Secret Codes with Python

Strings

Our function is going to take a message, go through each letter one by one and replace it with another letter. In programming, words and sentences are called **strings**. We write each string between inverted commas:

```
word = 'elephant'
sentence = 'can a python eat an elephant?'
```

In Python we can do some clever things with these strings. You probably already know that we can **print** them:

```
sentence = 'can a python eat an elephant?'
print(sentence)
```

We can also stick strings to each other, using the plus sign:

```
word1 = 'super'
word2 = 'fun'
word3 = 'time'
longword = word1 + word2 + word3
print(longword)
```

One trick that will be very useful for our program is to take a string and create a list of all its letters. That way we can go through the letters and encode them one at a time. We use the **list** function to do this:

```
word = 'elephant'
letters = list(word)
print(letters)
```

The encode() function

The main part of our program is going to be the encode() function. This function will take a message and encode it by shifting the letters along by 3. For this part of the exercise you should use the editor to write your code in.

In cryptography we call the original message the **plaintext**, and the encoded message is the **ciphertext**. So we know that the first line will be

```
def encode(plaintext):
```

and the last line will be

```
return ciphertext
```

OK, that's the easy bit. Now let's think about what goes in the middle.

First of all we'll make a list of the letters in the message. We're also going to make a list of all the letters in the alphabet.

This is a bit like making the cipher wheel that we used earlier.

OK, this is the clever part.

*Whenever we want to encode a letter, we'll look up it's position, or **index**, in the alphabet list.*

So if we get the letter *C* we look at where it is in the list, and see that its index is 2.

Then we just add 3 to get the index of the replacement letter (5 in this case), *and use that index to look up the new letter in the alphabet* (which turns out to be *F*).

*Now, to do that for every letter in the message, we'll need to go through each letter one by one. Sounds like it's time for a **for loop**!* Since we've already turned our message into a list of letters, it's really easy to just loop through all of the letters in the list.

Now, we're almost there. We can go through the letters in the message and encode them, but how do we create an encoded message out of the new letters?
Remember how we added different words together before, to make a longer word. We can do the same thing here to add new letters to the encoded message one at a time.

So when we put it all together, we'll have a for loop which goes through each of the letters in the original message, changes it to the new letter, and adds it to the encoded message. All we need at the start is an empty message that we can start adding letters to. To create an empty string, we just put the speech marks with nothing in between them.

And that's everything! So, here are all main ideas you need to put in your code. Now give it a go! Remember to be careful about using **indentation**.

```
def encode(plaintext):
```

1. *Create a list of all the letters in the alphabet.*
2. *Create a list of all the letters in the plaintext.*
3. *Create an empty string for the ciphertext.*
4. *For every letter from the plaintext:*
    a. *Get its index from the alphabet*
    b. *Add 3 to the index, and get the letter with the new index*
    c. *Save that new letter in the ciphertext*
5. **return** ciphertext

Think you've got it? Save the file, and see if it will encode the word *banana*:

```
encode('banana')
```

Did that work? How about a different word?

```
encode('berry')
```

OK, so I lied before. We're not *quite* done yet. Can you tell what the problem is? It's to do with one of the letters in this word.

The problem here is with the letter *y*. When the program tries to encode this letter, it looks up its index in the alphabet list, which is 24. To get the new letter, it adds 3, getting 27, and looks up that index in the list. The problem is that the alphabet is only 26 letters long, so the index 27 doesn't exist!

How can we fix this problem? Somehow we need the program to go back and start counting from 0 again when it reaches 26. Thankfully this sort of problem is well known in computing and mathematics, and the solution is something called **modulo**. It works a bit like when you do division and get a remainder. So if you divide 22 by 5, you get 4 remainder 2. The remainder is the modulo bit. In Python we use the percent sign to do it.

Let's try out a few modulo operations in Python. Can you guess what the answers will be?

```
print(9%6)
print(22%10)
print(12%4)
print(3%8)
```

Our alphabet is 26 letters long, so can you guess what modulo we would need to use? Right, it is 26. Whenever a number bigger than 26 gets put in, the program will start from 0 again. *All we have to do is change the line where we calculate the new letter index.*

Now try encoding the word *berry* again. Did it work? Use your cipher wheel to check it.

Additional exercises

If you've got this far, well done! We covered a lot of ground with that function. I think we can improve it though. Here are some ideas for what else you could do with it:

1. **Specify number of letters to shift.** Our cipher shifts the alphabet by 3 to encode messages, but with your cipher wheel you can choose any value between 0 and 25. Add an argument **k** to the function so that you can do this.
2. **Write a decode() function.** Now that you know how to do the **encode()** function, you shouldn't find this too difficult. Can you think of a way to "cheat" here?
3. **Other characters.** What happens if you try to decode a sentence like "i like bananas"? Our program doesn't know how to deal with any characters that aren't in its alphabet. When you write a message with a space, or a full stop, it tries to look it up in the alphabet and doesn't find it. What we'd like is for the program to first check if the character is in the alphabet. If it is, then it should encode it. If not, it should just add it to the cipher text unchanged. Can you think of a feature of Python that can help you with this?
4. **Capital letters.** Our program is looking very good now, but I can still think of a problem. What happens if you try to encode capital letters. There are different ways of fixing this problem. The easiest is probably to just change the message at the start so that all of the capital letters become small letters. You can use the **lower()** function to do this.
5. **More than one Caesar cipher.** What would happen if we try to encode a message twice? Try encoding it with step 3, and then with step 4. Now try encoding the same message separately, with a step 7.

You are not expected to complete the whole worksheet during the workshop, but it would be really good for you if you try completing it on your own. And don't worry if you get stuck! Just come to the drop-in and ask for help!