# L9b: Advanced data structures

## Generators and functional paradigm

# Iterators

`for` is used to traverse **iterable** objects (lists, strings…)

```
x=[1, 2, 3, 4, 5]
for i in range(0, len(x)):
	print(x[i])

x=[1, 2, 3, 4, 5]
for i in x:
	print(i)

def sum(x):
	res = 0
	for i in range(0, len(x)):
		res += x[i]
	return(res)

def sum(x):
	res = 0
	for i in x:
		res += i
	return(res)
```

```
x= 'hola!'
for i in range(0, len(x)):
	print(x[i])

x= 'hola!'
for i in x:
	print(i)
```

# Iterators

```
x = ['a', 'b', 'c', 'd', 'e’]

for i in range(0, len(x)):
      print(i)


for i in range(0, len(x)):
      print(x[i])


for i in x:
      print(i)
```

```
0
1
2
3
4

a
b
c
d
e
```

# Enumerate

- If we need both the **indices** and the **values**, we can use the **enumerate** function, which traverses an **iterable** object and returns 2 elements: the indices and the corresponding values
- We will have to use 2 variables in the `for`

```
x = ['a', 'b', 'c’]

for i, j in enumerate(x):
        print(i, j, x[i])

Output:
0 a a
1 b b
2 c c
```

# Zip

- To operate with several variables within a for loop, we can use the **zip** function, which will allow to traverse several iterable objects at once

```
x = ['a', 'b', 'c', 'd', 'e']
y = ['a', 'e', 'i', 'o', 'u’]

for i, j in zip(x,y):
      print(i,j)
```

```
a a
b e
c i
d o
e u
```

```
x = ['a', 'b', 'c', 'd', 'e']
y = ['a', 'e', 'i’]

for i, j in zip(x,y):
      print(i,j)
```

```
a a
b e
c i
```

If the iterable objects have different size, the `for` will end at the element from the shortest iterable object

# Zip

- The **zip** function can traverse more than 2 iterable objects

```
x = ['a', 'b', 'c', 'd', 'e']
y = ['a', 'e', 'i']
z = [1, 2, 3, 4, 5, 6, 7, 8, 9]

for i, j, k in zip(x,y,z):
       print(i,j,k)

a a 1
b e 2
c i 3
```

# Generators: Yield

The **yield** operator can be used to define functions that are **iterable:**

- We can define functions that, at each new iteration in a loop (e.g. *for*), it returns the next element, up to a stop condition.

```
def my_range(n):
        num = 0
        while num < n:
                yield num
                num += 1
```

```
for x in my_range(10):
        print(x)
```

The first time it is called, it returns 0, the second time it is called, it returns 1, then 2, later 3… up to 9

# Generators: Yield

- The **yield** operator returns the value, then it pauses the execution, and later, when the function is called again, it resumes the execution back (continuing from the instruction it was paused).

```
def myfactorial(x):
    yield 1   # because 0! is 1
    a=1
    b=1
    while b<x:
        a=a*b
        b +=1
        yield a


for x in myfactorial(5):
       print(x)
```

The first time it is called, it returns 0!
the second time, it returns 1!,
then 2!, later 3!… up to 4!

Equivalent code:

```
mf= myfactorial(5)

print(next(mf))
print(next(mf))
print(next(mf))
print(next(mf))
print(next(mf))
```

# Functional paradigm: Map

**Map** applies a function to each element in a list. It allows to make operations in a more elegant way

```
map(function_to_apply, list_of_elements)
```

Example: we want a list L2, with the square of each element in L

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

def square(x):
       return(x**2)

L2=[]
for i in L:
       L2.append(square(i))
```

- Is equivalent to:

```
L2 = list(map(square, L))
```

# Functional paradigm: Map

**Map** can take more than one iterable element

```
map(function_to_apply, iterable1, iterable2, …)
```

Example: the list L2 will have the sum of elements in three lists:

```
def f(a,b,c):
       return a+b+c

L2 = list(map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300]))
Output → [111, 222, 333]

L3 = list(map(f, [1, 2, 3], [10, 20, 30], [100, 200]))
Output → [111, 222]
```

If the iterable objects have different size, it will end at the element from the shortest iterable object

# Functional paradigm: Lambda

- But let us imagine that we only need the square function to get the square of L. We would not use it again.
- So, instead of defining the square function, we can use **lambda** functions. They are simple anonymous functions, (functions without a name).
- They are typically used with **map**

```
L2 = list(map(square, L))
```

- Is equivalent to:

Output: the square of each element

```
L2 = list(map(lambda x:x**2,L))
```

Input parameter: each element in L

# Functional paradigm: Map with list of functions

```
def mult(x):
    return(x*x)
```

```
def suma(x):
    return(x+x)
```

```
def par(x):
    return(x%2 == 0)
```

Output: execution of each function

```
myfunctions=[mult,suma,par]
for i in range(10):
       print(list(map(lambda x:x(i), myfunctions)))
```

Input parameter: each one of the functions

```
Output:
i=0 → [0, 0, True]
i=1 → [1, 2, False]
i=2 → [4, 4, True]
i=3 → [9, 6, False]
i=4 → [16, 8, True]
i=5 → [25, 10, False]
i=6 → [36, 12, True]
i=7 → [49, 14, False]
i=8 → [64, 16, True]
i=9 → [81, 18, False]
```

# Functional paradigm: Map with list of functions

```
def mult2(x,y):
    return(x*x)
```

```
def suma2(x,y):
    return(x+y)
```

```
myfunctions=[mult2,suma2]
for i in range(5):
      print(list(map(lambda x:x(i,i+3), myfunctions)))
```

Output: execution of each function

The parameters for the function

Input parameter: each one of the functions

```
Output:
i=0 → [0, 3]
i=1 → [4, 5]
i=2 → [10, 7]
i=3 → [18, 9]
i=4 → [28, 11]
```

# Functional paradigm: Filter

**Filter** can filter elements in an elegant way. It applies a function that returns Booleans, element by element over a list. **Filter** takes only those elements where the function has returned *True*

```
filter(function_returning_boolean, list_of_elements)
```

Example: L2 will contain the even numbers in L:

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

def par(x):
   return(x%2 == 0)

L2=[]
for i in L:
   if par(i):
      L2.append(i)
```

Equivalent code:

```
L2 = list(filter(par, L))
```

Or

```
L2 = list(filter(lambda x:x%2==0, L))
```

# Functional paradigm: Example

Example: Implement a code with *map*, *filter* and *lambda* that, given a list of integers, returns the square root of those nums >=0

```
L=[1, 9, -1, -4, 16, -2, 4]  → output: [1.0, 3.0, 4.0, 2.0]
```

```
list(map(lambda x:x**.5,filter(lambda x:x>=0,L)))
```

Equivalent code

LIST COMPREHENSION (reminder):

```
L2 = [x**.5 for x in L if x>=0]
```

# Functional paradigm: Example

Example: Implement a code with *map*, *filter* and *lambda* that, given a list of strings, returns those in uppercase

```
A = ["cat", "Cat", "CAT", "dog", "Dog", "DOG", "emu", "Emu", "EMU"]
```

```
def all_caps(s):
        return s.isupper()

list(filter(all_caps, A))
```

output-> ['CAT', 'DOG', 'EMU']

Equivalent code

```
list(filter(lambda s:s.isupper(),A))
```

output -> ['CAT', 'DOG', 'EMU']

Equivalent code

LIST COMPREHENSION (reminder): `[s for s in A if s.isupper()]`