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.

As such, always 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 a statement will not be made explicitly 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 (`bool`) that expresses whether 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 (`bool`) that expresses whether 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.

```python
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 (`bool`) 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.

```python
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 checked 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.

```python
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`.

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 carefully read whether the function 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*).

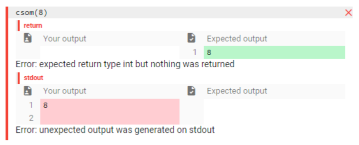
For example, in the assignment C-sum (series 05) you are asked to write a function `csum` that must **return** a result. One possible correct implementation of this function is

```python
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.

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

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



```
csom(8)
return
Your output        Expected output
                   1 8
Error: expected return type int but nothing was returned
stdout
Your output        Expected output
1 8
2
Error: unexpected output was generated on stdout
```

Figure 1: return vs print

The feedback contains two remarks. The first remark indicates that the function was expected to return an integer (8, `int`), 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 Dodona 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, whereas results that are printed by a function/method are marked with the text `stdout`.

Convert letters to their ordinal value

Each character has a corresponding ordinal value (`int`). For example, the question mark (`?`) has ordinal value 63. The value can be easily obtained using the built-in function `ord`.

```python
>>> ord('?')
63
```

The interesting thing about these ordinal values, is that successive letters in the alphabet have successive ordinal values. This holds for both uppercase and lowercase letters. If we know that the letter `a` has ordinal value 97, it immediately follows that the letter `b` must have ordinal value 98. Using this knowledge, we can easily determine the position of a letter in the alphabet.

```python
>>> letter = 'd'
>>> ord(letter)
100
>>> ord('a')
97
>>> ord(letter) - ord('a') + 1   # d is the 4th letter of the alphabet
4
>>> ord('z') - ord('a') + 1      # z is the 26th letter of the alphabet
26
```

Check if certain conditions hold

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

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

The `assert` statement checks whether the condition holds. If this is not the case, an `AssertionError` is raised with the message (`str`) 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*).