Four simple programs

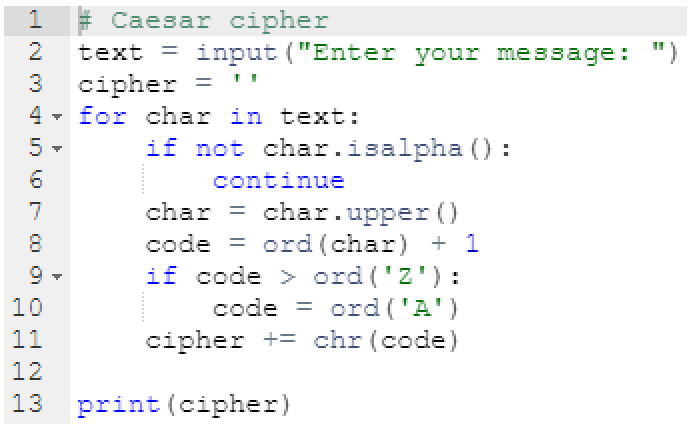
The Caesar Cipher: encrypting a message

We're going to show you four simple programs in order to present some aspects of string processing in Python. They are purposefully simple, but the lab problems will be significantly more complicated.

The first problem we want to show you is called the **Caesar cipher** - more details here: https://en.wikipedia.org/wiki/Caesar_cipher.

This cipher was (probably) invented and used by Gaius Julius Caesar and his troops during the Gallic Wars. The idea is rather simple - every letter of the message is replaced by its nearest consequent (*A* becomes *B*, *B* becomes *C*, and so on). The only exception is *Z*, which becomes *A*.

The program below is a very simple (but working) implementation of the algorithm.



```
1  # Caesar cipher
2  text = input("Enter your message: ")
3  cipher = ''
4  for char in text:
5      if not char.isalpha():
6          continue
7      char = char.upper()
8      code = ord(char) + 1
9      if code > ord('Z'):
10         code = ord('A')
11     cipher += chr(code)
12
13 print(cipher)
```

We've written it using the following assumptions:

- it accepts Latin letters only (note: the Romans used neither whitespaces nor digits)
- all letters of the message are in upper case (note: the Romans knew only capitals)

Let's trace the code:

- line 02: ask the user to enter the open (unencrypted), one-line message;
- line 03: prepare a string for an encrypted message (empty for now)
- line 04: start the iteration through the message;
- line 05: if the current character is not alphabetic...
- line 06: ...ignore it;
- line 07: convert the letter to upper-case (it's preferable to do it blindly, rather than check whether it's needed or not)
- line 08: get the code of the letter and increment it by one;
- line 09: if the resulting code has "left" the Latin alphabet (if it's greater than the *Z* code)...
- line 10: ...change it to the *A* code;
- line 11: append the received character to the end of the encrypted message;
- line 13: print the cipher.

The code, fed with this message:

```
AVE CAESAR
```

outputs:

```
BWFDBFTBS
```

Do your own tests.

The Caesar Cipher: decrypting a message

The reverse transformation should now be clear to you - let's just present you with the code as-is, without any explanations.

Look at the code.

```python
# Caesar cipher - decrypting a message
cipher = input('Enter your cryptogram: ')
text = ''
for char in cipher:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) - 1
    if code < ord('A'):
        code = ord('Z')
    text += chr(code)

print(text)
```

Check carefully if it works. Use the cryptogram from the previous program.

The Numbers Processor

The third program shows a simple method allowing you to input a line filled with numbers, and to process them easily. Note: the routine `input()` function, combined together with the `int()` or `float()` functions, is unsuitable for this purpose.

The processing will be extremely easy - we want the numbers to be summed.

Look at the code below.

```python
# Numbers Processor

line = input("Enter a line of numbers - separate them with spaces: ")
strings = line.split()
total = 0
try:
    for substr in strings:
        total += float(substr)
    print("The total is:", total)
except:
    print(substr, "is not a number.")
```

Let's analyze it.

Using list comprehension may make the code slimmer. You can do that if you want.

Let's present our version:

- line 03: ask the user to enter a line filled with any number of numbers (the numbers can be floats)
- line 04: split the line receiving a list of substrings;
- line 05: initiate the total sum to zero;
- line 06: as the string-float conversion may raise an exception, it's best to continue with the protection of the try-except block;
- line 07: iterate through the list...
- line 08: ...and try to convert all its elements into float numbers; if it works, increase the sum;
- line 09: everything is good so far, so print the sum;
- line 10: the program ends here in the case of an error;
- line 11: print a diagnostic message showing the user the reason for the failure.

The code has one important weakness - it displays a bogus result when the user enters an empty line. Can you fix it?

The IBAN Validator

The fourth program implements (in a slightly simplified form) an algorithm used by European banks to specify account numbers. The standard named **IBAN** (International Bank Account Number) provides a simple and fairly reliable method of validating the account numbers against simple typos that can occur during rewriting of the number e.g., from paper documents, like invoices or bills, into computers.

You can find more details here: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

An IBAN-compliant account number consists of:

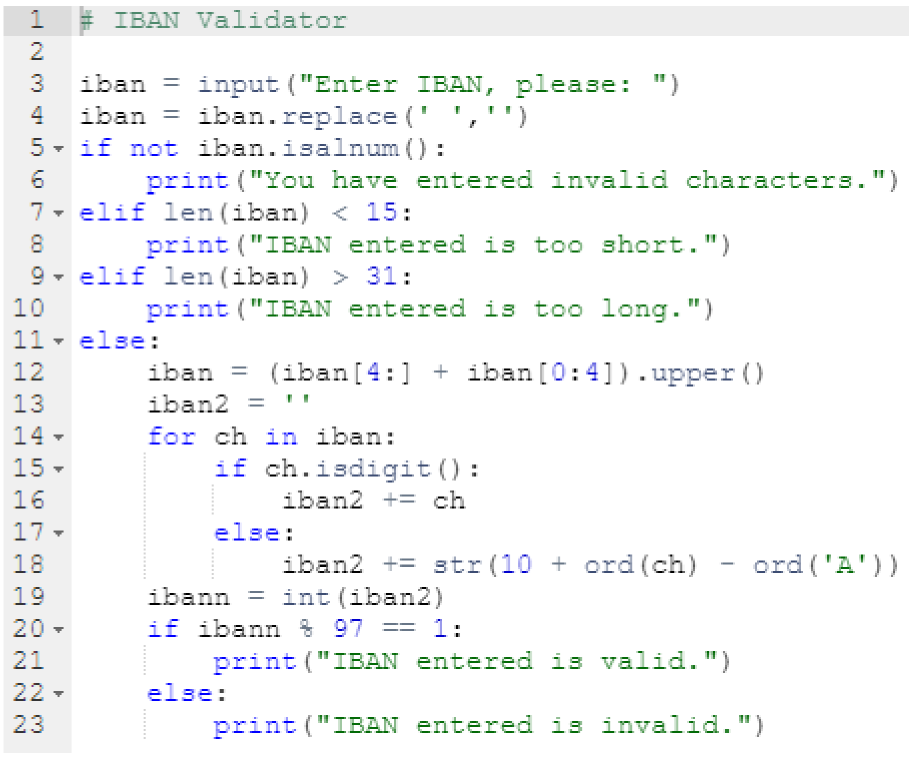
- a two-letter country code taken from the ISO 3166-1 standard (e.g., *FR* for France, *GB* for Great Britain, *DE* for Germany, and so on)
- two check digits used to perform the validity checks - fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good;
- the actual account number (up to 30 alphanumeric characters - the length of that part depends on the country)

The standard says that validation requires the following steps (according to Wikipedia):

- (step 1) Check that the total IBAN length is correct as per the country (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
- (step 2) Move the four initial characters to the end of the string (i.e., the country code and the check digits)

- (step 3) Replace each letter in the string with two digits, thereby expanding the string, where *A = 10*, *B = 11 ... Z = 35*;
- (step 4) Interpret the string as a decimal integer and compute the remainder of that number on division by 97; If the remainder is `1`, the check digit test is passed and the IBAN might be valid.

Look at the code.



```
1  # IBAN Validator
2
3  iban = input("Enter IBAN, please: ")
4  iban = iban.replace(' ','')
5  if not iban.isalnum():
6      print("You have entered invalid characters.")
7  elif len(iban) < 15:
8      print("IBAN entered is too short.")
9  elif len(iban) > 31:
10     print("IBAN entered is too long.")
11 else:
12     iban = (iban[4:] + iban[0:4]).upper()
13     iban2 = ''
14     for ch in iban:
15         if ch.isdigit():
16             iban2 += ch
17         else:
18             iban2 += str(10 + ord(ch) - ord('A'))
19     ibann = int(iban2)
20     if ibann % 97 == 1:
21         print("IBAN entered is valid.")
22     else:
23         print("IBAN entered is invalid.")
```

Let's analyze it:

- line 03: ask the user to enter the IBAN (the number can contain spaces, as they significantly improve number readability...
- line 04: ...but remove them immediately)
- line 05: the entered IBAN must consist of digits and letters only - if it doesn't...
- line 06: ...output the message;
- line 07: the IBAN mustn't be shorter than 15 characters (this is the shortest variant, used in Norway)
- line 08: if it is shorter, the user is informed;
- line 09: moreover, the IBAN cannot be longer than 31 characters (this is the longest variant, used in Malta)
- line 10: if it is longer, make an announcement;
- line 11: start the actual processing;
- line 12: move the four initial characters to the number's end, and convert all letters to upper case (step 02 of the algorithm)
- line 13: this is the variable used to complete the number, created by replacing the letters with digits (according to the algorithm's step 03)
- line 14: iterate through the IBAN;
- line 15: if the character is a digit...
- line 16: just copy it;
- line 17: otherwise...

- line 18: ...convert it into two digits (note the way it's done here)
- line 19: the converted form of the IBAN is ready - make an integer out of it;
- line 20: is the remainder of the division of `iban2` by `97` equal to `1`?
- line 21: If yes, then success;
- line 22: Otherwise...
- line 23: ...the number is invalid.

Let's add some test data (all these numbers are valid - you can invalidate them by changing any character).

- British: `GB72 HBZU 7006 7212 1253 00`
- French: `FR76 30003 03620 00020216907 50`
- German: `DE02100100100152517108`

If you are an EU resident, you can use you own account number for tests.