Python Essentials 2: Module 5

Modules, packages string and list methods, and exceptions

In this module, you will learn about:

Python modules: their rationale, function, how to import them in different ways, and present the content of some standard modules provided by Python;
the way in which modules are coupled together to make packages.
the concept of an exception and Python's implementation of it, including the try-except instruction, with its applications, and the raise instruction.
strings and their specific methods, together with their similarities and differences compared to lists.

(part 8)

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 in the editor is a very simple (but working) implementation of the algorithm.

```
# Caesar cipher.
text = input("Enter your message: ")
cipher = ''
for char in text:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) + 1
    if code > ord('Z'):
        code = ord('A')
    cipher += chr(code)

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

BWFDBFTBS
output

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

```
# 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)
```

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 in the editor. Let's analyze it.

```
# 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.")
```

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?

```
# Numbers Processor.

line = input("Enter a line of numbers - separate them with spaces: ")
strings = line.split()
total = 0

for substr in strings:
    try:
        total += float(substr)
    except:
        pass
print("The total is:", total)
```

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 in the editor.

```
# IBAN Validator.

iban = input("Enter IBAN, please: ")
iban = iban.replace(' ','')

if not iban.isalnum():
    print("You have entered invalid characters.")
elif len(iban) < 15:
    print("IBAN entered is too short.")
elif len(iban) > 31:
    print("IBAN entered is too long.")
else:
    iban = (iban[4:] + iban[0:4]).upper()
    iban2 = ''
    for ch in iban:
        if ch.isdigit():
            iban2 += ch
        else:
            iban2 += str(10 + ord(ch) - ord('A'))
    iban = int(iban2)
    if iban % 97 == 1:
        print("IBAN entered is valid.")
    else:
        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.

Key takeaways

1. Strings are key tools in modern data processing, as most useful data are actually strings. For example, using a web search engine (which seems quite trivial these days) utilizes extremely complex and complicated string processing, involving unimaginable amounts of data.

2. Comparing strings in a strict way (as Python does) can be very unsatisfactory when it comes to advanced searches (e.g. during extensive database queries). Responding to this demand, a number of fuzzy string comparison algorithms has been created and implemented. These algorithms are able to find strings which aren't equal in the Python sense, but are similar.

One such concept is the Hamming distance, which is used to determine the similarity of two strings. If this problem interests you, you can find out more about it here: https://en.wikipedia.org/wiki/Hamming_distance. Another solution of the same kind, but based on a different assumption, is the Levenshtein distance described here: https://en.wikipedia.org/wiki/Levenshtein_distance.

3. Another way of comparing strings is finding their acoustic similarity, which means a process leading to determine if two strings sound similar (like "raise" and "race"). Such a similarity has to be established for every language (or even dialect) separately.

An algorithm used to perform such a comparison for the English language is called Soundex and was invented – you won't believe – in 1918. You can find out more about it here: https://en.wikipedia.org/wiki/Soundex.

4. Due to limited native float and integer data precision, it's sometimes reasonable to store and process huge numeric values as strings. This is the technique Python uses when you force it to operate on an integer number consisting of a very large number of digits.

LAB

Estimated time
30-90 minutes

Level of difficulty
Hard

Pre-requisites
Module 1.11.1.1, Module 1.11.1.2

Objectives
improving the student's skills in operating with strings;
converting characters into ASCII code, and vice versa.
Scenario
You are already familiar with the Caesar cipher, and this is why we want you to improve the code we showed you recently.

The original Caesar cipher shifts each character by one: a becomes b, z becomes a, and so on. Let's make it a bit harder, and allow the shifted value to come from the range 1..25 inclusive.

Moreover, let the code preserve the letters' case (lower-case letters will remain lower-case) and all non-alphabetical characters should remain untouched.

Your task is to write a program which:

asks the user for one line of text to encrypt;
asks the user for a shift value (an integer number from the range 1..25 - note: you should force the user to enter a valid shift value (don't give up and don't let bad data fool you!)
prints out the encoded text.
Test your code using the data we've provided.

Test data
Sample input:

```
abcxyzABCxyz 123
2
```

Sample output:

```
cdezabCDEzab 123
```

Sample input:

```
The die is cast
25
```

Sample output:

```
Sgd chd hr bzrs
```

LAB

Estimated time
15-30 minutes

Level of difficulty
Easy

Objectives
improving the student's skills in operating with strings;
encouraging the student to look for non-obvious solutions.
Scenario
Do you know what a palindrome is?

It's a word which look the same when read forward and backward. For example, "kayak" is a palindrome, while "loyal" is not.

Your task is to write a program which:

asks the user for some text;
checks whether the entered text is a palindrome, and prints result.
Note:

assume that an empty string isn't a palindrome;
treat upper- and lower-case letters as equal;
spaces are not taken into account during the check - treat them as non-existent;
there are more than a few correct solutions - try to find more than one.
Test your code using the data we've provided.

Test data
Sample input:

```
Ten animals I slam in a net
```

Sample output:

```
It's a palindrome
```

Sample input:

```
Eleven animals I slam in a net
```

Sample output:

```
It's not a palindrome
```

LAB

Estimated time
30-60 minutes

Level of difficulty
Easy

Objectives
improving the student's skills in operating with strings;
converting strings into lists, and vice versa.
Scenario
An anagram is a new word formed by rearranging the letters of a word, using all the original letters exactly once. For example, the phrases "rail safety" and "fairy tales" are anagrams, while "I am" and "You are" are not.

Your task is to write a program which:

asks the user for two separate texts;
checks whether, the entered texts are anagrams and prints the result.
Note:

assume that two empty strings are not anagrams;
treat upper- and lower-case letters as equal;
spaces are not taken into account during the check - treat them as non-existent
Test your code using the data we've provided.

Test data
Sample input:

```
Listen
Silent
```

Sample output:

```
Anagrams
```

Sample input:

```
modern
norman
```

Sample output:

```
Not anagrams
```

LAB

Estimated time
15-30 minutes

Level of difficulty
Easy

Objectives
improving the student's skills in operating with strings;
converting integers into strings, and vice versa.
Scenario
Some say that the Digit of Life is a digit evaluated using somebody's birthday. It's simple - you just need to sum all the digits of the date. If the result contains more than one digit, you have to repeat the addition until you get exactly one digit. For example:

1 January 2017 = 2017 01 01
2 + 0 + 1 + 7 + 0 + 1 + 0 + 1 = 12
1 + 2 = 3
3 is the digit we searched for and found.

Your task is to write a program which:

asks the user her/his birthday (in the format YYYYMMDD, or YYYYDDMM, or MMDDYYYY - actually, the order of the digits doesn't matter)
outputs the Digit of Life for the date.
Test your code using the data we've provided.

Test data
Sample input:

19991229

Sample output:

6

Sample input:

20000101

Sample output:

4

LAB

Estimated time
30-45 minutes

Level of difficulty
Medium

Objectives
improving the student's skills in operating with strings;
using the find() method for searching strings.
Scenario
Let's play a game. We will give you two strings: one being a word (e.g., "dog") and the second being a combination of any characters.

Your task is to write a program which answers the following question: are the characters comprising the first string hidden inside the second string?

For example:

if the second string is given as "vcxzxduybfdsobywuefgas", the answer is yes;
if the second string is "vcxzxdcybfdstbywuefsas", the answer is no (as there are neither the letters "d", "o", or "g", in this order)
Hints:

you should use the two-argument variants of the pos() functions inside your code;
don't worry about case sensitivity.
Test your code using the data we've provided.

Test data
Sample input:

donor
Nabucodonosor
Sample output:

Yes

Sample input:

donut
Nabucodonosor
Sample output:

No

LAB

Estimated time
60-90 minutes

Level of difficulty
Hard

Objectives
improving the student's skills in operating with strings and lists;
converting strings into lists.
Scenario
As you probably know, Sudoku is a number-placing puzzle played on a 9x9 board. The player has to fill the board in a very specific way:

each row of the board must contain all digits from 0 to 9 (the order doesn't matter)
each column of the board must contain all digits from 0 to 9 (again, the order doesn't matter)
each of the nine 3x3 "tiles" (we will name them "sub-squares") of the table must contain all digits from 0 to 9.
If you need more details, you can find them here.

Your task is to write a program which:
reads 9 rows of the Sudoku, each containing 9 digits (check carefully if the data entered are valid)
outputs Yes if the Sudoku is valid, and No otherwise.
Test your code using the data we've provided.

Test data
Sample input:

```
295743861
431865927
876192543
387459216
612387495
549216738
763524189
928671354
154938672
```
Sample output:
```
Yes
```

Sample input:

```
195743862
431865927
876192543
387459216
612387495
549216738
763524189
928671354
254938671
```
Sample output:
```
No
```