Caesar Ciphers

Encrypting messages is older than Ancient Rome, although with the advent of computers, our encryption techniques have had to become ever more sophisticated. To begin with, we investigate the Caesar Cipher.

The Caesar Cipher is one of the most basic forms for encrypting a message:

- ***Identify the position of the letter you want to encode in the alphabet.***
- ***Move along the alphabet a fixed amount (the 'key' of the cipher).***
- ***Write this new letter in place of the old letter.***

An efficient way to encode or decrypt a cipher like this is to write out the shifted alphabet:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
CDEFGHIJKLMNOPQRSTUVWXYZAB

For example, the word CIPHER, encrypted with key 2, gives EKRJGT.

The following quote has been encrypted using a Caesar cipher whose key is unknown:

XYD KVV DRKD MYEXDC MKX LO MYEXDON,

KXN XYD KVV DRKD MKX LO MYEXDON MYEXDC.

Decrypt it.

Here's a somewhat longer message, taken from the 2019 Cipher Challenge test page:

WR FDSWDLQ RPHQ
IURP SURI. YDQGLYHU
VXEMHFW LQWHUFHSW

WKH VRYLHW PHVVDJH FRQWDLQHG YDOXDEOH LQIRUPDWLRQ DQG D WDQWDOLVLQJ FOXH FRQFHUQLQJ WKH REMHFW ZH DUH UHTXLUHG WR UHFRYHU. WKH NWKUHH LV EHOLHYHG WR EH WKH ILUVW VHUYLQJ QXFOHDU SRZHUHG ERDW LQ WKH VRYLHW IOHHW DQG LWV FDSWDLQ LV DQ HASHULHQFHG RIILFHU. WKH FKRLFH RI WKLV ERDW PDB LQGLFDWH WKDW WKH VRYLHWV LQWHQG WR VWDB VXEPHUVHG IRU VRPH WLPH, DQG WKLV FRQILUPV PB VXVSLFLRQ WKDW WKHB GR QRW KDYH JRRG LQWHOOLJHQFH FRQFHUQLQJ WKH ORFDWLRQ RI WKH REMHFW. JLYHQ LWV DSSDUHQW GHVLJQDWLRQ DV D EHDFRQ L DP SXCCOHG WKDW WKHUH LV QR VLJQDO IRU XV WR IROORZ. L ZRXOG UHFRPPHQG PRQLWRULQJ WKH VHFWUXP IRU XQHASHFWHG DFWLYLWB QHDU WKH FUDVK VLWH. RXU UDGLR URRP VHQW PH WKH HQFORVHG VLJQDO ZKLFK LV EHOLHYHG WR KDYH EHHQ WUDQVPLWWHG EB WKH NWKUHH SULRU WR GLYLQJ. DV BRX SUHGLFWHG LW LV HQFUBSWHG, EXW, LW DSSHDUV, RQOB OLJKWOB, DQG L ZLOO HQGHDYRXU WR GHFLSKHU LW TXLFNOB. PHDQZKLOH L ZRXOG UHFRPPHQG WKDW ZH LPSURYH VHFXULWB EB KLGLQJ WKH ZRUG VWUXFWXUH RI RXU PHVVDJHV XVLQJ WKH VWDQGDUG ILYH OHWWHU EORFN IRUPDW.

You can access the message directly by searching for National Cipher Challenge or go to: www.cipherchallenge.org. Here you can also register a team and sign up for the challenge, run by the University of Sheffield, which usually runs from the end of October each year.

Turn over to see how a bit of coding could help…

Cracking Codes *with Python*

Because cracking codes is a combination of computation and language, you will need to learn how to work efficiently with text in Python. Also, for long messages, you will find it helpful to be able to upload entire files into your program, and generate new files.

Your Toolkit: This is a summary of a few useful tools you have at your disposal when using Python to crack codes. It is up to you to see how to combine these into functional code.

Strings	Python stores text in objects called **strings**, which are indexed just like lists, so that `"hello"[0]` returns `"h"` and `"hello"[3:5]` returns `"lo"`. We can also find out where a letter is in a string using the **index** method: `"abcde".index("b")` returns `"b"`.

Files	Python can read from a text file, and save to a text file. Use:
	`with open("test.txt", "r") as my_file:`
	`  lines = my_file.readlines()`
	to create a list of the lines within the file. Similarly, to save text to a file:
	`with open("new_file.txt", "w") as new_file:`
	`  new_file.write("Hello!")`

Example code

The first step is always to decide what you want your program to do. This may sound obvious, but knowing exactly what you are going to feed into your program, and how, and what you want it to spit back out, and how, is a fundamentally important step in solving the problem.

In this case, we'll start by hard-coding an encrypted message directly into our program, and make it print out for us the result of Caesar-shifting the letters as the result (note: encrypting and decrypting are just reverse processes, so if the code was encrypted with a key of 2, we would need a key of 24 to decrypt it):

```
1: message = "B'w ktmaxk atox t lehp
   lbqixgvx matg t gbfuex labeebgz."
2: message = message.upper()
3:
```

Messages may be written in a combination uppercase and lowercase letters, and they may well include numerical and punctuation characters in addition to the 26-letter alphabet. By applying the **upper** method to the string, we convert any lowercase letters to uppercase.

Now we have a message in ALL CAPITALS, but may still contain numbers, punctuation, spaces, etc. We need some way to look at each character in turn, decide if it's a letter, and, if it is, encrypt it. Note that strings are immutable, which means that, unlike lists where new items can be added or removed, it's usually simplest with strings to create a whole new string using the original string. To test whether a character is a letter, we can define an alphabet:

```
1: abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2: message = "B'w ktmaxk atox t lehp
   lbqixgvx matg t gbfuex labeebgz."
3: message = message.upper()
4:
```

There are more efficient methods to find out what type of character you're dealing with, but defining an alphabet like this will be useful for us when we want to apply our cipher in a moment.

Now we're ready to loop through every character in our message, and check to see if it's in **abc**.

```
 1: abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 2: message = "B'w ktmaxk atox t lehp
    lbqixgvx matg t gbfuex labeebgz."
 3: message = message.upper()
 4:
 5: new = ""
 6: for char in message:
 7:   if char in abc:
 8:     new = new + shift(char, 1)
 9:   else:
10:     new = new + char
11: print(new)
```

We define a new (currently blank) string called **new** which we will redefine each time by either adding on a shifted character, or adding on the same character unchanged (if it isn't in our alphabet).
*You will notice that I made up a function called **shift** that we haven't yet created! This is a really helpful way of breaking down a problem into manageable chunks. I know what this function should do, and I'll write it later…*

We will need to define the function **shift**. Because it will form a key component of our Caesar shift code, and will be used repeatedly, it is an ideal candidate for becoming a function in its own right.

```
 1: abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 2: message = "B'w ktmaxk atox t lehp
    lbqixgvx matg t gbfuex labeebgz."
 3: message = message.upper()
 4:
 5: def shift(char, key):
 6:   pos = abc.index(char)
 7:   new_pos = (pos + key) % 26
 8:   return abc[new_pos]
 9:
10: new = ""
11: for char in message:
12:   if char in abc:
13:     new = new + shift(char, 1)
14:   else:
15:     new = new + char
16: print(new)
```

Functions are usually defined near the top of the program: they will be ignored when Python first reads through the code, and will only have their code carried out if and when they are 'called' (referenced) while Python is carrying out the main code.

Often functions take 'arguments' (input values) which are used to modify exactly what the function does. In this case, the arguments are **char** and **key**. Functions also usually produce a 'return' value (an output), which in this case is the shifted character.

This does a shift of 1. To automate further, enclose the main body of the code in a loop, or even make it into a function, then we can loop through all possible keys (there are only 26).

```
 1: abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 2: message = "B'w ktmaxk atox t lehp
    lbqixgvx matg t gbfuex labeebgz."
 3:
 4: def shift(char, key):
. . .
 8:
 9: def caesar(msg, key):
10:   new = ""
11:   for char in msg.upper():
12:     if char in abc:
13:       new = new + shift(char, key)
14:     else:
15:       new = new + char
16:   return new
17:
18: for i in range(26):
19:   print(caesar(message, i))
```

The **caesar** function takes as its input a message and a key.

The output is a shifted message.

*Notice that I have moved the **.upper()** call to within the **caesar** function so it is dealt with automatically. This ensures that even if a message in lowercase is passed to the function, it will still deal with it properly.*

Can you identify the correct original message from your 26 outputs?

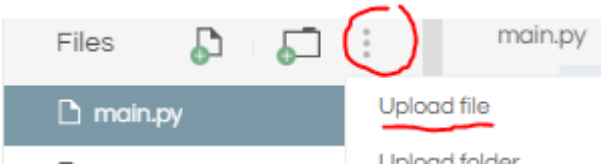
The next level… working with text files

For longer messages, hard-coding the message into the program is not very efficient.

Ideally, we would have a text document containing the secret message, have our program read the contents of the file, perform a trial decryption, then return the output result to a new file.

Python performs both reading and writing of files using the same notation.

1. Make a test file: write some things into Notepad, and save it somewhere as **test.txt**



2. If you're using repl.it, click the three dots in the left-hand pane to upload a file. Once it's done, it should appear in the list just under **main.py**.

3. Enter the code below. This will produce a list (called **lines**) of each line of text in **test.txt**. It will print this list to the shell (the black window on the right) so you can see what it looks like to Python, and it will use the **join** method to attach them all together into a single string.

```
 1: with open("test.txt", "r") as my_file:
 2:   lines = my_file.readlines()
 3:
 4: print(lines)
 5:
 6: message = "".join(lines)
 7: print(message)
 8:
 9: with open("output.txt", "w") as new:
10:   new.write(message)
```

Python opens the specified document in 'read' mode, and loops through the lines, adding them to a list (**lines**).

The **join** method creates a string from the list of lines, linking each one with, in this case, nothing in between ("" is an empty string. Try replacing it with "-" to see the difference).

Finally, a new file is opened (created or overwritten if it already exists), in 'write' mode, and the message is written into the code.

See if you can incorporate file input and output into your Caesar cipher program. If you can upload a file containing, say, the code from one of the National Cipher Challenge problems, you could run your code and produce a different text file for each cipher.

You can improve on this still further by having your code decide whether a particular decrypted message is readable or gobbledygook. Start with a basic (but surprisingly effective) test like:

```
if "THE" in cipher(message, i):
  print(cipher(message, i))
```