# Recursive functions

Intro to Python

# Definition

- A recursive function is a function that contains a call to itself

# Example

- The factorial function is defined by:

$$\forall n \in \mathbb{N}^{+}, \qquad n! = \begin{cases} 1 \; if \; n = 0 \\ n \times (n-1)! \; otherwise \end{cases}$$

- This definition is recursive, because we use the definition of factorial to define factorial
- The calculation always terminates, because we remove 1 at each step and will always end up with 0! which is known

# Example

- It is very easy to implement factorial in Python:

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1) # fact(n-1) is the recursive call
6
7  fact(6)
```

```
720
```

```
1  fact = lambda n: 1 if n == 0 else n * fact(n-1)
2  fact(5)
```

```
120
```

# Example 2

- You can do the same for any sequence defined recursively:

$$\begin{cases} u_o = 0.5 \\ \forall\, n \geq 0,\ u_{n+1} = \dfrac{u_n}{2} + \dfrac{1}{u_n} \end{cases}$$

```python
def f(x):
    return x/2 + 1/x

def u(n):
    return 0.5 if n == 0 else f(u(n-1))

u(10)
```

```
1.414213562373095
```

# Stop case

- For a recursive function to stop, it needs a stop case
- Otherwise, it will lead to an infinite recursion
- Unlike infinite loops, infinite recursions tend to stop with an error message
- They stop because the number of function calls that can be stacked is limited

# Stop case Example

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1) # fact(n-1) is the recursive call
6
7  fact(6)
```

This is our stop case

```
720
```

# Stop case How to

- For your recursive functions to always stop, they should always look like:

if stop condition then

stop instruction

else

recursive call

end if

- Rules:
  - Always have a stop case
  - The recursive call must be made on different data, that converges towards the stop case

# Other cases

- Sometimes we don't have a recursive math formula
- We can still use recursive functions to find a solution in many cases!
- The goal is to find an equivalent to the recursive equation for our problem
- Example : inverting a list
  - To invert a list, we can iterate through it in reverse order and store the element in a new list…
  - or we can say that the inverted list is:
    - The inverted list minus its first element
    - To which we add the first element at the end!
    - This is a recursive definition of inverting a list

# Example

```
:
1 def inverser(list1):
2     if len(list1) == 0: # The stop case is the empty list
3         return list1
4     else:
5         return inverser(list1[1:]) + [list1[0]] # Recursive call on a sublist
6 inverser(['a', 'b', 'c'])
```

```
['c', 'b', 'a']
```

# Multiple recursive calls

- Let's look at the Fibonacci sequence:

$$\forall n \in \mathbb{N}^+, F(n) = \begin{cases} 1 \quad if\ n \leq 1 \\ F(n-1) + F(n-2) \quad otherwise \end{cases}$$

```
1  def fibo(n):
2      if n == 0 or n == 1:
3          return 1
4      else:
5          return fibo(n-1) + fib(n-2)
```

# Converting a problem into a recursive problem

Let $a_1, a_2, \ldots, a_n, \ldots$ be a sequence of addable objects. Let us assume the sum of the $n$ primes: $S_n = a_1 + a_2 + \cdots + a_n$.

The following calculation naturally reveals the recursive interpretation of the sum:

$$\begin{aligned} S_n &= a_1 + \cdots + a_{n-1} + a_n \\ &= (a_1 + \cdots + a_{n-1}) + a_n \\ &= S_{n-1} + a_n \end{aligned}$$

The stopping criterion being $S_1 = a_1$.

Example:

```
:
```

```python
def a(k): # a(k) = k
    return k

def S(n): # Sum of k prime integers
    if n == 1:
        return a(1)
    else:
        return S(n-1) + a(n)

S(1), S(2), S(10)
```

```
(1, 3, 55)
```

# Converting a problem into a recursive problem

- When handling lists, it is frequently possible to convert a problem into a recursive problem

The recursive definition of a list can be formulated as follows:

- an empty list (stop criterion)
- [an element] + a list

```
1 list1 = list(range(10)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 print(list1)
3
4 list1 == [list1[0]] + list1[1:]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
True
```

# Terminal recursion

We speak of *terminal recursion* or *terminal recursion function* when the recursive call is the last operation performed in the body of the function.

A terminal recursion is equivalent to an iteration and allows not to saturate the call stack.

Exemple:

```
:
1 def fact_rt(n, acc):
2     if n == 0:
3         return acc
4     else:
5         return fact_rt(n-1, n*acc)
```

```
:
1 fact_rt(5,1)
```

```
120
```

# Conclusion

- Recursive functions are very powerful:
  - They allow us to get the results with less code
  - They often replace complicated loops
- But you must be careful:
  - You need a stop case
  - You must be careful about the arguments of your recursive call
  - With many recursive calls, recursive functions are harder to debug
  - Sometimes using a non-recursive version is more efficient for the computer