BOSS Technical Note 001

PROGRAMMING FOR CRYPTOLOGISTS

Programming for Cryptologists

September 2021

Foreword

This guide has been produced as an update to the original Programming for Cryptologists published in 2011, in celebration of the 20th anniversary of the National Cipher Challenge. Whilst the previous guide covered multiple programming languages, this updated version focuses solely on Python: a language which has experienced a meteoric rise in popularity over the last decade and is now commonplace across STEM education.

There are many brilliant free resources available for learning Python and this guide does not seek to replicate them, but rather provide some tips and tricks specific to cryptanalysis.

Context

Cryptology and computer science have a shared history. Ada Lovelace and Charles Babbage's work on the Difference Engine and Analytical Engine laid the foundations for general purpose computing: machines that could be programmed to solve many different problems. Lovelace and Babbage were ahead of their time, though, and government funding for their machine was sadly cut. It was not until World War II when cryptanalysis accelerated computer science forward. The need for computation machines to attack the Enigma led to the creation of Bombes: originally a Polish invention, these were the electro-mechincal devices which Alan Turing and his colleagues used to successfully decipher Enigma messages. Turing is most famously associated with his wartime cryptanalytic efforts which are often credited as having shortened the war by several years and saved millions of lives. This is far from the limit of Turing's legacy, though, and his work on theoretical computer science to this date defines the field as we know it. Cryptology continues to define modern computing, with the ubiquity of Internet technology only being possible due to the strong encryption that enables it. Communications security is a lot more complex these days, where threats to cyber security manifest in myriad forms other than weak encryption, but the tight coupling of cryptology and computer science is set to remain as quantum computing and quantum cryptography looks to define the next era.

Developing a literacy in computer programming is the underpinning to a good cryptanalyst, and for the budding computer scientist, algorithmic cryptanalysis is an effective way to learn.

Getting Started with Python

There are many tutorials available online for getting started with Python. Some of these might refer to an older version of Python, namely Python 2 which is no longer officially supported as of 1st January 2020 (twenty years after its initial release). Python 3 is the current version (at the time of writing!) which has been available since 2006 and is the best place to start.

There are a few different ways to write and run Python code. You can write code on your computer using a text editor and execute it on the 'command line' having already installed

Python, or a purpose-built Integrated Development Environment (IDE) which you can also download and install. You can also run Python interactively, where you can execute code line-by-line and see the results one step at a time. Alternatively, many websites will let you write and execute code in your web browser without the need to install anything on your computer. Growing in popularity is use of Jupyter Notebooks which allow you to interactively write and execute code - this guide is an example of a Jupyter Notebook itself! Notebooks can be a lot easier to use than learning how to execute Python on the command line, and are a really effective way to tackle ciphers. You can try out Jupyter online without needing to install anything, or download and install it alongside Python and a whole load of useful data science packages bundled together in Anaconda.

This guide provides a few code snippets and examples to get you started with using Python for cryptanalysis.

String Manipulation

Python allows you to define 'string literals' (i.e. some text) and iterate over the characters that make up that string:

In [1]:
```
message = "Attack at dawn"
for letter in message:
    print(letter)
```

```
A
t
t
a
c
k

a
t

d
a
w
n
```

Python also lets you to define 'byte strings' - these behaviour a little differently and allow us to access the underlying representation of the letters in the computer's memory. At the lowest level, computers only store numbers (ones and zeros, known as 'bits') therefore any other type of data (text, images, videos) must be 'encoded', in other words, represented as a string of ones and zeros (a 'byte' is 8 bits, for example 01000001). In Python, we can define a byte string by adding the letter 'b' before the string quotes:

In [2]:
```
message = b"Attack at dawn"
for letter in message:
    print(letter)
```

```
65
116
116
97
99
107
32
97
```

```
116
32
100
97
119
110
```

This time, when we iterate over the string instead of getting single characters, we receive a number that represents that letter. The mapping between letters and numbers is defined by the encoding scheme, which by default is ASCII. There are many ASCII tables that show this mapping available online. For instance, uppercase 'A' is represented by the decimal 65 (not to be confused with the hexadecimal representation, 0x41) and lowercase 'a' is represented by the decimal 97 (hexadecimal 0x61).

Working with byte strings is convenient to us as cryptanalysts because we'll often want to use a numerical representation of the letters. If we were using normal strings, we would need to use the ord() function to convert between a letter and its ASCII representation. Ordinarily, we are going to want letters to be mapped to numbers in the range 0 to 25, though. There's a simple trick here: we can just minus 65 from every letter and this maps 'A...Z' to '0...25' instead.

There is an alternative to using byte strings. If you find it easier, you can look up the index of each letter within the alphabet, like so:

In [3]:
```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
message = "ATTACKATDAWN"
for letter in message:
    print(alphabet.index(letter))
```

```
0
19
19
0
2
10
0
19
3
0
22
13
```

There are often many different ways to achieve the same result in programming, and Python offers many shortcuts if you choose to use them. These can often speed up writing code, though often at the cost of the 'readability' of your code (how easy it is for someone else to understand your code).

For instance, there is a very powerful library that can be installed with Python named **numpy**. One of the nice features of **numpy** is that it provides a data type called an array which comes with many useful features. One such feature is the ability to apply an operation to every element of an array at the same time, for instance:

In [4]:
```
# Import the numpy library
import numpy as np
# Convert to uppercase and strip spaces.
# It is easiest to work with strings with every character
# as upper or lower case when converting from ASCII.
message = b"Attack at dawn".upper().replace(b' ',b'')
# Create a list from the bytes string, convert it to a numpy array,
```

```
# then minus 65 from each item
msg_array = np.array(list(message)) - 65 # 'A' = 65
print(msg_array)
```

```
[ 0 19 19  0  2 10  0 19  3  0 22 13]
```

Converting back and forth between letters and numbers will come in handy, so we can define these as functions that we can call:

In [5]:
```
def encode(message):
    msg_array = np.array(list(message)) - 65
    return msg_array

def decode(msg_array):
    message = bytes(list(msg_array + 65))
    return message
```

You can condense this into a one-line function definition using a feature of Python called 'lambdas':

In [6]:
```
encode = lambda message: np.array(list(message)) - 65
decode = lambda msg_array: bytes(list(msg_array + 65))
```

You can then implement a Caesar Shift cipher with ease:

In [7]:
```
caesar_encrypt = lambda ciphertext, key: decode((encode(ciphertext)+key)%26)
caesar_decrypt = lambda ciphertext, key: decode((encode(ciphertext)-key)%26)
print(caesar_encrypt(b"ATTACKATDAWN", 12))
print(caesar_decrypt(b'MFFMOWMFPMIZ', 12))
```

```
b'MFFMOWMFPMIZ'
b'ATTACKATDAWN'
```

The percentage symbol here is used as modulus, i.e. to wrap a number between 0 and 25.

Transposition

Implementing transposition ciphers in Python can be a little tricky. One trick that can help is to make use of the transpose function that the **numpy** library provides for 2D arrays.

Consider the following message that has been encrypted using a columnar transposition cipher with a key of '2, 0, 1':

In [8]:
```
ct = b"TKDNAAAATCTW"
```

We can first populate this into a 3 by 4 array:

In [9]:
```
M = encode(ct).reshape(3,4)
print(M)
```

```
[[19 10  3 13]
 [ 0  0  0  0]
 [19  2 19 22]]
```

Then transpose the array and rearrange the columns:

In [10]:
```
# This notation looks scary!
```

```
# We are selecting [rows, columns], and saying we want all the rows (':')
# And columns 1, 2 then 0, in that order.
Mt = M.T[:,[1,2,0]] # Using inverse of the key for decryption
print(Mt)
```

```
[[ 0 19 19]
 [ 0  2 10]
 [ 0 19  3]
 [ 0 22 13]]
```

And flatten the array before decoding:

In [11]:
```
decode(Mt.flatten())
```

Out[11]:
```
b'ATTACKATDAWN'
```

Language Modelling

Frequency analysis lies at the heart of most cryptanalysis. Being able to count the occurance of letters (or combinations of letters) in a ciphertext is very useful, and luckily Python makes this easy for us:

In [12]:
```
from collections import Counter

ct = """ah ae psh hxo ibahai qxs iskphe; psh hxo mcp qxs vsaphe skh xsq hxo ehbspu m
ehkmfjoe, sb qxobo hxo lsob sr loole iskjl xcno lspo hxom fohhob. hxo ibolah fojspue
hs hxo mcp qxs ae cihkcjjw ap hxo cbopc, qxseo rcio ae mcbbol fw lkeh cpl eqoch cpl
fjssl; qxs ehbanoe ncjacphjw; qxs obbe, qxs ismoe exsbh cucap cpl cucap, foickeo hxo
bo ae ps orrsbh qahxskh obbsb cpl exsbhismapu; fkh qxs lsoe cihkcjjw ehbano hs ls hx
o loole; qxs gpsqe uboch ophxkeaceme, hxo uboch lonshaspe; qxs evople xameojr ap c q
sbhxw ickeo; qxs ch hxo foeh gpsqe ap hxo opl hxo hbakmvx sr xaux cixaonomoph, cpl q
xs ch hxo qsbeh, ar xo rcaje, ch joceh rcaje qxajo lcbapu ubochjw, es hxch xae vjcio
excjj ponob fo qahx hxseo isjl cpl hamal eskje qxs poahxob gpsq naihsbw psb loroch."
ct = ct.replace('\n','')

#This automatically converts the string into a list,
# and counts the frequency of each character.
letter_freq = Counter(ct)
letter_freq.most_common()
```

Out[12]:
```
[(' ', 139),
 ('o', 69),
 ('h', 64),
 ('s', 54),
 ('x', 49),
 ('e', 45),
 ('c', 44),
 ('p', 37),
 ('a', 36),
 ('b', 32),
 ('l', 26),
 ('q', 24),
 ('j', 21),
 ('i', 16),
 ('k', 14),
 ('m', 13),
 ('u', 10),
 ('r', 10),
 ('f', 9),
 (',', 9),
 ('n', 8),
 (';', 7),
```

```
('w', 7),
('v', 4),
('g', 3),
('.', 2)]
```

Converting these into percentages requires a little manipulation. To compare against expected letter frequencies for English, it'll be helpful to strip non-alphabet characters. We can do this using a 'regular expression':

In [13]:
```
import re
ct2 = re.sub(r"[^a-z]", "", ct)
print(ct2)
letter_freq = Counter(ct2)
[(letter, 100*count/len(ct)) for letter, count in letter_freq.most_common()]
```

```
ahaepshhxoibahaiqxsiskphepshhxomcpqxsvsapheskhxsqhxoehbspumcpehkmfjoesbqxobohxolsobs
rlooleiskjlxcnolspohxomfohhobhxoibolahfojspuehshxomcpqxsaecihkcjjwaphxocbopcqxseorci
oaemcbbolfwlkehcpleqochcplfjsslqxsehbanoencjacphjwqxsobbeqxsismoeexsbhcucapcplcucapf
oickeohxoboaepsorrsbhqahxskhobbsbcplexsbhismapufkhqxslsoecihkcjjwehbanohslshxolooleq
xsgpsqeubochophxkeacemehxoubochlonshaspeqxsevoplexameojrapcqsbhxwickeoqxschhxofoehgp
sqeaphxooplhxohbakmvxsrxauxcixaonomophcplqxschhxoqsbeharxorcajechjocehrcajeqxajolcba
puubochjweshxchxaevjcioexcjjponobfoqahxhxseoisjlcplhamaleskjeqxspoahxobgpsqnaihsbwps
bloroch
```

Out[13]:
```
[('o', 9.175531914893616),
 ('h', 8.51063829787234),
 ('s', 7.180851063829787),
 ('x', 6.51595744680851),
 ('e', 5.98404255319149),
 ('c', 5.851063829787234),
 ('p', 4.920212765957447),
 ('a', 4.787234042553192),
 ('b', 4.25531914893617),
 ('l', 3.4574468085106385),
 ('q', 3.1914893617021276),
 ('j', 2.7925531914893615),
 ('i', 2.127659574468085),
 ('k', 1.8617021276595744),
 ('m', 1.7287234042553192),
 ('u', 1.3297872340425532),
 ('r', 1.3297872340425532),
 ('f', 1.196808510638298),
 ('n', 1.0638297872340425),
 ('w', 0.9308510638297872),
 ('v', 0.5319148936170213),
 ('g', 0.39893617021276595)]
```

Visually plotting the letter frequency distribution compared to what you would expect for English can sometimes be useful:

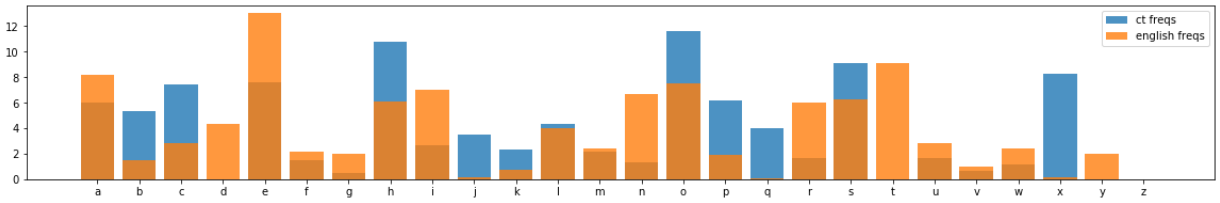
In [14]:
```
%pylab inline
import pylab as plt
plt.rcParams["figure.figsize"] = (20,3)

alpha = 'abcdefghijklmnopqrstuvwxyz'
engfreq = [8.2,1.5,2.8,4.3,13,2.2,2,6.1,7,0.15,0.77,4,
           2.4,6.7,7.5,1.9,0.095,6,6.3,9.1,2.8,0.98,2.4,0.15,2,0]

plt.xticks(range(len(alpha)), list(alpha))
plt.bar(range(len(alpha)),[100*letter_freq[a]/len(ct2) for a in alpha],
        alpha=0.8, label='ct freqs')
plt.bar(range(len(alpha)), engfreq, alpha=0.8, label='english freqs')
plt.legend()
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[14]: <matplotlib.legend.Legend at 0x7f3548313e10>



For a monoalphabetic substitution cipher, single letter frequency analysis can help you recover the alphabet mapping. Once you have the correct mapping, there are a few different ways you can apply it, for example:

In [15]:
```
alpha = 'abcdefghijklmnopqrstuvwxyz'
key = 'cfiloruxadgjmpsvybehknqtwz'
pt = ""
for letter in ct:
    if letter in alpha:
        pt += alpha[key.index(letter)]
    else:
        pt += letter
print(pt)
```

```
it is not the critic who counts; not the man who points out how the strong man stumb
les, or where the doer of deeds could have done them better. the credit belongs to t
he man who is actually in the arena, whose face is marred by dust and sweat and bloo
d; who strives valiantly; who errs, who comes short again and again, because there i
s no effort without error and shortcoming; but who does actually strive to do the de
eds; who knows great enthusiasms, the great devotions; who spends himself in a worth
y cause; who at the best knows in the end the triumph of high achievement, and who a
t the worst, if he fails, at least fails while daring greatly, so that his place sha
ll never be with those cold and timid souls who neither know victory nor defeat.
```

Alternatively, Python provides a string 'translate' function that lets you map between the ciphertext and plaintext alphabets directly:

In [16]:
```
table = str.maketrans('cfiloruxadgjmpsvybehknqtwz',
                      'abcdefghijklmnopqrstuvwxyz')
ct.translate(table)
```

Out[16]:
```
'it is not the critic who counts; not the man who points out how the strong man stum
bles, or where the doer of deeds could have done them better. the credit belongs to
the man who is actually in the arena, whose face is marred by dust and sweat and blo
od; who strives valiantly; who errs, who comes short again and again, because there
is no effort without error and shortcoming; but who does actually strive to do the d
eeds; who knows great enthusiasms, the great devotions; who spends himself in a wort
hy cause; who at the best knows in the end the triumph of high achievement, and who
at the worst, if he fails, at least fails while daring greatly, so that his place sh
all never be with those cold and timid souls who neither know victory nor defeat.'
```

Calculating a score for how much like English (or another language) a putative plaintext is will be useful when it comes to attacks that include an element of trial and error. For this, single letter frequencies are often not enough (and certainly no help with transposition ciphers!) so you might want to consider scoring based on a frequency of n-grams, i.e. groups of 2, 3 or 4 letters.

How you combine frequencies to calculate a score for 'how English' a putative plaintext looks is left as an exercise to the reader, though it is recommended that you work with the logarithms of the frequencies. You'll find many advanced techniques for language modelling out there, including machine learning methods such as neural networks, but often simple methods are highly effective.

Algorithms

For simple ciphers, we can use a 'brute-force' approach: we can exhaust over the keyspace (all possible keys) until we find the right one. This assumes we know how to recognise the 'right' key when we find it. For a smaller enough keyspace (like a Casaer Shift where there are only 26 possible keys), we could manually inspect each 'putative' plaintext until we spot the right one. As the keyspace grows, we will need to use one of the approaches to language modelling described above to score putative plaintexts and identity the right one.

The brute-force approach very quickly becomes infeasible. You do not even need something as complex as the Engima machine to start to struggle, even with modern computing power. A simple monoalphabetic substitution cipher has a keyspace of 26 factorial (26!) which is roughly 403 septillion (million billion billion), or in modern computing terms, that's roughly equivalant to a 88-bit key.

Many traditional ciphers, however, have a weakness in that they do not provide sufficient 'confusion' between the ciphertext and the key. This means attempting to decrypt a ciphertext with a key that is 'close' to the right one but not exactly right will result in a plaintext that is partially right. We can exploit this to conduct a smarter search of the keyspace by scoring each putative plaintext and attempting to adjust the key to get a plaintext that scores higher. There are a whole family of different algorithms that do this using varying methods, but this approach is generally referred to as Hill climbing. This is a powerful technique for cracking many ciphers, therefore the implementation of such an algorithm is left as an exercise to the reader!