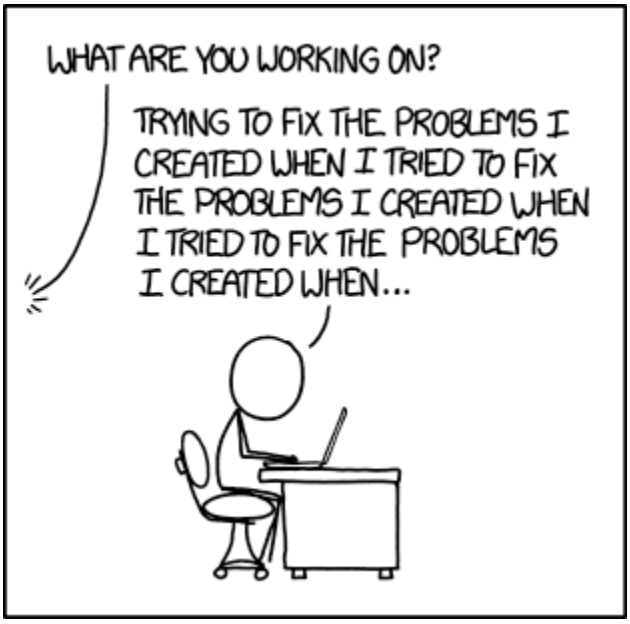
1. What is recursion?

Recursiveness is a fundamental concept in mathematics and programming that implies that in the definition of a notion, reference is made to itself.
VERY IMPORTANT! Within any recursive definition there must be at least one basic (elementary) case, which can be reached after a certain number of steps.



2. Recursive functions

We already know from previous labs that we can define functions that inside themselves can call other functions. We can even define functions that call themselves. Such a function that appears in its own definition is called a recursive function.

Example: Factorial
As an example we will try to calculate the recursive factorial for the number$n$. We start from its recursive relation:

$$factorial(n)=n!=\{n\times(n-1)!1\text{for}n\neq1sauna\neq0\text{for}n=1sauna=0$$

We notice that if we want to calculate the term $n!$, we must first find the term$(n-1)!$
$(n-1)!=(n-1)\times(n-2)!$
To find out the term$(n-1)!$we must first determine the term$(n-2)!$
$(n-2)!=(n-2)\times(n\text{-}3)!$

We notice that this process is repeated, so that to calculate the value of the current term we will always need the previous term, and when we try to calculate this one we need the term that is before it.

By repeating this process we will arrive at a given moment to calculate the values: $4!=4\times3!3!=3\times2!2!=2\times1!$

It is known that $1!=1$and$0!=1$. So now we know everything we need. Since we have arrived at a case that we know (a base case) viz $1!$, we can go back and calculate the$2!,3!$, ...,$(n-1)!$and finally on $n!$.

We will now try to define a function to do all this factorial calculation. As we have established, we want to calculate the factorial of a number n, so this will be the parameter of our function.

```
deffactorial(n):
 # WE WILL COMPLETE WITH CODE

result=factorial(6)
```

Now that we have defined our function header, we need to determine what statements we will write. If we remember what we did at the beginning, to calculate the $n!$ We needed to do$n)\times(n-1)!$. Who is$n-1)!$? We can write it as the result of our function call that receives as an argument$(n-1)!$.

```
deffactorial(n):
 return n*factorial (n-1) # I WROTE THE RECURRENT
RELATION

result=factorial(6)
```

Basically, at the moment, in our function we were able to implement that factorial calculation string that we went through earlier, because our function will call itself to calculate the previous

term. What else is missing? Stop condition (base case). We knew we had to stop when we got to calculating $1!=1$ and $0!=1$. Therefore we should specify within the function that if the parameter n is 1 or 0, the result of the factorial is 1.

```
deffactorial(n):
 ifn==0or n==1: # I DEFINED THE BASE CASE
 return 1
 return n*factorial (n-1) # I WROTE THE RECURRENT
RELATION

result=factorial(6)
```

Example: The sum of the digits of a number

In recursive problems where we are asked to process the digits of a number, we can consider that the number consists either of a single digit (the base case) or of a string of digits (n//10) preceding the last digit (n%10). Based on this recursive definition of a number in base 10 we can very easily think about how we would write a recursive function that calculates the sum of the digits of a number:

$$the\ amount(n)=\{n\%10+the\ amount(n//10)\ nforn>9forn\leq 9$$

For the basic case (when the number consists of a single digit) obviously this amount would be the only digit of the number.

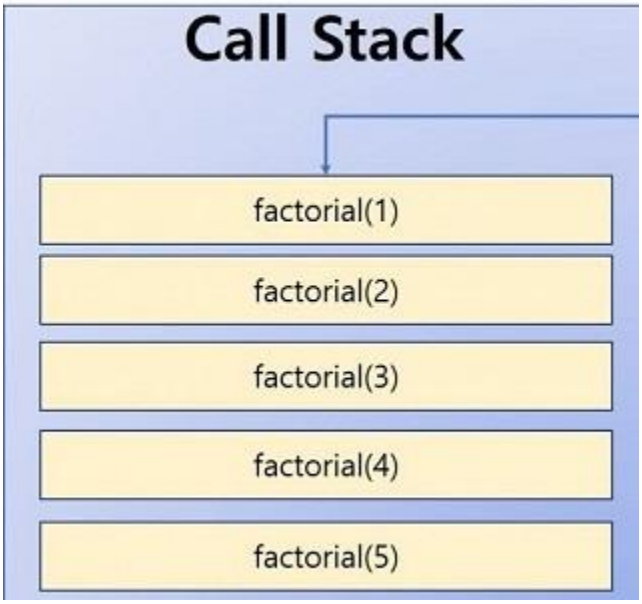
```
def recursive_sum(n):
  ifn <=9:# BASE CASE
 returnn
 else:
 returnn %10+ recursive_sum(n //10)# // DIVIDE AND ROUND
DOWN THE OBTAINED NUMBER

prince(recursive_sum(123456))
```

3. How does recursion work?

When we make a function call in Python, the values of parameters and local variables, as well as the location from where the function call is made, are put on a call stack. The stack is a LIFO (Last In First Out) data structure which means we will only add a new element to the top of the

stack and take an element only from the top of the stack.



For example, the factorial function call: since factorial(1) is the last call added to the stack, it is the first to be popped off the stack and processed. Then, the other calls are popped off the stack and processed: factorial(2), factorial(3), factorial(4), factorial(5).