General

Functions/methods: return vs print

In assignments where you are asked to implement functions (from series 05 onwards) or methods (from series 10 onwards), 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 Dodona platform would evaluate the submission as a **wrong answer**, where the following feedback would be given on the feedback page.

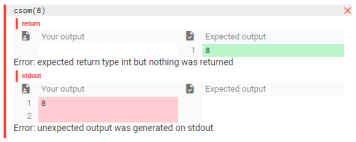

Figure 1: return vs print

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

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 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 `maxsum` (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`.

Iterate both the elements and their positions 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 how this can be used to simultaneously iterate the positions of a string and the characters on those position.

```
>>> for index, character in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'character: {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(f'{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(f'{character1}-{character2}')
...
a-d
b-e
c-f
```