Workshop 16: Ciphers

Written by Jack Kelly

Learn how classical ciphers worked, and how to write Python programs to encode, decode and crack them!

Difficulty: Introductory workshop.

Contents

0 Introduction

Last month in workshop 15, we *encoded* a message in Morse code, a method that makes it simpler to transmit written language through rudimentary means, such as a spotlight, or a pulse in an electrical wire.

We also often use the word *code* to refer to an encrypted **cipher**. A different type of encoding, when we encode a message in a cipher, we attempt to make the message difficult—or ideally impossible— for an outsider to read.

Today, we'll have a go at encoding, decoding and cracking some famous ciphers of the classical era using Python programs.

This is an introductory workshop, all of the Python concepts will be covered from scratch.

How to use these booklets

The aim of these booklets is to help you attempt these workshops at home, and to explain concepts in more detail than at the workshop. You don't need to refer to use these booklets during the workshop, but you can if you'd like to.

When you need to make changes to your code, they'll be presented in *listings* like the example below. Some lines may be repeated across multiple listings, so check the line numbers to make sure you're not copying something twice.

```
1 import string
2
3 #create a list containing every letter of the alphabet
4 list(string.ascii_lowercase)
```

Occasionally, a concept will be explained in greater detail in *asides*, like the one below. You can read these as you wish, but they're not required to complete the workshop.

Cryptography

We call the creation and study of ciphers *cryptography*. 'Crypto-' comes from the greek word *kruptos*, meaning 'hidden'.

What you'll need

All of the software you for this workshop is pre-installed on recent versions of Raspbian.

Terminology

Some of the words used to describe programming and cryptography often cross, so the following words will be used in this booklet.

Program	Code written in Python
Cipher	A cryptographic code
Plaintext	A phrase that *isn't* encrypted
Ciphertext	A phrase that *is* encrypted

This should avoid confusing sentences, like the fact that last month we were coding programs to encode Morse code, this month we're coding cipher codes, a different type of encoding.

Everything else

These booklets were created using LaTeX, an advanced typesetting system used for several sorts of books, academic reports and letters.

If you'd like to have a look at using LaTeX, We recommend looking at TeXstudio, which is available on most platforms, and also in the Raspbian repository.

 Latex source documents are available at http://github.com/McrRaspJam/booklet-workshops

If you get stuck, find errors or have feedback about these booklets, please email me at: jam@jackjkelly.com

1 Python Basics

To be able to write programs to encode ciphers, we'll need to be able to pull letters and words out of the plaintext.

When we store text in a Python variable, it is called a `string`. These first few Python programs will teach us everything we need to manipulate strings.

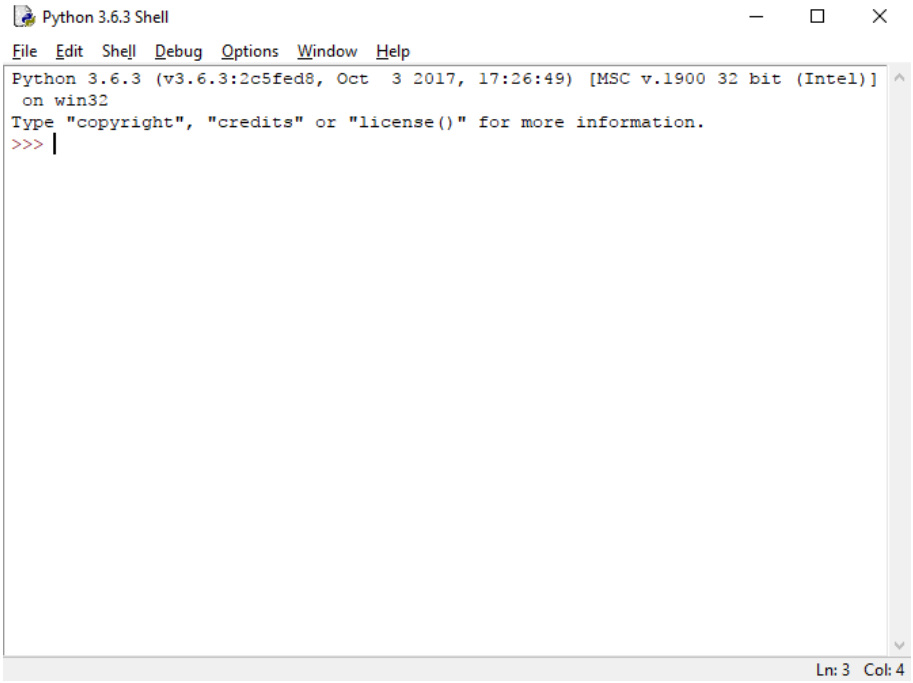

Figure 1: The IDLE *shell* window

Go to the main menu on your Raspberry Pi desktop and **open** the application titled *Python 3 (IDLE)*.

After a few seconds, IDLE will open to the *Python Shell*. In this window, go to `File` → `New File`, which will open an empty window where we can write our programs.

Hello, World!

In your empty window, **write** the following program:

helloworld.py

```
1 print("Hello, World!")
```

then, **run** the program by selecting `Run` → `Run Module` at the top of the window.

Inputting text

We need to be able to type in text at the start of our programs, which we can do with the `input()` function. **Modify** your program as shown, then **run** it.

```
1 text = input("Enter Text: ")
2 print(text)
```

Variables

When we store data (like text or numbers) in our programs, we store them as *variables*.

A variable has a *label*—what the variable is called—and a *value*—the data we wish to store in it.

When we 'set' a variable, we call this an *assignment*, signified by the = sign.

Lists

It is often useful to store several pieces of data in a single variable. A common way of doing this in Python is to use a list.

Open a **new** file then **write** the following program in the new window:

list.py

```
1 icecream = ["Vanilla", "Strawberry", "Chocolate", "Raspberry Ripple
    "]
2 print (icecream[0])
3 print (icecream[3] + icecream[0] + icecream[1])
```

Lists

When we add *elements* of data to a list, they are given the next available *index* number.

0	Vanilla
1	Strawberry
2	Chocolate
3	Raspberry Ripple

To access a single element from a list, we use square brackets, as we have done in our program above.

Strings are Lists

In python, a text `string` is actually a list of single `characters`. This makes it easy to grab single letters from a plaintext.

Return to your hello world program, and **modify** the print statement as follows.

helloworld.py

```
1 text = input("Enter Text: ")
2 print(text[0])
```

Now when you run your program, only the first letter of whatever you typed in will be printed out. Next, we'll try printing out each letter one-by-one, which we can do using a *loop*:

helloworld.py

```
1 text = input("Enter Text: ")
2
3 for letter in text:
4     print(letter)
```

The `for` loop we used repeats once for each element in the list that we provide it, in this case each letter in our string.

Run your program once more, and this time your text should be printed one character at a time, each on its own line.

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext	z	y	x	w	v	u	t	s	r	q	p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a

Figure 2: Atbash substitution for the Latin alphabet

2 Substitution Ciphers

Ciphers have existed in some form since the classical era. One of the earliest examples of ciphers being used to keep information are clay tablets from Mesopotamia—dating back to 1500 BCE—which were used to keep trade secrets like the recipe for pottery glaze.

Origins of Ciphers

From Wikipedia:

"The earliest known use of cryptography is found in non-standard hieroglyphs carved into the wall of a tomb from the Old Kingdom of Egypt circa 1900 BCE.

"These are not thought to be serious attempts at secret communications, however, but rather to have been attempts at mystery, intrigue, or even amusement for literate onlookers."

2.1 Atbash

Atbash was a simple substitution cipher that emerged around 500 BCE, and was originally used to encode the Hebrew alphabet. (the name is a shortening of the letters Aleph-Tav-Beth-Shin)

In a substitution cipher, the characters of the plain-text are *substituted* with another set of characters—usually the same alphabet in a different order. For Atbash, this is very simple, the alphabet is reversed.

An Example

The tradition when practising cryptography is to use faux military commands, so we'll start with the plaintext "**Attack at dawn**".

using the table in Figure 2, we find each letter in the first row, and replace it with the letter in the following row.

A → Z
t → g
t → g
a → z
c → x
k → p

Our ciphertext for this phrase would be "**Zggzxp zg wzdm**".

Now, using a pen and paper and Figure 2, have a go at encoding the following plaintext

using Atbash.

"**We are discovered, flee at once**"

Atbash in Python

Now we know how Atbash works, we can try writing a Python program that encodes it.

When run in a terminal, the program should look like the following:

```
$ python3 atbash.py
Enter plaintext: attackatdawn
zggzxpzgwzdm
```

Open a new script file in IDLE by going to `File` → `New File`.

The first thing our program needs to do when run is get the plaintext. We'll use an `input()` function, just like we did in section 1.

atbash.py

```
1 plaintext = input("Enter plaintext: ")
```

We'll also set an empty `string` variable, where we'll add our ciphertext.

```
2 ciphertext = ""
```

The other thing we need is a copy of the table in Figure 2, which we'll place in lists.

We could do this by manually defining the list:

```
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
    'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
     'y', 'z']
```

but let's save some time and take advantage of one of Python's libraries to automate the generation of this list:

```
1 import string
2
3 plaintext = input("Enter plaintext: ")
4 ciphertext = ""
5
6 alphabet = list(string.ascii_lowercase)
```

Then we'll create the substitute alphabet by copying the list in reverse:

```
7 subalphabet = list(reversed(alphabet))
```

You can check these lists are correct by running the program, then typing `alphabet` and `subalphabet` into the shell window prompt, which will print their contents.

We have everything we need, we can now perform the encryption. Just like on pen and paper, we convert each letter one-by-one, so we'll use the loop we learned in section 1.

```
8
9  for letter in plaintext:
```

At this point we have one more problem to consider. To pull a letter from the alphabet list, we need to access it using an index number. (In `alphabet` 0=a, 1=b, 2=c, etc.)

We need a way to number letters based on their position in the alphabet. Fortunately, the ASCII encoding scheme already assigns a number to each letter, and because it is widely used it is very easy to get the code for a letter in Python.

ASCII

ASCII (*American Standard Code for Information Interchange*) was invented as a way to standardise how letters were stored on different file formats and computer systems, starting in the early 1960's.

ASCII encodes a small number of characters, mostly, alphanumeric characters, punctuation and control characters (things like 'new line', 'end of file' or 'backspace'.)

These days, most text is actually encoded in the much larger Unicode standard.

To get a number for each letter in our loop:

```
9  for letter in plaintext:
10     charnumber = ord(letter)
```

However, the ASCII number for "a" is '97', "b" is '98' etc. so we'll need to subtract 97 to match the 0–25 range of our alphabet list

```
10     charnumber = ord(letter) - 97
```

Now, we can simply take the character number, and access the substitute alphabet list using the same index.

```
11     subalphabet[charnumber]
```

Remember that we're in a loop, so we just need to add the current cipher letter to the ciphertext.

```
11     ciphertext += subalphabet[charnumber]
```

Once the loop has completed, our ciphertext should be complete! We simply need to add a print statement outside of the loop.

```
11     ciphertext += subalphabet[charnumber]
12
13 print (ciphertext)
```

Try testing the program using the two plaintexts from the pen and paper exercise and see if the output from your program matches.

Bear in mind your program currently has some limitations:

- Capital letters won't work, as they have different ASCII codes to lowercase letters.
- We haven't told the program what to do with spaces, so we can't use them either.

If you wanted to extend your program, fixing these would be a good place to start. You should also have a think about the following:

- Our program currently *encodes* Atbash ciphers. What would we have to do to write a program that *decodes* the cipher?
- How secure do you think Atbash is?

2.2 Caesar Cipher

The Caesar cipher is another substitution cipher named after the Julius Caesar, who used it to encrypt messages of military significance.

This cipher is similar to Atbash, but instead of reversing the alphabet, the substitute alphabet is created by shifting letters.

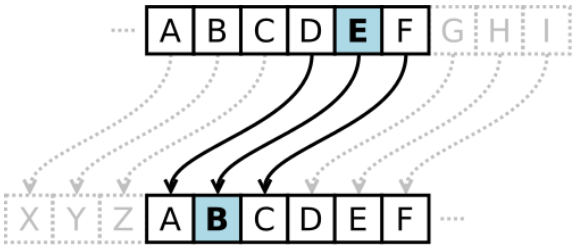

Figure 3: Shifting letters to create the substitute alphabet

An Example

If we wish to encrypt the plaintext "**The enemy surrounds us**" with a rightward shift of **1** place, we can use the table in Figure 4 as follows:

T	→	u	e	→	f
h	→	i	n	→	o
e	→	f	e	→	f
			m	→	n
			y	→	z

Our full ciphertext for this phrase would be "**Uif fofnz tvsspvoet vt**".

Returning to your pen and paper, have a go at encoding the following phrase.

"**Attack from the west**", right shift of **3**.

Then try the following. You will need to construct your own substitute alphabet for this:

"**The city has fallen**", *left* shift of **2**.

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Right 1	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a
Right 2	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b
Right 3	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Figure 4: Caesar shift substitutions for the first three values of right shift

Caesar Cipher in Python

Because the Caesar cipher is so similar to Atbash, we can reuse most of the code from that program.

Copy the Atbash program—either through the file manager, or by `File` → `Save As...`—and rename, then open the new file.

The first addition we will add to our program will be to allow the user to determine the shift for the cipher. We can do this with another `input()` function.

caesar.py

```
3  # input plaintext
4  plaintext = input("Enter plaintext: ")
5  shift = int(input("Enter shift (positive=right): "))
6  ciphertext = ""
```

Then, we need to create the substitute alphabet based on the shift number. This is more complex than Atbash, so we'll start with an empty list, and fill it using a loop.

```
7
8  alphabet = list(string.ascii_lowercase)
9  subalphabet = []
```

To shift the letters, we'll first get the ASCII code for the letter and subtract 97, just like we did last time:

```
10
11 # generate substitute alphabet
12 for letter in alphabet:
13         charnum = ord(letter) - 97
```

then we'll add the shift number. To wrap from one end of the alphabet to the other, we'll use a *modulus* operator.

```
14         charnum = (charnum + shift) % 26
```

Modulus Operation

The modulus operation (signified by %) finds the remainder of a division.

For the letter 'x' (23)

$$23 \div 26 = 0 \text{ remainder } \mathbf{23}$$

When shifted by 5, it 'wraps around' to the start of the alphabet :

$$28 \div 26 = 1 \text{ remainder } \mathbf{2}$$

Finally, we grab the letter at this place in the alphabet, and add that to our substitute alphabet

```
15          subalphabet.append(alphabet[charnum])
```

Your program should now be complete. Try encoding the phrases from the pen and paper examples, and see if they match.

Have a think about the same questions we asked after we finished the Atbash program.

- Our program currently *encodes* Caesar ciphers. What would we have to do to write a decoding program?
- What makes the Caesar cipher more secure than Atbash?

Cryptographic Key

When we introduce a variable into an encryption method, we call this a cryptographic **key**. (sometimes called a *cryptovariable*)

In Atbash, there is only one way that the alphabet is arranged, so there is no key. In the Caesar cipher, the cipher can be varied each time the cipher is used, so the shift number is the key for this cipher.

By introducing a key, an encrypted message can stay secure even if an intercepting party knows what cipher was used. However, the key *must* be kept secret to maintain the security of the ciphertext.

2.3 Cracking the Caesar Cipher

If we don't hold the key to a ciphertext, we'll need to crack the code to access the plaintext.

Because of the amazing computing speed of our Raspberry Pi, the easiest way to do this is to try every possible key in turn until we find the plaintext.

Open the file `crackcaesar.py`, which contains a modified version of the program you just wrote.

The program has been modified to use functions—which we will use shortly—and variable names have been switched to show that the program now goes from ciphertext to plaintext.

The first modification we need is to make the decryption repeat to try each possible key. We'll place all of our previous code in a loop. (except for the input ciphertext)

crackcaesar.py

```
9           ciphertext = input("Enter ciphertext: ")
10
11          # Try each possible key
12          for shift in range(1, 26):
13                  plaintext = ""
```

We also need a way to tell when the solution has been found. To do this, we will check for the following suspect keywords: "*Attack*", "*enemy*", "*ally*" and "*retreat*".

After generating each possible plaintext, we'll call the `check()` function. **Change** the following line:

```
25 print(plaintext)
```

to:

```
25 if check() == True:
26          print(plaintext)
27          break
```

The `check()` function will be very simple. It simply returns `True` if one of the words is found.

```
28 def check():
29          global plaintext
30          return true
```

Currently, the function always returns true, meaning the program always thinks it's found the solution on the first attempt.

The following code returns true if the word "attack" is found in the possible plaintext. **Extend** the function to find the other 3 words.

```
35 def check():
36          global plaintext
37          found = False
38
39          if "attack" in plaintext:
40                  found = True
41
42          return found
```

Your program should now be complete. Try decoding the following ciphertexts:

"jwljwslslgfuw"
"exxegoirvsyxi"
"nbyyhygsulyxyzyunyx"
"wjnsktwhjfqqdytxtzym"

3 Transposition Ciphers

We've taken a look at substitution ciphers, when letters are replaced by other letters. Another common type of cipher is a *transposition* cipher, where the letters stay the same, but are reordered in the ciphertext.

3.1 Scytale

Figure 5: A scytale device

The scytale dates back to Ancient Greece and Sparta, and is a very simple to use device for encrypting methods.

A person wishing to encrypt a message would simply wrap a long strip of parchment around the cylinder of the scytale and write their message across in rows. When unwrapped, the message would appear to be nonsense.

- The scytale has a cryptographic key, what is it?

w	e	a	r	e
d	i	s	c	o
v	e	r	e	d
h	e	l	p	

w d v h e i e e a s r l r c e p e o d

Figure 6: A scytale plaintext, and its unwrapped ciphertext