Introduction to `python`

Functions and Recursivity

Mauricio Sevilla

`email= j.sevillam@uniandes.edu.co`

25.01.19

We have been working on some structures on `python` such as, variables (`int` , `float` , `str` , `list`), operators (Arithmetic: `+` , `-` , `*` , `/` , `**` , `//` ; Logical: `!` , `and` , `or` ; Comparison: `>` , `<` , `==` , `!=`), control structures (`if` , `for`), so we are ready to learn one of the most useful therefore important structures, the `functions` .

A `function` is a segment of code that usually is going to be used several times during the program, or you want to leave appart so that your code doesn't look messy.

As you have noticed so far, sometimes read code gets very complicated and it is very ofter to have specific sections of the code that gets repeated, here is where functions take place.

In [1]:
```
def function(a,b):
    c=a*b
    return c
```

Here we can see the basic structure, it MUST initiate with `def`, after the name of the function, then in parenthesis the parameters that it function recive and then the `:` and the indentation in the same way we do with `if` or `for`, and at the end we have a `return`.

Parameters and `return` are optional.

To use a function, we write the name and give the parameters,

In [2]:
```
a1=2
b1=3
print(function(a1,b1))
```

```
6
```

Look that the names of the variables are not the same we used when defining the function. They do not have to coincide, in fact it is better if they dont.

One of the advantages of `python` when compared with different programming languages, is that it is possible to use the same function with different data types,

```
In [3]: a2=2
        b2=[1,2,3,4]
        print(function(a2,b2))
```

```
[1, 2, 3, 4, 1, 2, 3, 4]
```

A function can have two different behaviours in terms of the parameters,

> *There are two different ways to give the parameters to functions,*
> *and `python` does this automatically depending on the data type*
>
> - *By value*
> - *By reference*

Let us explore the difference.

Here, we define a function that recieve a value `a` , create a variable and save there `a+1` , then return the value of that varialbe

```
In [4]: def function2(a):
            temporal=a+1
            return temporal
```

```
In [5]: print(a1,function2(a1))
```

```
2 3
```

The variable `temporal` is created locally inside de function, so that outside it doesn't exist

```
In [6]: print(temporal)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-154d002d4153> in <module>
----> 1 print(temporal)

NameError: name 'temporal' is not defined
```

We can save the result of a function on a variable, for example,

```
In [7]: b=function2(a1)
```

```
In [8]: print(b)
```

```
3
```

But, what if we do not use an auxiliar variable?

```
In [9]:  def function3(a):
             a=a+1
             return a
```

```
In [10]: a=2
         print(a,function3(a))
```

```
2 3
```

```
In [11]: print(a)
```

```
2
```

It didn't change!

And if we do not use the return?

In [12]:
```
def function4(a):
    a=a+1
```

In [13]:
```
print(a)
```

```
2
```

In [14]:
```
function4(a)
```

In [15]:
```
print(a,function4(a))
```

```
2 None
```

It does the calculation, but when the function is finished the result is not used.

On the prevous examples, `python` created a copy of the variable and work with the copy without changing the parameter, this is called *Parameter passed by value*.

On the other hand, we have a different behaviour when working with lists

```
In [16]: def function5(list1):
             list1.append(2)
```

```
In [17]: lista=[1]
```

```
In [18]: function5(lista)
```

```
In [19]: print(lista)

         [1, 2]
```

The main difference here is that lists are *passed by value or reference.*

```
In [20]: a=[1,2]
```

```
In [21]: b=a
```

```
In [22]: print(b)
```

```
[1, 2]
```

```
In [23]: b.append(3)
```

```
In [24]: print(a)
```

```
[1, 2, 3]
```

```
In [25]: c=a.copy()

In [26]: print(c)

[1, 2, 3]

In [27]: c.append(4)

In [28]: print(a)

[1, 2, 3]

In [29]: print(c)

[1, 2, 3, 4]
```

Recursivity

Recursivity solves the queston:

> *How can we use a function inside itself?*

It is not always possible, and it has a particular structure.

1. You must have a *minimum* (*maximum*) value of the function.
2. The result must depend on the same function but in a different parameter

For example they can be used to compute,

- *Sum of the smaller integers*
- *Factorial*
- *Fibonacci numbers*
- *Pascal Triangle*

On class we discussed the calculation of the sum of the smaller numbers (`int`), for example for 10

$$10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55$$

if we think this operation as `sum(n)` , we can see this as `n+sum(n-1)` or `n+(n-1)+sum(n-2)` until `n+(n-1)+...+0` , lets program this.

We need to separate two cases, the first one, indicates that the recursion is finished!, and the second implements the recursion

```
In [30]: def sum_n(n):
             if n == 0:
                 return 0
             else:
                 return n + sum_n(n-1)
         print(sum_n(10))
```

```
55
```

Let see a more complex problem,

We are going to program the Sieve of Eratosthenes.

This is an algorithm for finding prime numbers in a given interval.

The method work as follows,

1. List all numbers in the interval,

   $$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.$$

2. Select the first one

   $$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.$$

3. Take out all the numbers that are multiples of the selected number

   $$2, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, 9, \underline{10}, 11, \underline{12}, 13, \underline{14}, 15, \underline{16}, 17, \underline{18}, 19, \underline{20}.$$

   $$2, 3, 5, 7, 9, 11, 13, 15, 17, 19.$$

1. Select the next one,
   $2, 3, 5, 7, \underline{9}, 11, 13, \underline{15}, 17, 19.$

1. Repeating this for any remaining number leads to have only the prime numbers
$$2, 3, 5, 7, 11, 13, 17, 19.$$

In [31]:

```python
import math as m

def sieve(n):
    primes = list(range(2,n+1))
    maxi = m.sqrt(n)
    num = 2
    while num < maxi:
        i = num
        while i <= n:
            i += num
            if i in primes:
                primes.remove(i)
        for j in primes:
            if j > num:
                num = j
                break
    return primes
data=sieve(100)
print(data)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97]
```