Python exercises – series 2

Question 1: Functions

In this exercise we will write functions that produce strings to make a grid:

```
#----#----#
|    |    |
|    |    |
|    |    |
#----#----#
|    |    |
|    |    |
|    |    |
#----#----#
```

If you examine the grid above, you can distinguish two types of lines: the ones containing the corners of the grid elements and the ones with mid pieces.

a) Write a function *create_corners* that produces the following string as a result:

```
"#----#----#\n"
```

and write a function *create_central* that returns the following string

```
"|    |    |\n"
```

Try to use as much as possible the operators + and *!

b) Write a function *create_grid* that uses the two previous functions to produce the example grid shown above. Use the + and * operators. You can test your function by printing the result it produces: *print(create_grid()).*

c) Adapt the functions *create_corners* and *create_central* so that they expect an argument that corresponds to the width of the grid. That parameter determines how wide every square of your grid will be and hence determines how many "-" and " " characters will have to be generated. For example: *create_corners(10)* has to return the following string:

```
"#----------#----------#"
```

d) Extend the *create_grid* function so that it has a parameter that controls the width of the grid squares.

e) Add a second parameter to the function create grid that determines the height of every square of the grid. This corresponds to the amount of midsections that have to be generated. The end result of the function call *create_grid(8,10)* has to produce the following result:

```
#--------#--------#
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
#--------#--------#
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
#--------#--------#
```

f) ***Extra exercise:*** Extend the function to that you can change the number of squares in the grid. Make sure to keep the number of code repetitions as small as possible!

Question 2: Making choices

As a small reminder, the general structure of an **if…elif…else** structure:

```
if condition:
        <block>
elif other_condition:
        <block>
else:
        <block>
```

a) Write a function that inspects a given year and returns what type of year it is, based on the table below:

Type 1	Not a leap year: not divisible by 4
Type 2	Leap year: divisible by 4 but not by a 100
Type 3	Not a leap year: divisible by a 100 but not by 400
Type 4	Leap year: divisible by 400

Is it more interesting to use print or return? Which datatypes can be used and what is the best choice in case we want to use this function in future code?

Question 3: Iterations

A ***FOR*** loop (or counted loop) allows you to repeat a block of code a number of times. The for loop goes over each element in a sequence (list, tuple, string), assigns that element to the iteration variable and executes the iterated code. As a reminder, the for loop structure is:

```
for iteration-variable in sequence:
        <iterated block >
```

The sequence can be an existing list, tuple or string. The **range()** function can be used as well: *range(stop)*, *range(start, stop)* and *range(start,stop,step)*. Note that *stop* is not part of the sequence.

a) Write a function that takes as argument the number of times the word 'bla' has to be printed. Do not work with the operators * and + this time but use a for loop in combination with the *range* function.

b) Write a function that prints the following arrow on the screen:

```
*
**
***
****
***
**
*
```

The size of the arrow (in the example above the size is 4) should be a parameter of your function.

In some cases, it is not known beforehand how many iterations are needed. In those particular cases a **WHILE** (conditional loop) can be used. That type of loop repeats code as long as a certain condition is met. Its structure is:

```
while condition:
        <iterated block >
```

c) Repeat exercise a) and b) but with a WHILE loop instead of a FOR loop.

d) Make a function *my_division* to which you give a dividend and a divisor. The function returns the quotient and remainder. Use only the + and – operators!

Question 4: Iterating over lists

When working on lists you will often make calculations on all (or on a part of) their elements. Fortunately, in Python this can easily be done with a FOR loop that iterates over the elements of a list.

a)

1. Write a function than creates a list with as elements the subsequent powers of -2. Do not use the ** operator. The argument of the function is the number of elements you wish to generate.
2. Write a function that returns the average of the elements of a list. Use this function to computer the average of the first 15 powers of -2.

b) Write a function that reverses a list (the last element becomes the first etc.). Implement this function by copying the values of the list to a new list. Do NOT use the *list.reverse()* method!

c) Write a function that has a string parameter. The function should determine if the string is a palindrome (e.g. *civic, radar, level, rotor, etc.*).

Question 5: Errors and Debugging

a) In the next piece of code there is a common error:

```
temp = 0
while temp<100:
    temp = temp/2-1
```

Copy this snippet of code and evaluate it in PyCharm. What is happening and why? Let's use the debugger!

Place a breakpoint in the margin by clicking next to a line of code. Start the debugger by clicking on the little bug instead of the green arrow. We can now see all values of all assigned variables. You can execute the code step by step and see how the values evolve in each iteration.