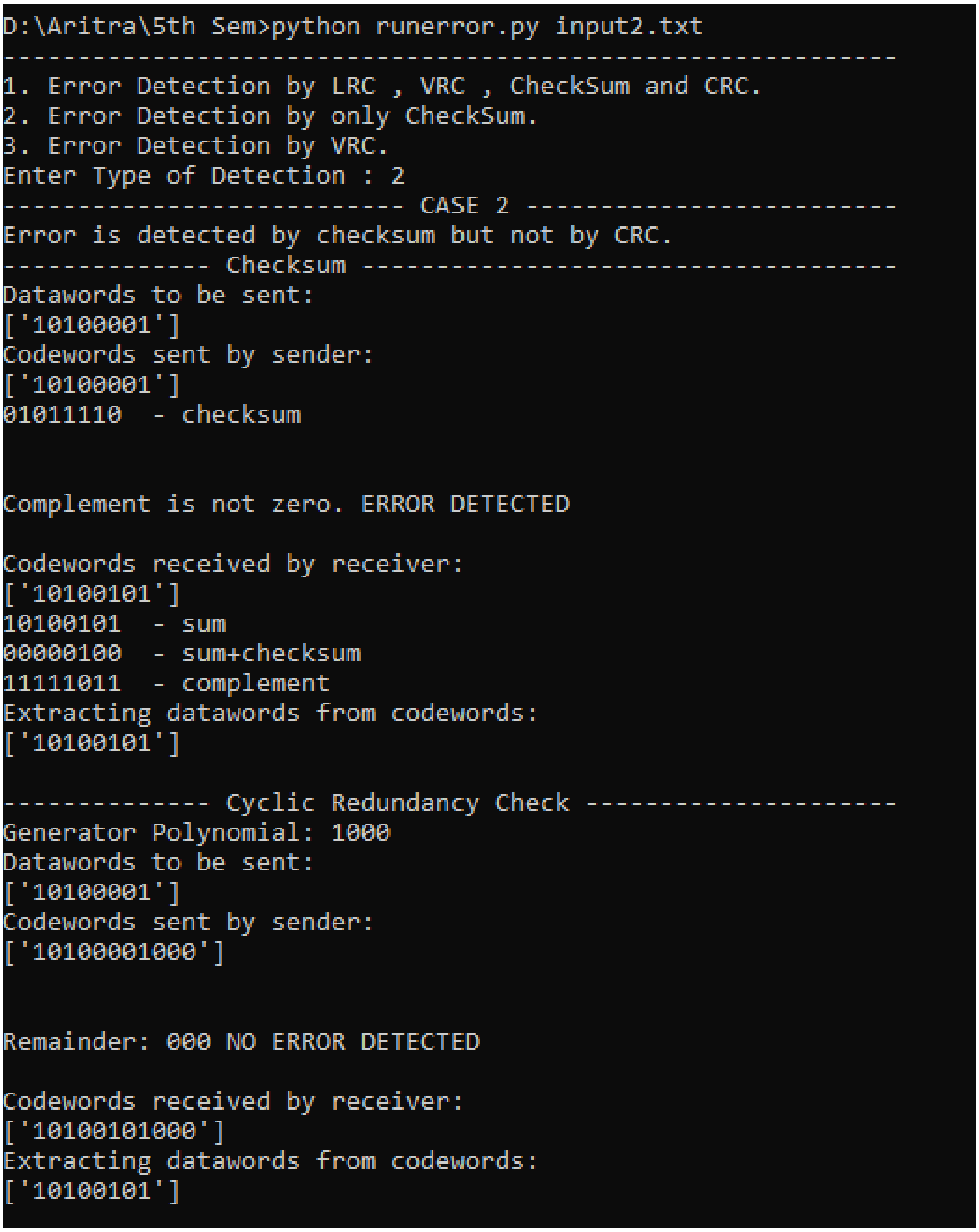
```
D:\Aritra\5th Sem>python runerror.py random-input.txt
------------------------------------------------------------
1. Error Detection by LRC , VRC , CheckSum and CRC.
2. Error Detection by only CheckSum.
3. Error Detection by VRC.
Enter Type of Detection : 1
Enter Size of the Input DataWord: 8
---------------------------- CASE 1 ------------------------
Error is detected by all four schemes.
-------------- Vertical Redundancy Check -------------------
Datawords to be sent:
['10000110', '00111011', '00110100', '01000110']
Codewords sent by sender:
['100001101', '001110111', '001101001', '010001101']


Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED

Codewords received by receiver:
['110001101', '001110111', '011101001', '010001101']
Extracting datawords from codewords:
['11000110', '00111011', '01110100', '01000110']

-------------- Longitudinal Redundancy Check ---------------
Datawords to be sent:
['10000110', '00111011', '00110100', '01000110']
Codewords sent by sender:
['10000110', '00111011', '00110100', '01000110']
11001111  - parity
```



```
Parity did not match. ERROR DETECTED

Codewords received by receiver:
['10001110', '00111011', '10110100', '01000110']
01000111  - parity
Extracting datawords from codewords:
['10001110', '00111011', '10110100', '01000110']

-------------- Checksum ------------------------------
Datawords to be sent:
['10000110', '00111011', '00110100', '01000110']
Codewords sent by sender:
['10000110', '00111011', '00110100', '01000110']
11000011  - checksum


Complement is not zero. ERROR DETECTED

Codewords received by receiver:
['10001110', '00111011', '01110100', '01000110']
10000100  - sum
01001000  - sum+checksum
10110111  - complement
Extracting datawords from codewords:
['10001110', '00111011', '01110100', '01000110']
```



```
-------------- Cyclic Redundancy Check ---------------------
Enter Generator Polynomial: 111010101
Datawords to be sent:
['10000110', '00111011', '00110100', '01000110']
Codewords sent by sender:
['1000011001101110', '0011101101110101', '0011010000001000', '0100011000011100']


Remainder: 00101001 ERROR DETECTED
Remainder: 00000000 NO ERROR DETECTED
Remainder: 10000000 ERROR DETECTED
Remainder: 00000001 ERROR DETECTED

Codewords received by receiver:
['1000111001101110', '0011101101110101', '0011010010001000', '0100011000011101']
Extracting datawords from codewords:
['10001110', '00111011', '00110100', '01000110']

------------------------------------------------------------
```



```
D:\Aritra\5th Sem>python runerror.py input2.txt
------------------------------------------------------------
1. Error Detection by LRC , VRC , CheckSum and CRC.
2. Error Detection by only CheckSum.
3. Error Detection by VRC.
Enter Type of Detection : 2
--------------------------- CASE 2 -------------------------
Error is detected by checksum but not by CRC.
-------------- Checksum ------------------------------------
Datawords to be sent:
['10100001']
Codewords sent by sender:
['10100001']
01011110  - checksum


Complement is not zero. ERROR DETECTED

Codewords received by receiver:
['10100101']
10100101  - sum
00000100  - sum+checksum
11111011  - complement
Extracting datawords from codewords:
['10100101']

-------------- Cyclic Redundancy Check ---------------------
Generator Polynomial: 1000
Datawords to be sent:
['10100001']
Codewords sent by sender:
['10100001000']


Remainder: 000 NO ERROR DETECTED

Codewords received by receiver:
['10100101000']
Extracting datawords from codewords:
['10100101']
```

# RESULTS

## Vertical Redundancy Check (VRC):

**Vertical Redundancy Check** is also known as Parity Check. In this method, a redundant bit also called parity bit is added to each data unit. This method includes even parity and odd parity. Even parity means the total number of 1s in data is to be even and odd parity means the total number of 1s in data is to be odd

If the source wants to transmit data unit 1100111 using even parity to the destination. The source will have to pass through Even Parity Generator.

Parity generator will count number of 1s in data unit and will add parity bit. In the above example, number of 1s in data unit is 5, parity generator appends a parity bit 1 to this data unit making the total number of 1s even i.e 6 which is clear from above figure.

Data along with parity bit is then transmitted across the network. In this case, 11001111 will be transmitted. At the destination, This data is passed to parity checker at the destination. The number of 1s in data is counted by parity checker.

If the number of 1s count out to be odd, e.g. 5 or 7 then destination will come to know that there is some error in the data. The receiver then rejects such an erroneous data unit.

## Longitudinal Redundancy Check (LRC):

In contrast to VRC, LRC assigns a parity byte or a nibble along with the message to be transmitted.

Suppose the message to be transmitted is

Word1 Word2 Word3

Then, we compute the even parity nibble as follows:

(Counting 1s in all bit positions odd->1 , even->0)

Word1
Word2
<u>Word3</u>
Codeword for Sender

We note that in this scheme, the number of 1's in each column including the bit in the parity nibble must be even.

In this method, data which the user want to send is organised into tables of rows and columns. A block of bit is divided into table or matrix of rows and columns. In order to detect an error, a redundant bit is added to the whole block and this block is transmitted to receiver. The receiver uses this redundant row to detect error. After checking the data for errors, receiver accepts the data and discards the redundant row of bits.

## Checksum:

**Checksum** is the error detection method used by upper layer protocols and is considered to be more reliable than LRC, VRC and CRC. This method makes the use of **Checksum Generator** on Sender side and **Checksum Checker** on Receiver side.

At the Sender side, the data is divided into equal subunits of n bit length by the checksum generator. This bit is generally of 16-bit length. These subunits are then added together using one's complement method. This sum is of n bits. The resultant bit is then complemented. This complemented sum which is called checksum is appended to the end of original data unit and is then transmitted to Receiver

If the data unit to be transmitted is 10101001 00111001, the following procedure is used at Sender site and Receiver site

**Sender Site :**

| | |
|---|---|
| 10101001 | Block 1 |
| 00111001 | Block 2 |
| 11100010 | sum (using 1s complement) |
| **00011101** | **checksum** (complement of sum) |

**Receiver Site :**

| | |
|---|---|
| 10101001 | Block 1 |
| 00111001 | Block 2 |
| 00011101 | Block 3 or **Checksum** |
| 11111111 | sum |
| **00000000** | sum's complement |

**Result is zero, it means no error**

## Cyclic Redundancy Check (CRC):

- Unlike checksum scheme, which is based on addition, CRC is based on binary division.
- In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number.
- At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted.

- A remainder indicates that the data unit has been damaged in transit and therefore must be rejected

CRC performs mod 2 arithmetic (exclusive-OR) on the message word using a divisor polynomial. CRC Bits are calculated by finding the remainder of division of message word by the divisor and then CRC bits are appended to the word and send to Receiver as Code Word. Division can be achieved through polynomial division, CRC bits size is of length -> L-1, where L is the size of the Divisor Word.

## ANALYSIS

**Vertical Redundancy Check (VRC):**

- It Successfully detects all single bit errors, but partially detects multiple bit errors, like in the case of odd bit errors
- Ex : message at the sender side : 1 0 0 1 1 1   1 0 0 1 0 0
- Considering corruption of bit 4 of the 1st block
- The above example also suggests that not all even bit erros (multiple bit error with the number of bits corrupted even) are detected by this scheme. If even bit error is such that the even number of bits are corrupted in each nibble, then such error is unnoticed at the receiver. However, if even bit is such that at least one of the nibble has odd number of bits in error, then such error is detected by VRC.

**Longitudinal Redundancy Check (LRC):**

LRC can detect the errors which get distributed in such a way that all the columns have even numbered bit flips. In such a situation the xor will remain unchanged and error will go undetected. This is because the LRC divides the code word in packets of equal bits and then calculates bitwise XOR of the packets, so an odd number of bit flips in a particular position will generate '1' for that position and the resultant check bits will be non-zero. So single bit errors and burst errors of length 8 can be detected as in these cases, only 1 bit per column is changed. However in case of burst error of length 9, the first and last error bit will overlap, and if the bits in between do not flip, no error is detected. In general, if word_size is odd, LRC can detect all even-numbered errors and if word_size is even, LRC can detect all odd-numbered errors.

**Checksum:**

1. If multiple bit error is such that in each column, a bit '0' is flipped to bit '1', then such an error is undetected by this scheme. Essentially, the message received at the

receiver has lost the value 1 1 1 1 1 with respect to the sum. Although, it loses this value, this error is unnoticed at the receiver.

2. Checksum works by dividing the codeword into words and then calculatin1's complement binary addition of these words. It is unable to detect any error if the net change in sum is 0 or a multiple of 2^(size of each word)

3. Also, multiple bit error is such that the difference between the sum of the sender's data and the sum of receiver's data is 1 1 1 1, then this error is unnoticed by the receiver.
4. The above two errors are undetected by the receiver due to the following interesting observations.
5. Similar to other error detection schemes, checksum detects all odd bit errors and most of even bit errors.

**Cyclic Redundancy Check (CRC):**

1. In CRC-8, the coefficient of x0 is 1 and 6 high bits are present, so it can detect all single bit errors. If the difference between 2 error bits is less than the number of check bits, the error is always detected. This CRC module hence can detect all burst errors of burst length less/equal to the degree of polynomial. Since the degree of the divisor polynomial is 8, the remainder in case of burst errors of length <= 8 will always be non-zero and hence these errors will be detected with 100% accuracy
2. CRC has the ability to detect multiple bit errors efficiently as compared to other methods but in case of transmission errors where flipping the sender bits results in an polynomial that still divides the divisor in that case the receiver does not detect the error correctly
3. Thus, choosing an appropriate divisor is crucial in detecting errors at the receiver if any during the transmission.

## COMMENTS

This assignment has helped me to achieve a in better understanding of Error Detection Methods used in computer Networks