General

Reuse existing functionality

Functions are control structures that allow to avoid unnecessary **code duplication**. Code duplication is the phenomenon where multiple copies of the same or highly similar code occur in the source code of a program. It is always a good idea to avoid code duplication.

Also take into account the possibility to call other functions while implementing a function. Sometimes it will be explicitly stated in the description of an assignment that you have to reuse an existing function (one that you implemented earlier) in the implementation of a new function. But in other cases such an statement will not be made explicit in de description of the assignment, while it implicitly remains a goal to detect possible code reuse while implementing the functions.

Say for example that you were asked to implement two functions: `maxsum` and `mindiff`. The first function `maxsum` takes three arguments, and needs to return a Boolean value that expresses whether or not the sum of the first two arguments is less than the value of the third argument. The second function `mindiff` takes three arguments, and needs to return a Boolean value that expresses whether or not the absolute value of the difference of the first two arguments is larger than the value of the third argument. Both functions can be implemented as follows.

```
def maxsum(x, y, a):
    return x + y < a

def mindiff(x, y, b):
    return abs(x - y) > b
```

Now, say that you are also asked to implement a third function `minmax` that takes four arguments, and needs to return a Boolean$ value that expresses whether or not the sum of the first two arguments is less than the value of the third argument AND the absolute value of the difference of the first two arguments is larger than the value of the fourth argument. You could implement this function in the following way.

```
def minmax(x, y, a, b):
    return x + y < a and abs(x - y) > b
```

However, this implementation completely *reinvents the wheel* since the condition that needs to be check in this function is nothing but the composition of the two conditions that need to be checked in the functions `maxsum` and `mindiff`. As a result, it is a far better solution to implement the function `minmax` in the following way.

```
def minmax(x, y, a, b):
    return maxsum(x, y, a) and mindiff(x, y, b)
```

In case there is a need to make a modification to your implementation of the function `maxsom` (because you have found out there is a more efficient strategy for the implementation, or the initial implementation contained a *bug*), you only have to make the adjustments at a single location in your source code, and not in two locations if you had copied the source code for the implementation of the function `minmax`.

How does Pythia check floating point numbers

If you have to output a *floating point* number for a given assignment, without an explicit indication about the exact number of decimal digits that has to be displayed on the output (without rounding or truncating), Pythia will check by default that the number is accurate up to six decimal digits. As a result, it does not really matter how many digits are shown on the output.

Functions/methods: return vs print

In assignments where you are asked to implement functions or methods, you should read carefully if the function either needs to **return** a result, or if the function needs to **print** a result. A `return` statement must be used to let the function return a result. The built-in function `print` must be used to let the function print a result to *standard output* (short: *stdout*).

In the assignment C-sum (series 05) you are asked, for example, to write a function `csum` that must **return** a result. One possible correct implementation of this function is

```
def csum(number):
    return number % 9
```

where a `return` statement is used to have the function return a computed value. Suppose that we erroneously used the built-in function `print` to write the computed value to *stdout*, and submitted the following incorrect solution for this assignment.

```
def csum(number):
    print(number % 9)
```

In this case, the Pythia platform would evaluate the submission as a **wrong answer**, where the following feedback would be given on the feedback page.

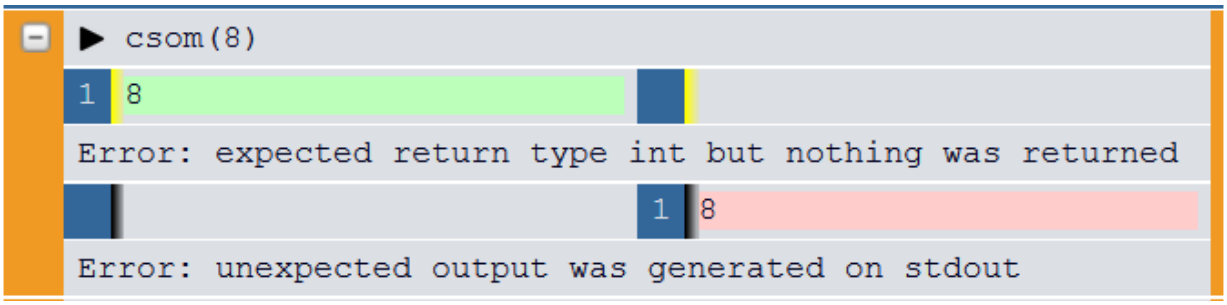

Figure 1: return vs print

The feedback contains two remarks. The first remark indicates that the function was expected to return an integer value (8), but instead the function did not return any value (or more precisely, the function returned the value `None`). This is the meaning of the error message

```
Error: expected return type int but nothing was returned
```

In addition, a second remark indicates that the function has written some information to *stdout*, whereas no information was expected on this output channel. This is the meaning of the error message

```
Error: unexpected output was generated on stdout
```

If you would have used a `return` statement instead of the built-in function `print`, both error message would have disappeared and the Pythia platform would have evaluated the submission as a **correct answer**. Also note that results returned by a function/method are marked with a thick yellow line in the feedback table, whereas results that are printed by a function/method are marked with a thick black line.

Rövarspråket

Specific information

The solution to this assignment comes with realizing that processing a group of successive consonants needs to be postponed until a character is encountered that is not a consonants (which completes the group of successive consonants). This is illustrates by means of the following example, which translates the string `abcdefg` into Rövarspråket.

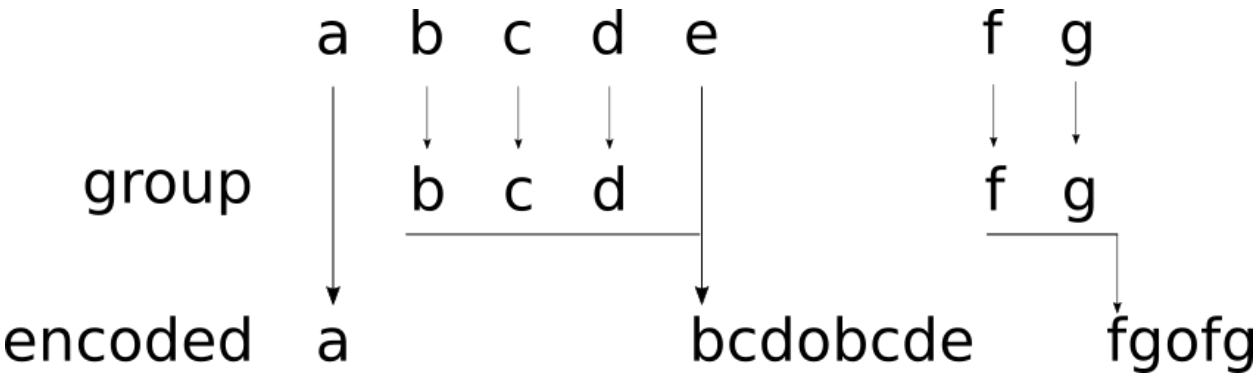

Figure 2: Rövarspråket

The idea is to traverse the string that needs to be encoded character per character. When a consonant is encountered, it is not immediately appended to the encoded string, but instead it is appended to the current group of consecutive consonants. When a non-consonant is encountered, it closes the current group of consecutive consonants. In case such a group has been formed, it is appended to the encoded string, followed by the letter `o` and another repetition of the group of consecutive consonants. Only after this has been done, the non-consonant is appended to the encoded string, and a new group of consonants is started.

After all character of the string that needs to be encoded have been processes, there might still be a group of consecutive consonants that has not been processed. This is the case if the string ends in a consonant. In the example given above, this is the case for the group `fg`. If you end up in this situation, you still need to append the current group of consecutive consonants to the encoded string, followed by the letter `o` and another repetition of the group of consecutive consonants.

Pangrammatic window

Specific information

The easiest strategy to implement the function `window` is to iterate over all possible substrings of the given text. For each of these substrings you then determine whether or not it is a pangram, and you remember which one of the pangram is the shortest one.7

To iterate over all substrings of a given string, you can iterate over all possible start positions of substrings, and the iterate over all possible stop position for each possible start position. This can be implemented using a double `for`-loop. The following code snippet, for example, outputs all substrings of the string `'abc'`, where we only take into account substrings having at least length 1.

```
>>> s = 'abc'
>>> for start in range(len(s) - 1):
...     for stop in range(start, len(s)):
```

```
...         print(s[start:stop + 1])
...

a
ab
abc
b
bc
```

If you think twice, there's even no need to iterate over all possible substrings to find the shortest pangram. If you want to avoid spurious computations, you can give it a thought how you can avoid that Python should do unnecessary work.

Recombination

General information

Iterate both positions and values of sequence types

The built-in function `enumerate` can be used to request an iterator for a given sequence type (strings, lists, tuples, files, ...) that both returns the position and the value at that position for the next element of the sequence type. The example below illustrates for example how this can be used to iterate simultaneous the positions and the characters on those position of a string.

```
>>> for index, character in enumerate('abc'):
...     print("index: {}".format(index))
...     print("character: {}".format(character))
...
index: 0
character: a
index: 1
character: b
index: 2
character: c
```

You can also use this to simultaneously iterate over the characters of two strings.

```
>>> first = 'abc'
>>> second = 'def'
>>> for index, character in enumerate(first):
...     print("{}-{}".format(character, second[index]))
...
a-d
b-e
c-f
```

However, in this case it is better to use the built-in function `zip`, which is especially equipped to iterate over multiple iterable objects at once.

```
>>> first = 'abc'
>>> second = 'def'
>>> for character1, character2 in zip(first, second):
...     print("{}-{}".format(character1, character2))
...
a-d
b-e
c-f
```

Split strings into multiple parts

Sometimes you need to split a string into multiple parts. One way to do this job makes use of the string method `split`. This method takes an optional string argument, that indicates the sequence of characters (called the *separator*) that needs to be used to split the string on which the method is called.

```
>>> string = 'a-b-c-d'
>>> string.split('-')
['a', 'b', 'c', 'd']
```

By default, the method splits the string at all positions where the separator occurs in the string, and places each of the parts in a list that is returned by the method. In case no argument is passed to the method, the string is split at each occurrence of a sequence of whitespace characters (spaces, tabs, newlines, ...). The string method also has another optional argument that you can use to indicate the maximal number of parts in which the string needs to be split.

Because the string method `split` returns a list, you can directly iterate over the elements of the list using a `for`-loop.

```
>>> string = 'a-b-c-d'
>>> for element in string.split('-'):
...     print(element)
...
a
b
c
d
```

Remarks

Because strings are *immutable* you cannot directly change a string. If you want to obtain a modified string, you'll always have to create a new string.

Say, for example, that we want to modify the string `'azyc'` by replacing the substring `'zy'` with the string `'bc'`, without making use of the string method `replace`. This can be done in the following way.

```
>>> string = 'azyd'
>>> substring = 'bc'
>>> # this the wrong way to modify the string
>>> string[1:2] = substring
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> # this is the correct way to modify the string
```

```
>>> string = string[:1] + substring + string[3:]
>>> string
'abcd'
```

Sometimes it is needed to explicitly check if certain conditions hold when executing part of your source code, and the program needs to respond if one of the conditions is not met. One of the easiest ways this can be done in Python is by using the `assert` statement.

```
>>> x = 2
>>> y = 2
>>> assert x == y, 'the values are different'
>>> x = 1
>>> assert x == y, 'the values are different'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: the values are different
```

The general syntax of the `assert` statement is

```
assert <condition>, <message>
```

The `assert` statement checks whether or not the condition is met. If this is not the case, an `AssertionError` will be raised with the message that is given at the end of the `assert` statement. In case this exception is not caught elsewhere in the code (which will always be the case in this course), the execution of the codes halts at the point where the `AssertionError` was raised (*runtime error*).