Functions and recursion

Recall that in mathematics the factorial of a number n is defined as n! = 1 · 2 · ... · n (as the product of all integer numbers from 1 to n). For example, 5! = 1 · 2 · 3 · 4 · 5 = 120. It is clear that factorial is easy to calculate, using a for loop. Imagine that we need in our program to calculate the factorial of various numbers several times (or in different places of code). Of course, you can write the calculation of the factorial once and then using Copy-Paste to insert it wherever you need it:

```
# compute 3!
res = 1
for i in range(1, 4):
    res *= i
print(res)


# compute 5!
res = 1
for i in range(1, 6):
    res *= i
print(res)
```

However, if we make a mistake in the initial code, this erroneous code will appear in all the places where we've copied the computation of factorial. Moreover, the code is longer than it could be. To avoid re-writing the same logic in programming languages there are functions.

Functions are the code sections which are isolated from the rest of the program and executed only when called. You've already met the function `sqrt()`, `len()` and `print()`. They all have something in common: they can take parameters (zero, one, or several of them), and they can return a value (although they may not return). For example, the function `sqrt()` accepts one parameter and returns a value (the square root of the given number). The `print()` function can take various number of arguments and returns nothing.

Now we want to show you how to write a function called `factorial()` which takes a single parameter — the number, and returns a value — the factorial of that number.

```
def factorial(n):

    res = 1

    for i in range(1, n + 1):

        res *= i

    return res



print(factorial(3))

print(factorial(5))
```

We want to provide a few explanations. First, the function code should be placed in the beginning of the program (before the place where we want to use the function `factorial()`, to be precise). The first line `def factorial(n):` of this example is a description of our function; the word *factorial* is an identifier (the name of our function). Right after the identifier, there goes the list of parameters that our function receives (in parentheses). The list consists of comma-separated identifiers of the parameters; in our case, the list consists of one parameter `n`. At the end of the row, put a colon.

Then goes the function body. In Python, the body must be indented (by Tab or four spaces, as always). This function calculates the value of n! and stores it in the variable `res`. The last line of the function is `return res`, which exits the function and returns the value of the variable `res`.

The `return` statement can appear in any place of a function. Its execution exits the function and returns specified value to the place where the function was called. If the function does not return a value, the return statement won't actually return some value (though it still can be used). Some functions do not need to return values, and the return statement can be omitted for them.

We'd like to provide another example. Here's the function `max()` that accepts two numbers and returns the maximum of them (actually, this function has already become the part of Python syntax).

```
def max(a, b):

    if a > b:

        return a
    else:

        return b


print(max(3, 5))

print(max(5, 3))

print(max(int(input()), int(input())))
```

Now you can write a function `max3()` that takes three numbers and returns the maximum of them.

```
def max(a, b):

    if a > b:

        return a

    else:

        return b


def max3(a, b, c):

    return max(max(a, b), c)


print(max3(3, 5, 4))
```

The built-in function `max()` in Python can accept various number of arguments and return the maximum of them. Here is an example of how such a function can be written.

```
def max(*a):

    res = a[0]

    for val in a[1:]:

        if val > res:

            res = val

    return res
```

```
print(max(3, 5, 4))
```

Everything passed to this function will gather the parameters into a single tuple called `a`, which is indicated by the asterisk.

2. Local and global variables

Inside the function, you can use variables declared somewhere outside of it:

```
def f():
    print(a)
a = 1
f()
```

Here the variable `a` is set to 1, and the function `f()` prints this value, despite the fact that when we declare the function `f` this variable is not initialized. The reason is, at the time of calling the function `f()` (the last string) the variable `a` already *has* a value. That's why the function `f()` can display it.

Such variables (declared outside the function but available inside the function) are called *global*.

But if you initialize some variable inside of the function, you won't be able to use this variable outside of it. For example:

```
def f():
    a = 1
f()
print(a)
```

We receive error `NameError: name 'a' is not defined`. Such variables declared within a function are called *local*. They become unavailable after you exit the function.

What's really charming here is what happens if you change the value of a global variable inside of a function:

```
def f():
    a = 1
    print(a)
a = 0
f()
print(a)
```

This program will print you the numbers 1 and 0. Despite the fact that the value of the variable `a` changed inside the function, outside the function it remains the same! This is done in order to «protect» global variables from inadvertent changes of function. So, if some variable is modified inside the function, the variable becomes a local variable, and its modification will not change a global variable with the same name.

More formally: the Python interpreter considers a variable local to the function, if in the code of this function there is at least one instruction that modifies the value of the variable. Then that variable also cannot be used prior to initialization. Instructions that modify the value of a variable — the operators `=`, `+=`, and usage of the variable as a loop `for` parameter. However, even if the changing-variable statement is never executed, the interpreter cannot check it, and the variable is still local. An example:

```
def f():
    print(a)
    if False:
        a = 0
a = 1
f()
```

An error occurs: `UnboundLocalError: local variable 'a' referenced before assignment`. Namely, in the function `f()` the identifier `a` becomes a local variable, since the function contains the command which modifies the variable `a`. The modifying instruction will never be executed, but the interpreter won't check it. Therefore, when you try to print the variable `a`, you appeal to an uninitialized local variable.

If you want a function to be able to change some variable, you must declare this variable inside the function using the keyword `global`:

```
def f():
    global a
    a = 1
    print(a)
a = 0
f()
print(a)
```

This example will print the output of 1 1, because the variable `a` is declared as global, and changing it inside the function causes the change of it globally.

However, it is better not to modify values of global variables inside a function. If your function must change some variable, let it *return* this value, and you choose when calling the function explicitly assign a variable to this value. If you follow these rules, the functions' logic works independent of the code logic, and thus such functions can be easily copied from one program to another, saving your time.

For example, suppose your program should calculate the factorial of the given number that you want to save in the variable f. Here's how you should not do it:

```
def factorial(n):
    global f
    res = 1
    for i in range(2, n + 1):
        res *= i
    f = res
n = int(input())
factorial(n)
print(f)
# doing other stuff with variable f
```

This is the example of bad code, because it's hard to use another time. If tomorrow you need another program to use the function "factorial", you will not be able to just copy this function from here and paste in your new program. You will have to ensure that that program doesn't contain the variable `f`.

It is much better to rewrite this example as follows:

```
# start of chunk of code that can be copied from program to program

def factorial(n):

    res = 1

    for i in range(2, n + 1):

        res *= i

    return res

# end of piece of code

n = int(input())

f = factorial(n)

print(f)

# doing other stuff with variable f
```

It is useful to say that functions can return more than one value. Here's the example of returning a *list* of two or more values:

```
return [a, b]
```

You can call the function of such list and use it in multiple assignment:

```
n, m = f(a, b)
```

3. Recursion

As we saw above, a function can call another function. But functions can also call itself! To illustrate it, consider the example of the factorial-computing function. It is well known that 0!=1, 1!=1. How to calculate the value of n! for large n? If we were able to calculate the value of (n-1)!, then we easily compute n!, since n!=n·(n-1)!. But how to compute (n-1)!? If we have calculated (n-2)!, then (n-1)!=(n-1)·(n-2)!. How to calculate (n-2)!? If... In the end, we get to 0!, which is equal to 1. Thus, to calculate the factorial we can use the value of the factorial for a smaller integer. This calculation can be done using Python:

```
def factorial(n):

    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

The situation when function calls itself is called *recursion*, and such function is called recursive.

Recursive functions are powerful mechanism in programming. Unfortunately, they are not always effective and often lead to mistakes. The most common error is *infinite recursion*, when the chain of function calls never ends (well, actually, it ends when you run out of free memory in your computer). An example of infinite recursion:

```
def f():
    return f()
```

The two most common reasons causing infinite recursion:

1. Incorrect stopping condition. For example, if in the factorial-calculating program we forget to put the check `if n == 0`, then `factorial(0)` will call `factorial(-1)`, that will call `factorial(-2)`, etc.
2. Recursive call with incorrect parameters. For example, if the function `factorial(n)` calls the function `factorial(n)`, we will also obtain an infinite chain of calls.

Therefore, when coding a recursive function, it is first necessary to ensure it will reach its stopping conditions — to think why your recursion will ever end.

Problem «The length of the segment» (Easy)

Statement

Given four real numbers representing cartesian coordinates: $(x_1,y_1),(x_2,y_2)$(x1,y1),(x2,y2). Write a function `distance(x1, y1, x2, y2)` to compute the distance between the points (x_1,y_1)(x1,y1) and (x_2,y_2)(x2,y2). Read four real numbers and print the resulting distance calculated by the function.

The formula for distance between two points can be found at Wolfram.

Your solution

```
import math

x1 = float(input())

y1 = float(input())

x2 = float(input())

y2 = float(input())

def distance(x1, y1, x2, y2):

   return math.sqrt((x2-x1)**2+(y2-y1)**2)

print (float(distance(x1, y1, x2, y2)))
```

Suggested solution

```
from math import sqrt

def distance(x1, y1, x2, y2):

   return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

x1 = float(input())

y1 = float(input())

x2 = float(input())

y2 = float(input())

print(distance(x1, y1, x2, y2))
```

Problem «Negative exponent» (Easy)

Statement

Given a positive real number aa and **integer** nn.

Compute anan. Write a function `power(a, n)` to calculate the results using the function and print the result of the expression.

Don't use the same function from the standard library.

Your solution

```
def power(a, n):
    res = 1
    for i in range(abs(n)):
        res *= a
    if n >= 0:
        return res
    else:
        return 1 / res
print(power(float(input()), int(input())))
```

Problem «Uppercase» (Medium)

Statement

Write a function `capitalize(lower_case_word)` that takes the lower case word and returns the word with the first letter capitalized. Eg., `print(capitalize('word'))` should print the word `Word`.

Then, given a line of lowercase ASCII words (text separated by a single space), print it with the first letter of each word capitalized using the your own function `capitalize()`.

In Python there is a function `ord(character)`, which returns character code in the ASCII chart, and the function `chr(code)`, which returns the character itself from the ASCII code. For example, `ord('a') == 97`, `chr(97) == 'a'`.

Your solution

```
def capitalize(lower_case_word):
    return str.upper(lower_case_word[0])+ lower_case_word[1:]
s = [str(s) for s in input().split()]
for i in range(len(s)):
    s[i] = capitalize(s[i])
    print (s[i], end = " " )
```

Suggested solution

```
def capitalize(word):

    first_letter_small = word[0]

    first_letter_big = chr(ord(first_letter_small) - ord('a') + ord('A'))

    return first_letter_big + word[1:]



source = input().split()

res = []

for word in source:

    res.append(capitalize(word))

print(' '.join(res))
```

Problem «Exponentiation» (Medium)

Statement

Given a positive real number aa and a non-negative integer nn. Calculate a^nan without using loops, `**` operator or the built in function `math.pow()`. Instead, use recursion and the relation $a^n=a\cdot a^{n-1}$an=a·an−1. Print the result.

Form the function `power(a, n)`.

Your solution

```
def power(a, n):

    if n == 0:

        return 1

    else:

        return a * power(a, n - 1)
```

```
print(power(float(input()), int(input())))
```

Suggested solution

```
def power(a, n):
    if n == 0:
        return 1
    else:
        return a * power(a, n - 1)
print(power(float(input()), int(input())))
```

Problem «Reverse the sequence» (Hard)

Statement

Given a sequence of integers that end with a 00. Print the sequence in reverse order.

Don't use lists or other data structures. Use the *force* of recursion instead.

Your solution

```
def f(n):
    l.append(n)
    if n == 0:
        for elem in l[::-1]:
            print(elem)
    else:
        n = int(input())
        f(n)
l = []
n = int(input())
```

f(n)

Suggested solution

```
def reverse():
   x = int(input())
   if x != 0:
      reverse()
   print(x)
reverse()
```

Problem «Fibonacci numbers» (Hard)

Statement

Given a non-negative integer nn, print the nnth Fibonacci number. Do this by writing a function `fib(n)` which takes the non-negative integer nn and returns the nnth Fibonacci number.

Don't use loops, use the *flair* of recursion instead. However, you should think about why the recursive method is much slower than using loops.

Your solution

```
n = int(input())
def fib(n):
   if n == 0:
      return 1
   elif n == 1:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

```
print(fib(n-1))
```

Suggested solution

```
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


print(fib(int(input())))
```