What do we already know?In the previous lab we established what functional programming entails and learned to define simple functions in Python.

Objective: This week we'll delve into the topic of functions and argue why Python supports the functional programming paradigm.

Python treats functions like any value (int, float, ...) you've encountered so far. You can for example assign a function to a variable:

```
def   sum(a, b):
      return a + b

sum_numbers = sum
print(sum_numbers(2, 3))
```

If we want to see the type of the variable defined above we can write:

```
a = -3
print(type(a))
print(type(sum_numbers))          # the function from the previous example
```

Notice the type of the number_sum variable.

Python and functional programming

To facilitate writing programs according to the functional programming paradigm, a programming language should provide the following two mechanisms:

- can receive a function as a parameter
- be able to return a function (to the entity that called it)

Since we saw above how Python treats functions, the above two requirements are successfully provided by Python.

Functions as a parameter

A function can in turn be passed as a parameter to another function. For example, we can define our multiply_5 function, which takes a function and a number as parameters. We also define a function

simple function called increment_1, which takes a parameter and returns it incremented by 1. Now we can call the function multiply_5 with the parameters increment and a number, for example 5.

```
def   increment_1(x):
      return x + 1
print(increment_1(5))

def   multiply_5(f, x):
      return f(x) * 5

print(multiply_5(increment_1, 5))
```

Note: function multiply_5 can be called with any parameter for f, and x. However, if we called it with two integer values, we would have a runtime error. That's why it's important to know what parameters the function we're calling expects.

Anonymous functions

The notation`lambda` *argument*`:` *phrase* defines in Python an anonymous function (lambda function). This is a *phrase* of type function and can therefore be used in other expressions. We can evaluate directly without having to give the function a name first. This simple example illustrates that in Python, a function (here `lambda x : x + 3`) can be used as simply as any other value.

```
> > > (lambda x : x + 3)(2) 5

> > > (lambda x, y, z: x + y + z)(1, 2, 3) 6
```

Returning to the notation `def`, it is equivalent to define:

```
> > >  def f(x):
. . .          return x + 3
```

or

```
> > >  def f(x):
. . .          return (lambda x : x + 3)(x)
```

This way of defining a function doesn't make much sense, because inside the function f we define an anonymous function that we call as soon as we define it. Conversely, anonymous functions can be useful if we want to return a function. For example, we can define a function that returns an increment function by 1 if the received parameter is True or a decrement function by 1 if the received parameter is False. In this case the anonymous functions are

useful.

```
def   return_function_incrementare(x): if x:
            return lambda x: x + 1 else:
            return lambda x: x - 1

Inc.  =  return_increment_function(True)
December  =  return_increment_function(False)

print(inc(5))
print(dec(5))
```

It can be seen that the result of calling the function return_function_incrementare is saved in the variables Inc., respectively December. These two variables are themselves functions, so they can be called.

As we saw in the previous subsection, functions can also be passed as a parameter. Instead of a previously defined function, we can pass an anonymous function as a parameter. Thus, the example with multiply_5 becomes:

```
def   multiply_5(f, x):
      return f(x) * 5
print(multiply_5((lambda x: x + 1), 5))
```

Composition of functions

One of the simplest and most widely used operations with functions is *UP* them, in this way we can easily obtain complex functions (processing) starting from simple functions. In mathematics, if $f: A \to B$ and $g : C \to A$, the composition $f \circ g : C \to B$ it is defined by the relationship $(f \circ g)(x) = f(g(x))$. So, starting from a value $x \in C$ a value is obtained $g(x) \in A$, and then by applying it f the value $f(g(x)) \in B$.

in Python we can define a composition function with two parameters, which returns another function, representing the function f composed with the function g.

```
def   comp(f,g):
      return lambda x : f(g(x))

def   f1(x):
      return x * 2
def   f2(x):
      return x + 1

f = comp(f1, f2)
print(f(6))
f = comp(f2, f1)
```

```
print(f(6))
```

Calling the comp function with parameters f1 and f2, which are themselves functions, gives us another function. Of course, the order in which we give the parameters matters, so the result of the first composition is a different function than the result of the second composition. This can also be seen by the fact that the results of calling place with the same parameter.

Repeated application (composing a function with itself)

In particular, if a function has the same scope of definition and values, it can be composed with itself: $f \circ f$, where $f : A \to A$. Starting from the composing function `comp` we can define a higher-order function that composes a function (given as a parameter) with itself.

```
def   appl2(f):
      return comp(f, f)

f = appl2(f1)
print(f(6))
```

In the same way, we can compose the function `appl2` with itself, yielding a function that applies 4 times the function received as a parameter:

```
appl4 = comp(appl2, appl2)

f = appl4(f1)
print(f(6))
```

However, we can also define another function:

```
def   appl4(f):
      return appl2(appl2(f))

f = appl4(f1)
print(f(6))
```

In this case, the first `appl2` produces the application of the function given as a parameter twice, that's all `appl2`. Trying with multiple functions and integer arguments, we get equal values, suggesting that in the two variants we have in fact defined the same function.

Operators are functions

To have access to the operators, we must import the module *operator. Then we can use an operator as follows:*

```
import operator
>>> operator.add(2,3)
5
>>>2 + 3
5
```

Type your text

For more details consult the link(https://www.geeksforgeeks.org/operator-functions-in-python-set-1/).

We can define a generic operations function, which takes as parameters a function representing the operation we want it to perform and two numeric values. The function is defined and called as follows:

```
def generic_operation(op, x, y):
    return op(x, y)

print(generic_operation(operator.add, 3, 4))
```

Functions with default parameters

Python allows us to declare functions with parameters that have a default value. Such functions can also be called normally, but also without giving a value to the default parameter. In this case, the parameter will be equal to the default value. Below you can see an example of a function that has a default parameter.

```
def increment(x = 0):
    return x + 1

print(increment(4))
print(increment())
```

If we have a function with several parameters, some of which are defaults, it is important that the default parameters are at the end of the parameter list. For example, we cannot have a function with the first parameter implicit and the second explicit. An example with more parameters can be seen below. It is advisable to run it to understand how explicit parameters work in multi-parameter functions.

```
def print_params(a, x = "a", y = "b", z = "c"):
    print(a, x, y, z)

print_params("aaa")
print_params("aaa",          "x")
print_params("aaa",          "y")
print_params("aaa", y = "x")
print_params("1", "2", "3", "4")
```

This code prints the following lines:

```
aaa abc
aaa xbc
aaa ybc
aaaaa
1 2 3 4
```