Elementary Cryptographic Programming

Useful, but Simple Programs You Can Write to Help Solve Ciphers and Reduce Tedium

By SHMOO

Preface

Are you new to cryptography? New to programming for cryptography or just plain new to programming? This booklet is for you.

Who might you be? Maybe someone who has never written code, but is tired of writing out everything by hand. Maybe someone who writes a little code at work now and then, when absolutely forced to. Maybe you're able to do some things with spreadsheets and are intrigued by "real" programming.

The good news, for anyone like that, is that hobby cryptography is a wonderful place to teach yourself programming – as little or as much as you like. Starting from the beginning if that's where you are. The bonus is that each example is something that is at least potentially useful; always a plus.

This booklet cannot teach you programming from scratch. That's the work of many textbooks or perhaps on-line tutorials. These are all larger than this entire document. But, if you've at least written "Hello, World" in some computer language, that's probably enough to start this discussion. If you're really just starting out, try out a tutorial in some language (this booklet will use Python) and then follow along with this as you go.

Preliminaries

An important underlying assumption: This work assumes the reader has access to *The ACA and You.* It briefly but adequately describes ACA systems commonly seen. If you want a comprehensive list of these systems, and basic solution methods, obtain The Cryptogram Index, currently at https://www.cryptogram.org/members/downloads/CmIndexND2023.pdf and then find the index entries for *Novice Notes.* Or, download the *Practical Cryptanalysis* series from the same website (five volumes). Either covers most ACA ciphers, past and present.

So, if you've not heard of an Aristocrat, a Patristocrat, or a Morbit before, find one of those sources first to know exactly what kind of cipher is being discussed. This booklet assumes you've read up on that.

Very Basic Basics

Some things show up regularly in any kind of computer program. So much so, that every computer language has them, one way or another. We use these in writing cryptography programs:

1. Integers. Integers are exactly what they sound like. Humans call them "counting numbers". In most computer languages, integers take on positive and negative values. In some languages, a special variation of integer only takes on positive values. As one programs, one does find cases where negative numbers aren't wanted and/or get in the way. But Python, like several other languages, only provides integers that are positive and negative which is, in the end, very workable after all.
2. Characters. Individual characters. The letter A is a character.
3. Floating point. That is, what algebra class called "real" numbers. Numbers with fractions. In computers, fractions are finite which turns out to matter a bit. Computer floating point can represent very small numbers indeed, but they are still fractions and they can have rounding error when least expected. absoluteValue(a-b) < 0.00001 is safer to code than the comparison a

== b in many languages when either a or b are floating point. Much of the time, rounding errors would not matter, but when they happen, they can cause subtle errors.

4. Booleans (often called "bool"). Many earlier languages didn't have booleans, but instead defined 0 as false and either 1 as true or "anything but zero" as true. Python booleans are assigned True or False and these are not character strings but actually *mean* True and False.
5. Arrays. Collections of something. That's what an array is for. Suppose you want to create a frequency count. The natural implementation is an array of 26 slots, each slot containing the frequency counts for one particular letter in a cryptogram. The frequency of letter A might be the first slot; the frequency of E might be the fifth.
6. Strings. Collections of characters. Many languages treat strings a little differently than other kinds of arrays, but in Python the distinction is basically syntactic. They are arrays of characters.
7. Comments. In every computer language there is a way to "set off" text that isn't meant to be part of the program, but instead explain the program. In Python the # character means "comment." Everything to the right of a # will be treated as comment, not code. In many other computer languages these two characters: // commence a comment.

Now, most readers will have seen these ideas, but we're establishing a common understanding. It turns out, you can do a great deal of cryptography with nothing more than these basic primitives. It is easy to find tutorials in a variety of computer languages and most start with these things. So, long before you get to the exotic bits, you can learn these basic elements in a handful of lessons and then start writing useful cryptography code in your language of choice.

A Brief Word About Declaring Things

In most computer languages, a variable (an item you use to store data) has to be declared before it is used. A typical declaration might look like this:

```
int myScore = 0;  // this declares "myScore" as an "integer", initializes to 0.

string myProgHeader = "This is my Program";  // this is a character string
```

In Python, there is instead "duck typing" which is weird and wonderful; easy to get used to. In essence, you just start using a variable and the interpreter figures out what the type is based on what value you gave it on a "if if quacks like a duck, it's a duck" basis:

```
myScore = 0   # myScore is treated like an integer

myProgHeader = "This is my Index of Coincidence Program" # myProgHeader is a string.
```

You can even arrange to set myScore to a string later on, but though allowed, it is a bad practice; you might yourself lose track of what myScore is. Variables and their names are plentiful. Keep the usage consistent.

An Aside About Indenting and "Control" Statements

In many computer languages, it has become customary to "indent" code that is dependent on other code. The main concepts are the "if", the "while" and the "for". There are sometimes other names for these, but most languages use these names about the same way. In Python, indenting is not a nicety, but indicates what is to be done when "if", "while" or "for" is true. When you stop indenting, the "if" or "while" or "for" no longer is controlling anything. You've moved on with the program.

Examples:

1. “if” with “else”. The if statement means “if” the statement evalues to “true”, then do some body of work. Otherwise, skip down to a matching “else” if there is one. In Python, you might see:

```
if(someChar>=’A’ and someChar<=’Z’) :
    doSomethingTo(someChar)
else
    doSomethingElseTo(someChar)
# Continue here whether “if” or “else” was executed.
```

2. “if” without “else”. The if statement without an else simply means the body of work executes when true, is skipped when it isn’t.

```
If(someChar>=’A’ and someChar<=’Z’) :
   onlywhenTrueDoSomethingTo(somechar)
# Continue here whether you executed the “onlywhenTrue..” code or not
```

3. “for” and “while” are similar. A “for” statement implies some kind of iteration.

```
for i in range(0,26) :  # i will take on values 0…12…25 over time
   freq[i]=0
# Continue here only after “freq[i]=0” executes 26 times, once per array slot
i=0
while(i<26) :
   freq[i]=0
   i=i+1
# Continue here only after freq[i]=0 and the i=i+1 execute 26 times.
# Yes, in this example, the “for” and the “while” have identical results.
```

What Computer Language?

Oh, dear. Like politics and religion, do not bring up language choices to your programmer friends. There really are two classes of programmer. The first class of programmer learns one computer language early in their career and then never learns another. They will forcefully state that their language, whatever it is, can be deployed against any problem. This is actually true, and that’s very fortunate. It is also true that some languages are advantaged over other languages for certain problems. The second class of programmer learns this, appreciates it, and masters many computer languages so as to deploy the most effective one for a given purpose. It doesn’t matter which class you decide to join. But, it does raise a problem for the author. How to express ideas that at least “look like” code?

In this tome, the author has favored Python. Why? Because it is a great teaching language. It is very concise and yet, in some critical places, able to express complex ideas simply. As important, it expresses simple ideas simply. It is also available on your PC, on your Mac, even on your Linux box. Programs easily move from one environment to the other if that interests you. That’s not always true of other languages. Just be sure to use Python 3 (3.8, 3.12, 3 point something) and not earlier versions.

What Kind of Environment?

This is really two questions. The first is "what will your program look like?" For beginners, I recommend against using graphical interfaces. There is a lot more to learn to have "do" graphical and even simple things can take elaborate structure. By contrast, what are usually called "console" applications are much simpler and, for the hobby cryptographer, often every bit as useful. It just means the output comes to an ordinary terminal session or a Windows Command Line session, line by line.

The second is "where am I developing the code?" In the old days, one simply used an ordinary editor like Windows' Notepad and then invoked the compiler to create the machine code and then executed the machine code.

Nowadays, some sort if "interactive development environment" (IDE) is preferred. In Python, I use the one called IDLE and it is built-in to the Python distribution. Invoking it in Windows is a little obscure however. Look carefully at the instructions. If you are already familiar with an IDE (Microsoft Visual Studio, Microsoft Visual Code, or the open source Eclipse), you might use that environment and pick a language it supports. Python, used as an exemplar here, is still very relevant and it should be possible to convert the examples to your own favorite language.

An IDE does at least three significant things: It mostly or entirely checks your syntax as you edit your code, giving instant feedback on errors. It makes assembling larger programs from multiple source files easier (something we all eventually need). Finally, it has good, built-in facilities for debugging code.

What would the "Hello World" of Cryptography Be?

Most introductory texts usually have a very short example where the program prints out the immortal phrase "Hello World". But, let's go beyond that and talk cryptography.

Our equivalent of "Hello World" would be a frequency count. That is, input cipher, output a list of values, one per English alphabetic letter.

There are two design issues that may or may not be obvious.

1. The first is that the input is in characters. These need to be converted to integers. This is both easy and hard. It is easy in the sense that inside the computer, everything is numbers. It is a little hard in that computer language may be very strict about what a character is, so you will have to start with whatever a character is and transform it to a number using functions the language defines. Even then, the numeric values, in "raw" form, are in something called ASCII. That means our upper case ("capital letter") ciphertext will be ASCII A through ASCII Z. That turns out to be integer values 65 through 90, consecutively. As you'll see, the "consecutively" part will be exploited. But, as even short experience will make clear, we would also prefer to represent A as zero, B as one, E as four, and Z as 25 whenever we treat them as integers. So, each character A, B, C…Z must transition to an integer representing the ASCII value and then must be corrected from the 65, 66, 57…90 "raw" ASCII value to the 0, 1, 2…25 value we want.
2. The other is how to represent the array. The obvious choice is to make it an array of integers. Slightly less obvious is that position 0 should be for counting up letter A, position 1 should be B, position 25 should be Z. But why position "zero"? Well, it turns out that more modern computer languages number their arrays from zero and not the more human intuitive value of

one. There’s a long list of reasons for this, but you don’t care about a fifty-year-old argument, especially as the argument is settled. We all just have to program to it.

These two factors make it possible to see what the actual summation would look like:

```
val = ord(currPTLetter) – ord(‘A’) # Subtracting ord(‘A) converts
                  # the original ASCII value to the 0..25 we want
freq[val]=freq[val]+1;
```

What’s going on here? Well, “val” is an integer that receives the value 0 for A, 1 for B, 2 for C and 25 for Z by subtracting the ASCII value of A. “ord(x)” is Python speak for “give me the ASCII value of the character x”. Other languages may have a name like “asc” for what Python calls “ord”. Ord handles a little more that ASCII, as there is a world of characters beyond ASCII, but ignore that for now.

So, the overall shell is something like this:

```
TrivalAlph=”ABCDEFGHIJKLMNOPQRSTUVWXYZ”
CT=”Four score and seven years ago our forefathers brought forth”
CT=CT.upper()   # convert all lower case to upper with a function
freq=[]
for i in range(0,26) :
    freq[i]=0
for i in range(0,len(CT)) :
    currPTLetter = CT[i]
    if(Within_ASCII_Alphabet(currPTLetter)) : # skip spaces, punctuation.
       val= ord(currPTLetter) – ord(‘A’) # Val is 0 for A, 4 for E, 25 for Z
       freq[val]=freq[val]+1  # Notice how val is used to access the array.
# Now print out the results
for i in range(0,len(freq)) :
    print(TrivalAlph[i]+”: “+str(freq[i]),end=””) # does not end the line
    If((i%5)==4) : print() # ends “this” line, starts a new line of output
print() # ends the final line
```

Now all that remains is for the reader to figure out how to code “Within_ASCII_Alphabet(letter)” to return True for A through Z, False for the rest.

Note the printing of the answer at the end. First, the “trivial alphabet” is used to label what each slot in the array means. Note also that it prints five results per line of output. Every language has some variation of print that either ends the line (often called PrintLine) and one that doesn’t (often called Print). In Python, there is only *print*; one includes the special parameter *end=””* to mean “do not start a new line.” If you don’t, then *print* does end the line. The % function is “modulus” (the remainder after old fashioned long division), so values of i between 0 and 3, 5 and 8, etc. all end up on the same line with value 4, 9, etc. This is perhaps not vital in this particular example, but the need to print full or partial lines occurs often enough that it is worth showing in this example.

There are many programs that fit this basic shell. For instance, enciphering programs for Aristocrats, Patristocrats, Vigenere, even Quagmire follow this pattern. Details change; the shell remains.

Exercises For the Beginner

The above shell is, as already pointed out, useful. There are a couple of recurring ideas:

1. Somehow, the input ciphertext gets read into a string variable. One way or another, the letters are all capitalized. So it is A-Z and maybe some spaces and punctuation.
2. There is a basic loop that starts at the beginning of each letter and processes them, one by one. In effect, the string is treated as an array of characters.
3. The alphabetic letters (A-Z) have "something" done to them. The other characters are (usually) ignored.
4. Part of the "something" done to each letter is to turn it into an integer number with A as zero, B as one, E as four, and Z as 25 (for a variety of reasons that experience will make clear, computers like to number their arrays from zero and so keeping the letters as 0 to 25 also makes a lot of sense).

There are many elementary but useful programs that are a variation of the above shell.

Problem 1. Perform a simple substitution encipherment. Instead of the freq array above, create an array that has a fixed, *standard* alphabet in it. Perhaps use this one: `"EXPOUNDABCFGHIJKLMQRSTVWYZ".` Remember that in Python, a string is an array. Use that as the ciphertext alphabet to encipher the letters. You can write this without a second plaintext alphabet for the trivial alphabet ("ABC…XYZ") – that can be implied. If your plaintext is: FOUR SCORE AND SEVEN YEARS and EXPOUND…WYZ is the key, the ciphertext is: NJSM QPJMU EIO QUTUI YUEMQ.

Problem 2. Shift the original standard alphabet by one or more positions. Simple math reveals that when you shift 26 positions, you have shifted zero positions. This means that you only need to write code to shift in one direction and only one letter. Let's say you shift left one position. If you want to shift right one position, you shift left 25 positions. So, shift the alphabet a specified amount and then re-perform the simple substitution. If you shift the EXPOUND *standard* alphabet two positions left ("POUNDABCFGHIJKLMQRSTVWYZEX"), the new ciphertext is: ALVR SULRD PKN SDWDK EDPRS.

Problem 3. Make the cipher alphabet the *trivial* alphabet and the plaintext alphabet the *standard* alphabet. Teach yourself the Python "find" function so as to avoid an extra loop operation. The magic line you need will look something like: offset = ptAlphabet.find(plaintext[i]) where i is your loop variable. Remember that *find* returns offset -1 if you have spaces and punctuation in your plaintext because they are not present in ptAlphabet. Account for that and simply repeat the plaintext in that case. Variable "offset" will tell you how to create the ciphertext.

With "EXPOUNDABCFGHIJKLMQRSTVWYZ" as the plaintext key and the *trivial* alphabet as ciphertext key, the new encipherment of FOUR SCORE AND SEVEN YEARS is: KDET UJDTA HFG UAWAF YAHTU

Problem 4. Construct a *standard* alphabet from a keyword and the *trivial* alphabet. Remember that the keyword may have repeated letters (EXPRESS). Account for that. EXPRESS should produce `EXPRSABCDFGHIJKLMNOQTUVWYZ.`

Problem 5. With problem 3 solved, you now have the ability to do decipherment as well. Figure out how to rearrange the alphabets so that you can repeat problem three, but with KDET UJDTA HFG UAWAF YAHTU as the input and FOUR SCORE AND SEVEN YEARS as the output.

Problem 6: With Problems 3, 4, and 5 solved, you now have the essence of a program that can construct Aristocrats. This will take more than one "core" loop as above, because the editor will want to see a worksheet as well as the enciphered text, but the pieces are in place.

With the previous four problems solved, you now have the ability to do both encipherment and decipherment. That means that your deciphering routine can be used to form an interactive Aristocrat solver. You simply start with the *trivial* alphabet for plaintext and 26 dot characters for the ciphertext alphabet. So, decipherment must now account for unassigned letters.

Problem 7. Add an outer loop that accepts input from the user of the form Z=E which means "cipher Z is now represented by plaintext E". Figure out how to adjust the ciphertext alphabet and redisplay the results. Remember to initialize the ciphertext alphabet to all periods so that unassigned letters are handled and displayed. Display the keys as well. You now have replicated "solve a cipher" on the cryptogram.org site.

Two Dimensional Arrays and Another Useful Program Template

Another element that is so far not part of the discussion is two-dimensional arrays.

Consider this Polybius square, keyword GERMINATE, located in a variable named "square":

	0	1	2	3	4
0	G	E	R	M	I/J
1	N	A	T	B	C
2	D	F	H	K	L
3	O	P	Q	S	U
4	V	W	X	Y	Z

We can see the classic Polybius square, but what about the numbers above and to the left? Those are what would be the array indices of a two-dimensional array.

So, in some programming languages, letter N would be at square[1,0] and letter U would be at square[3,4].

But, starting with the C language, another notation has dominated: square[1][0] for N and square[3][4] for U.

Declaring such arrays is a little more challenging in these languages:

```
square = [
['G','E','R','M','I'],
['N','A','T','B','C'],
['D','F','H','K','L'],
['O','P','Q','S','U'],
['V','W','X','Y','Z'] ]
```

Note the omission of letter J from the example. One has to pre-process the plaintext using the Python *replace* function: pt=pt.replace(“J”,”I”) . . .to prepare the text for Polybius encipherment.

By the way this square2:

```
square2=[
["GERMI"],
["NATBC"],
["DFHKL"],
["OPQSU"],
["VWXYZ"] ]
```

. . .is not quite the same as “square”. It is the same as this square3:

```
square3= [
[['G','E','R','M','I']],
["NATBC"],
["DFHKL"],
["OPQSU"],
["VWXYZ"] ]
```

The interested reader can find out why (hint: think three dimensions, not two).

Now, at any rate, as long as we put exactly one character per cell, we have a standard Polybius square and we can encipher things like Playfair, Phillips, Checkerboard. By having more than one, we can do Two-Square, Tri-Square and Four-Square.

When we know the dimensions ahead of time, we can declare them as above. But, what if we don’t know? Well, sometimes we can do so anyway if we can stand lots of text in our code. We might precompute, for instance, all the reasonable rectangle sizes for a route transposition. How would we do that? We would write a program that generates snippets of program and edit those snippets into the program we care about. Some of the output might look like this:

```
rect3x5 = [ [‘*’,’*’,’*’,’*’,’*’], [‘*’,’*’,’*’,’*’,’*’], [‘*’,’*’,’*’,’*’,’*’]]
```

The program we care about, when it knows it has a ciphertext of only fifteen characters, might select rect3x5 from whatever alternatives exist, and replace the ‘*’ characters with the ciphertext, according to the route being tried. Well, a fifteen character length is actually a bit short, but it’s not a big deal to adapt this idea to larger sized square. The code to select them is likewise tedious but straightforward. A given text length will have one, two, four, maybe six alternatives. Seventy two will have more rectangles than fifty two. This idea isn’t all bad; a big mass of tedious, but correct code is often a nice trade-off. Modern PCs have big disks on them; even a few megabytes of program code isn’t a big issue these days and it is seldom that large.

The author finds the idea of generating this kind of boilerplate, pre-computed code compelling. Yet, it is not much discussed in the literature nor did others in his professional circles do much of it either. The nice thing about it is that if there is an error, fixing one error tends to fix a lot of things at once. And, much of it is so bland that errors are not common; yet, it can replace tricky code that avoids precomputation.

However, there remain cases where precomputation isn't available.

Suppose one decides to take an Aristocrat and break it up into an array of words. Python, like many modern languages, can handle irregularly sized arrays. So, the plaintext "THE QUICK BLUE FOX" which is represented as a single array of characters, can be broken up into an array of this sort:
["THE", "QUICK", "BLUE", "FOX"]. As can be seen, the final dimension varies. But there is another issue. How many words are there going to be? In the end, the computer needs to deal not just with the fact that words vary in length, but in a given puzzle, the number of words vary. So, here's a short program that breaks up a text into words. For simplicity, it assumes that the text is entirely letters of the alphabet with only a single space character between them. The general case is not that much harder; just messier and left to the reader:

```
CT="THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG" # CT/PT all the same to this code.
wordList=[]
word= ""
for i in range(0,len(CT)) :
   if(CT[i]>="A" and CT[i]<="Z") :
      word=word+CT[i] # this is concatenation for strings
   else :
      wordList.append(word) # this extends the array wordlist by "word".
      word=""
if(len(word)>0) :
   wordList.append(word)
# The 'for w…' is a way of accessing an array without a "range" variable.
for w in wordList :
   print(w)
print(str(wordList))  # str means "turn whatever wordlist is into a string"
```

output is:

```
THE
QUICK
BROWN
FOX
JUMPS
OVER
THE
LAZY
DOG
['THE', 'QUICK', 'BROWN', 'FOX', 'JUMPS', 'OVER', 'THE', 'LAZY', 'DOG']
```

As can be seen, the code does what was asked of it. This, too, is a template for a lot of other programs.

1. The phrase “word=word+Ct[i]” is concatenation when strings are involved. If *word* contains QUIC and *CT[i]* is K, then the result of the concatenation changes word to QUICK. Why is the “+” operator abused this way? Well, adding up two character strings doesn’t make mathematical sense. So, Python and several other languages decided to make + mean “concatenation” for strings.
2. Note the CT[i]>=’A” and CT[i]<=’Z’ . . . which means that *CT[i]* is between ASCII A and Z. So, if the input is upper case, it means that we have a letter of the alphabet and will capture all letters of the alphabet. If it is something else, be it a space or punctuation, it will go to the “else” clause.
3. Append is the function that adds a new element to an array. This kind of dynamic growth is straightforward in Python. It exists in other languages, but it takes more effort to express it.
4. print(str(wordList)) is a very nice bit of function. Str looks at whatever it is given and figures out some way to make it into a string. Because it is an array, the str function supplies the brackets and the quotation marks. You will need str if you want to mix characters and numbers in the same print statement. You will also use concatenation.

Subroutines and Deleting from Arrays

Another commonplace issue that can arise with arrays is the need to delete something from it. In traditional arrays, this isn’t really possible, but in Python’s arrays, it can be done. This also allows an excuse to talk about another basic building block: Subroutines and functions.

The only difference between a subroutine and a function is that the function returns something. It could be a number, a string, an array; whatever makes sense. Subroutines do their work and just. . .return. One codes a subroutine if one wished to invoke something many times or even just to isolate key code.

Many of the above examples can and should be turned into subroutines or be implemented, in part, with subroutines, because they would be a piece of a larger whole. One big advantage of subroutines is that with a little work, they can be tested without every other part of the program working or even before everything else is written. Here, subroutines permit an elegant trick: Returning as a way of avoiding trouble. In Python, one runs into situations like this:

```
for w in wArray : # a more direct way of accessing wArray
    If(hasTooManyVowels(w)) :  # w is a bad thing we don’t want
      delete w  # This deletes w, exactly w, from the array
      # Now what???
```

The problem with the “now what” is that, like many languages, Python doesn’t really handle the case where you are looping through some array, with or without range numbers, and then you delete something. They basically promise errors galore from here if you don’t do something drastic. Well, in the case coming up, employ a clever idea: Return false after every delete and return true if one never deletes. True would indicate the code completed.

Why do this? Because, one way or another, after a delete you must start over anyway. So, why not start at the beginning?

Problem: Extending the “chains” of a K3 Alphabet recovery.

Consider the following partially recovered K3 alphabet:

```
Y--EUR-AB-TFGHZ--KLM--X-V-
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Notice that we can, trivially, construct the following “chains” to start by noticing which pairings had both letters. That would produce this:

```
YA  ED  UE  RF  AH  BI  TK  FL  GM  HN  ZO  KR  LS  MT  XW  VY
```

Now, in a K3 alphabet, if we have enough material, we could assemble these (eventually) into a single chain yielding a full 26 letter alphabet. That is, there will be one chain with 26 letters instead of just fragments. With a full 26, we can “decimate” it (explained below) to get all the others and eventually, the keyword would be revealed. However, here, at least J and Q are missing from both the CT and PT. Still, let’s start with the longest available chains. It’s an important prelude to K3 recovery.

```
# K3 'longest chains' app
CTA = "Y--EUR-AB-TFGHZ--KLM--X-V-"
PTA = "abcdefghijklmnopqrstuvwxyz".upper()chains= []
# "def" means "start a function or subroutine"
def joinChains() :
    global chains  # chains is defined outside; we use, modify it here
    for i in range(0,len(chains)) :
        for j in range(i+1,len(chains)) : # note where j starts.
            lastI = len(chains[i])-1
            lastJ = len(chains[j])-1
            if(chains[i][0]==chains[j][lastJ]) :
                temp = chains[i]
                chains[j]=chains[j]+temp
                del chains[i]  # The Python "delete me from array"
                return False # the'chains'array changes size, return
            if(chains[j][0]==chains[i][lastI]) :
                temp = chains[j]
                chains[i]=chains[i]+temp
                del chains[j]  # The Python "delete me from array"
                return False # new size, so return
            # continue if we get this far
    return True  # all chains combined, no changes this run.
# “Main” program starts here.
for i in range(0,len(CTA)) :
    if(CTA[i]!='-') :
        chn = CTA[i]+PTA[i]
        chains.append(chn)
print("Initial chains "+str(chains))
res = False
while (res==False) :  # when False, "chains" was altered, start over.
    res=joinChains()
    print("intermed chains "+str(chains))
print("Final chains "+str(chains))
# OK, our coding leads to doubled letters. Clean up, get rid of them.
for i in range(0,len(chains)) :
    wrd = chains[i][0] # preserve first letter.
    for j in range(1,len(chains[i])) :
        if(chains[i][j]!=chains[i][j-1]) : # if not doubled
            wrd=wrd+chains[i][j]  # add it into wrd
    chains[i] = wrd   # replace chains[i] with cleaned up version
wrd=""
if(len(chains)==1 and len(chains[0])==27) : # very special case
    for i in range(0,26) :   # clean up single 26 letter chain.
        wrd=wrd+chains[0][i]
    chains[0]=wrd
print("Clean chains "+str(chains))
```

An execution looks like this:

```
Initial chains ['YA', 'ED', 'UE', 'RF', 'AH', 'BI', 'TK', 'FL', 'GM', 'HN', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VY']
intermed chains ['YAAH', 'ED', 'UE', 'RF', 'BI', 'TK', 'FL', 'GM', 'HN', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VY']
intermed chains ['YAAHHN', 'ED', 'UE', 'RF', 'BI', 'TK', 'FL', 'GM', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VY']
intermed chains ['ED', 'UE', 'RF', 'BI', 'TK', 'FL', 'GM', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'RF', 'BI', 'TK', 'FL', 'GM', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'RFFL', 'BI', 'TK', 'GM', 'ZO', 'KR', 'LS', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'TK', 'GM', 'ZO', 'KRRFFL', 'LS', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'TKKRRFFL', 'GM', 'ZO', 'LS', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'TKKRRFFLLS', 'GM', 'ZO', 'MT', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'GM', 'ZO', 'MTTKKRRFFLLS', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'GMMTTKKRRFFLLS', 'ZO', 'XW', 'VYYAAHHN']
intermed chains ['UEED', 'BI', 'GMMTTKKRRFFLLS', 'ZO', 'XW', 'VYYAAHHN']
Final chains ['UEED', 'BI', 'GMMTTKKRRFFLLS', 'ZO', 'XW', 'VYYAAHHN']
Clean chains ['UED', 'BI', 'GMTKRFLS', 'ZO', 'XW', 'VYAHN']
```

With the intermediate chains being printed, you can see the combinations happening. The coding style has one modest problem – the letters in the interior of joined chains ends up with "doubled" letters. But that's easily cleaned up at the end ("Clean chains").

For this example, the rest is, for now, by hand. Perhaps reconstruction of the full alphabet could be automated, but it is an intermediate to advanced problem. For now, given the above chains, ordinary manual methods can reconstruct the full 26 letter alphabet. See *The Cryptogram* archives for how.

Decimation

We've talked a lot about "decimation". Decimation actually motivates the whole issue of chaining to start with. In the above example, we had only a partial alphabet. If we had all 26 letters, it would form a single chain. We could "decimate" it and produce all the shifts for a given K3 keyword. One of these shifts would reveal the keyword and validate the recovery. The reason is a peculiarity of the K3 keyword scheme. For a given keyword alphabet, there are twenty six possible ways to shift the alphabet against itself. The odd shifts have all twenty six letters and you can use decimation to get all of the others.

Decimation makes a nice example (see Problem 12). There are three cases: The even shifts, the odd shifts and a shift of 13. All require a little different handling.

Suppose we had a complete alphabet of 26:
CTA = "YWOEURDABCTFGHZIJKLMNPXQVS"

. . .then re-running the chaining program above produces this single chain:
SZOCJQXWBIPVYAHNUEDGMTKRFL

The game now turns to "decimation" because with the right shift (that is, the right decimation) the keyword simply appears. Problem 12 includes some of the decimations as a clue for debug. The one with the answer is of course the one of interest, but we typically must produce all decimations to find it.

Summary

At this point, the reader has been given a short survey of Python programming and program templates. Note that many of the Python specific ideas here are not unique to Python – other languages have many of these concepts under their own name or syntax, so most of this applies whether one chooses to use Python or not. Note that most languages that allow deleting from an array actually give it a different name – a "list" object – and this must be used instead of an array. In Python, lists and arrays are actually the same thing.

As this point, the beginner, perhaps coupled with an on-line tutorial, can begin to write code to solve various problems. In fact, here are a few that should be manageable now:

Problem 8. Take a frequency count of diagrams using a two-dimensional array. Use easy texts like "ABABCDCDEFEQEQ" for a test bed.

Problem 9. Write a program to encipher a checkerboard cipher (the kind with a single keyword for the row and column indices). Don't forget to translate all plaintext "J" to "I" first. Example:

```
SD SG SE SU BD BG RG SE SG EE OJ OJ OG BG BG BE OG RU BU EU SE EE
q  u  i  c  k  l  y  j  u  s  t  t  e  l  l  m  e  w  h  o  i  s
SE EJ EJ EU SU OG EJ OJ
i  n  n  o  c  e  n  t

  JUDGE
-------
S|ACQUI
O|TBDEF      <== keysquare
B|GHKLM
E|NOPRS
R|VWXYZ
```

Problem 10. Write a program to encipher a Playfair cipher. This includes a "pre-pass" to detect doubled letters on an even boundary and place the letter X between such letters, accounting for such additions as you look for more doubled letters. Do not add the X to doubled letters on odd boundaries of the original plaintext as they are in different enciphering pairs. Your final result after the pre-pass is a new plaintext variable that will usually be slightly longer than the original. Example:

```
UI AQ LM ZU IR DV BF KY MG BY OW FZ AS OP UB VG
qu ic kl yi us tx te lx lm ew ho is in no ce nt

ACQUI
TBDEF
GHKLM        <== keysquare
NOPRS
VWXYZ
```

Problem 11. Write a program to encipher a Foursquare cipher. See the *ACA & You* or https://en.wikipedia.org/wiki/Four-square_cipher for examples. Can you think of a clever trick to do decipherment with your encipherment program?

Problem 12. Write the "decimation" program. Start with this: SZOCJQXWBIPVYAHNUEDGMTKRFL . . .which is also "decimation 1". Decimation 2 is: SOJXBPYHUDMKFZCQWIVANEGTRL . . .and requires the code that has two chains of 13. Can you see how it works from the answer? Decimation 21 is: STUVXZKEYWORDABCFGHIJLMNPQ . . . and requires the code that deals with a single chain of 26. Do you see how it works? Look particularly at where T, U, V, X, and Z are in the original SZOCJ… alphabet. It also contains the actual keyword which was going to "pop out" of one of the decimations. Decimation 3, SCXIYNDTFZJWPAUGKLOQBVHEMR, might be even more revealing about odd decimations.

Bibliography

There are opportunities to learn Python on the web. The author has used this site before: https://www.w3schools.com/python/

General Cryptography can be learned on the ACA's website: https://www.cryptogram.org There are some facilities for anyone, but a lot of the site requires a membership and a password. Particularly recommended are "The ACA and You", the "Practical Cryptanalysis" series (several volumes) and The Cryptogram Archive and its index. "Novice Notes" was a long running series describing most ACA ciphers.

There are brief public descriptions of the ACA cipher types here: https://www.cryptogram.org/resource-area/cipher-types/