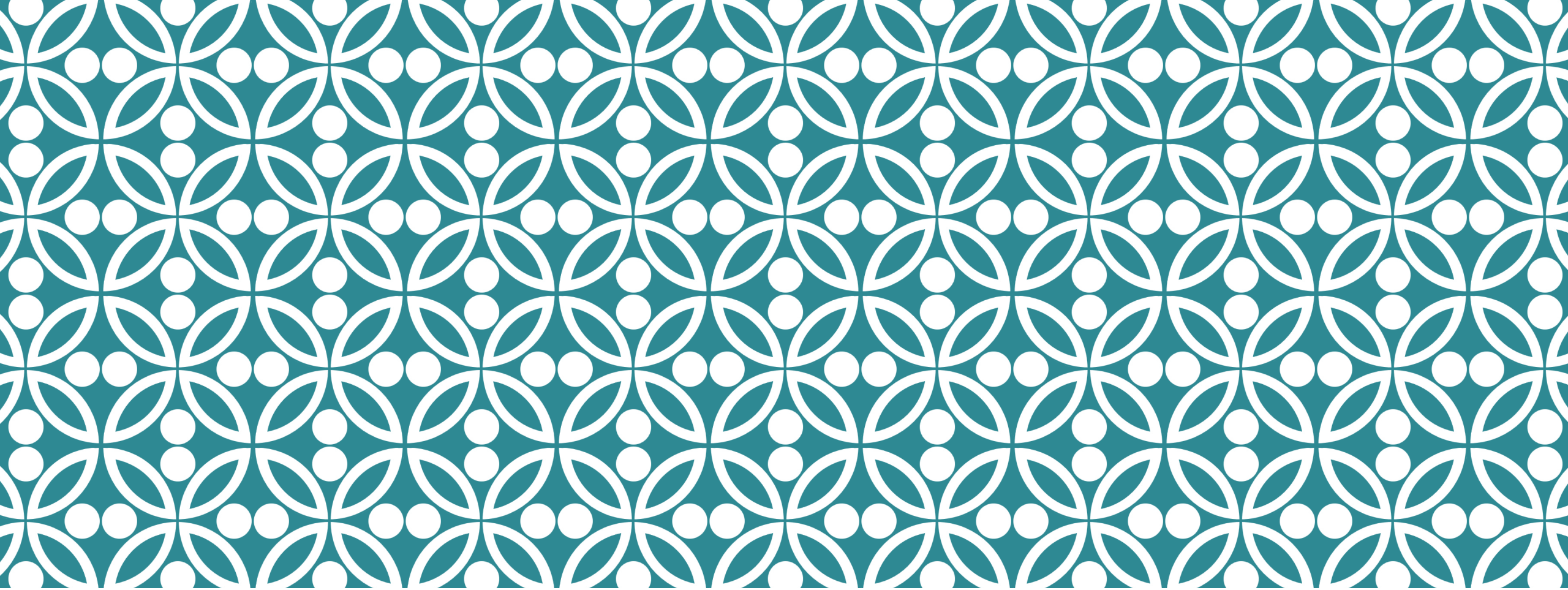

PROGRAMMING IN PYTHON

Gavrilut Dragos
Course 2

LAMBDA FUNCTIONS

A lambda function is a function without any name. It has multiple roles (for example it is often use as a pointer to function equivalent when dealing with other functions that expect a callback).

Lambdas are useful to implement closures.

A lambda function is defined in the following way:

```
lambda <list_of_parameters> : return_value
```

The following example uses lambda to define a simple addition function

Python 3.x(without lambda)	Python 3.x(with lambda)

`def addition (x,y):`
`  return x+y`
`print (addition (3,5))`
	`addition = lambda x,y: x+y`
	`print (addition(3,5))`

LAMBDA FUNCTIONS

Lambdas are bind during the run-time. This mean that a lambda with a specific behavior can be build at the run-time using the data dynamically generated.

Python 3.x

```
def CreateDivizibleCheckFunction(n):
        return lambda x: x%n==0
fnDiv2 = CreateDivizibleCheckFunction (2)
fnDiv7 = CreateDivizibleCheckFunction (7)
x = 14
print ( x, fnDiv2(x), fnDiv7(x) )
```

Output

```
14 True True
```

In this case fnDiv2 and fnDiv7 are dynamically generated.

This programming paradigm is called **closure**.

SEQUENCES

A sequence in python is a data structure represented by a vector of elements that don't need to be of the same type.

Lists have two representation in python:

- **list** ➔mutable vector (elements from that list can be added, deleted, etc). List can be defined using […] operator or the **list** keyword
- **tuple** ➔ immutable vector (the closest equivalent is a constant list) ➔ addition, deletion, etc operation can not be used on this type of object. A tuple is usually defined using (…) or by using the **tuple** keyword

list and **tuple** keywords can also be used to initialized a tuple or list from another list of tuple

SEQUENCES

Python 3.x

```
x = list ()              #x is an empty list
x = []                   #x is an empty list
x = [10,20,"test"]       #x is list
x = [10,]                #x is list containing [10]
x = [1,2] * 5            #x is list containing [1,2,  1,2,  1,2,  1,2,  1,2]
x,y = [1,2]              #x is 1 and y is 2
```

```
x = tuple ()             #x is an empty tuple
x = ()                   #x is an empty tuple
x = (10,20,"test")       #x is a tuple
x = 10,20,"test"         #x is a tuple
x = (10,)                #x is tuple containing (10)
x = (1,2) * 5            #x is tuple containing (1,2,  1,2,  1,2,  1,2,  1,2)
x = 1,2 * 5              #x is tuple containing (1,10)
x,y = (1,2)              #x is 1 and y is 2 (the same happens for x,y = 1,2 )
```

SEQUENCES

Elements from a list can be accessed in the following way

Python 3.x

```
x = ['A', 'B', 2, 3, 'C']

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ['A', 'B', 2]
x[3:]     #Result is [3, 'C']
x[1:3]    #Result is ['B', 2]
x[1:-3]   #Result is ['B']
```

SEQUENCES

Elements from a tuple can be accessed in the same way

Python 3.x

```
x = ('A', 'B', 2, 3, 'C')

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ('A', 'B', 2)
x[3:]     #Result is (3, 'C')
x[1:3]    #Result is ('B', 2)
x[1:-3]   #Result is ('B')
```

SEQUENCES

tuple and **list** keywords can also be used to convert a tuple to a list and vice-versa.

Python 3.x

```
x = ('A', 'B', 2, 3, 'C')
y = list (x) #y = ['A', 'B', 2, 3, 'C']
```

```
x = ['A', 'B', 2, 3, 'C']
y = tuple (x) #y = ('A', 'B', 2, 3, 'C')
```

Both lists and tuples can be concatenated, **but not with each other**.

Python 3.x

```
x = ('A', 2)
y = ('B', 3)
z = x + y
#z = ('A', 2, 'B', 3)
```

```
x = ['A', 2]
y = ['B', 3]
z = x + y
#z = ['A', 2, 'B', 3]
```

```
x = ('A', 2)
y = ['B', 3]
z = x + y
#!!! Error !!!
```

SEQUENCES

Tuples are also used to return multiple values from a function.

The following example computes both the sum and product of a sequence of numbers

Python 3.x

```
def ComputeSumAndProduct(*list_of_numbers):
        s = 0
        p = 1
        for i in list_of_numbers:
                s += i
                p *= i
        return (s,p)

suma,produs = ComputeSumAndProduct(1,2,3,4,5)
#suma =15, produs = 120
```

SEQUENCES

tuple and **list** can also be organized in matrixes:

Python 3.x

```
x = ((1,2,3), (4,5,6))
x = ([1,2,3], (4,5,6)) #matrix sub components don't have to be of the
                       #same type
x = ( ((1,2,3), (4,5,6)), ((7,8), (9,10,11, 12)) )
#a matrix does not have to have the same number of elements on each
#dimension
```

```
#the same rules from tuples apply to lists as well
x = [[1,2,3], [4,5,6]]
x = [[1,2,3], (4,5,6)]
```

SEQUENCES

Both **tuples** and **lists** can be enumerated with a **for** keyword:

Python 3.x

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

Python 3.x

```
for i in (1,2,3,4,5):
        print(i)
```

Output

```
1
2
3
4
5
```

Lists and tuples have a special keyword (**len**) that can be use to find out the size of a list/tuple:

Python 3.x

```
x = [1,2,3,4,5]
y = (10,20,300)
print (len(x), len(y))
```

Output 3.x

```
5 3
```

SEQUENCES

One can also use the **enumerate** keyword to enumerate a list and get the index of the item at the same time:

Python 3.x

```
for index,name in enumerate(["Dragos","Mihai","Nicu","Vlad"]):
        print("Index:%d => %s"%(index,name))
```

Or use an external variable:

Python 3.x

```
index = 0
for name in ["Dragos","Mihai","Nicu","Vlad"]):
        print("Index:%d => %s"%(index,name))
        index += 1
```

Output

```
Index:0 => Dragos
Index:1 => Mihai
Index:2 => Nicu
Index:3 => Vlad
```

SEQUENCES

enumerate functions also allows a second parameter to specify the index base (default is 0 → just like in C-like langueages).

Python 3.x

```
for index,name in enumerate(["Dragos","Mihai","Nicu","Vlad"], 2 ):
        print("Index:%d => %s"%(index,name))
```

In this example, the index base will be 2:

- Dragos (the first name) will have index 2
- Mihai (the second name) will have index 3
- And so on …

Output

```
Index:2 => Dragos
Index:3 => Mihai
Index:4 => Nicu
Index:5 => Vlad
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of numbers from 1 to 9

Python 3.x

```
x = [i for i in range(1,10)] #x = [1,2,3,4,5,6,7,8,9]
```

- A list of all divisor of 23 smaller than 100

Python 3.x

```
x = [i for i in range(1,100) if i % 23 == 0] #x = [23, 46, 69, 92]
```

- A list of all square values for number from 1 to 5

Python 3.x

```
x = [i*i for i in range(1,6)] #x = [1, 4, 9, 16, 25]
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of pairs of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 3.x

```
x=[[x, y] for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [[1, 6], [2, 5], [3, 4], [4, 3], [5, 2], [5, 9], [6, 1],
#     [6, 8], [7, 7], [8, 6], [9, 5]]
```

- A list of tuples of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 3.x

```
x=[(x, y) for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (5, 9), (6, 1),
#     (6, 8), (7, 7), (8, 6), (9, 5)]
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of prime numbers that a smaller than 100

Python 3.x

```
x=[x for x in range(2,100) if len([y for y in range(2,x//2+1) if x % y==0])==0]
#x = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
      59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Using functional programming in Python drastically reduces the size of code. However, depending on how large the expression is to build a list, functional programming may not be advisable if your aim is readability.

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Add a new element in the list (either use the member function(method) **append** or the operator +=). To add lists or tuples use **extend** method

Python 3.x

```
x = [1,2,3]           #x = [1, 2, 3]
x.append(4)           #x = [1, 2, 3, 4]
x+=[5]                #x = [1, 2, 3, 4, 5]
x+=[6,7]              #x = [1, 2, 3, 4, 5, 6, 7]
x+=(8,9,10)           #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x[len(x):] = [11]     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
x.extend([12,13])     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
x.extend((14,15))     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                      #     14,15]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieved by using some operators.

- Insert a new element in the list using member function(method) **insert**

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
x.insert(1,"A")             #x = [1, "A", 2, 3]
x.insert(-1,"B")            #x = [1, "A", 2, "B", 3]
x.insert(len(x),"C")        #x = [1, "A", 2, "B", 3, "C"]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Insert a new element or multiple elements can be done using [:] operator. Similarly [] operator can be used to change the value of one element

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x[2] = 20                #x = [1, 2, 20, 4, 5]
x[3:] = ["A","B","C"] #x = [1, 2, 20, "A", "B", "C"]
x[:4] = [10]             #x = [10, "B", "C"]
x[1:3] = ['x','y','z'] #x = [10, "x", "y", "z"]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Remove an element in the list ➔ using member function(method) **remove**. This method removes the first element with a given value

.

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
x.remove(1)                 #x = [2, 3]
x.remove(100)               #!!! ERROR !!! - 100 is not a value from x
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To remove an element from a specific position the **del** keyword can be used.

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2]                 #x = [1, 2, 4, 5]
del x[-1]                #x = [1, 2, 4]
del x[0]                 #x = [2, 4]
del x[1000]              #!!! ERROR !!! - 1000 is not a valid index
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[4:]                #x = [1, 2, 3, 4]
del x[:2]                #x = [3, 4]
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2:4]               #x = [1, 2, 5]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To **pop** method can be use to remove an element from a desire position an return it. This method can be use without any parameter (and in this case it refers to the last element)

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
y = x.pop(2)             #x = [1, 2, 4, 5]      y = 3
y = x.pop(0)             #x = [2, 4, 5]         y = 1
y = x.pop(-1)            #x = [2, 4]            y = 5
y = x.pop()              #x = [2]               y = 4
y = x.pop(1000)          #!!! ERROR !!! - 1000 is not a valid index
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To clear the entire list the **del** command can be used

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[:]                 #x = []
```

- Python 3.x also has a method **clear** that can be used to clear an entire list

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x.clear()                #x = []
```

LISTS

Be aware that using the operator (=) does not make a copy but only a reference of a list.

Python 3.x

```
x = [1,2,3]
y = x
y.append(10)
#x = [1,2,3,10]
#y = [1,2,3,10]
```

If you want to make a copy of a list, use the **list** keyword:

Python 3.x

```
x = [1,2,3]
y = list (x)
y.append(10)
#x = [1,2,3]
#y = [1,2,3,10]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Python 3.x also has a method **copy** that can be used to create a shallow copy of a list

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
b = x.copy()                #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                    #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

- The operator [:] can also be use to achieve the same result

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
b = x[:]                    #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                    #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **index** method to find out the position of a specific element in a list

Python 3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = x.index("C")      #y = 2
y = x.index("Y")      #!!! ERROR !!! - ”Y” is not part of list x
```

- The operator **in** can be used to check if an element exists in the list

Python 3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = "C" in x          #y = True
y = "Y" in x          #y = False
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **count** method to find out how many elements of a specific value exists in a list

Python 3.x

```
x = [1,2,3,2,5,3,1,2,4,2] #x = [1,2,3,2,5,3,1,2,4,2]
y = x.count(2)            #y = 4 [1,2,3,2,5,3,1,2,4,2]
y = x.count(0)            #y = 0
```

- The **reverse** method can be used to reverse the elements order from a list

Python 3.x

```
x = [1,2,3]               #x = [1,2,3]
x.reverse ()              #x = [3,2,1]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **sort** method to sort elements from the list

sort (*key=None, reverse=False*)

Python 3.x (version 3.7.4 → sort algorithm might be different from one version to another)

```
x = [2,1,4,3,5]
x.sort()                                        #x = [1,2,3,4,5]
x.sort(reverse=True)                            #x = [5,4,3,2,1]
x.sort(key = lambda i: i%3)                     #x = [3,4,1,2,5]
x.sort(key = lambda i: i%3,reverse=True)        #x = [5,2,4,1,3]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **map** to create a new list where each element is obtained based on the lambda expression provided.

map (*function, iterableElement*$_1$*, [iterableElement*$_2$*,... iterableElement*$_n$*]*)

Python 3.x

```
x = [1,2,3,4,5]
y = list(map(lambda element: element*element,x))   #y = [1,4,9,16,25]
```

```
x = [1,2,3]
y = [4,5,6]
z = list(map(lambda e1,e2: e1+e2,x,y))             #z = [5,7,9]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- **map** function returns an iterable objetc in Python 3.x

Python

```
x = [1,2,3]
y = map(lambda element: element*element,x)
#y = iterable object  →Python 3.x
```

- to create a list from an iterable object, use the **list** keyword

Python

```
x = [1,2,3]
y = [4,5,6,7]
z = list(map(lambda e1,e2: e1+e2,x,y))   #z = [5,7,9] → Python 3.x
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **filter** to create a new list where each element is filtered based on the lambda expression provided.

Filter (*function, iterableElement*)

Python 3.x

```
x = [1,2,3,4,5]
y = list(filter(lambda element: element%2==0,x))   #y = [2,4]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Both **filter** and **map** can also be used to create a list (usually in conjunction with **range** keyword)

Python 3.x

```
x = list(map(lambda x: x*x, range(1,10)))
#x = [1, 4, 9, 16, 25, 36, 49, 64, 81]

x = list(filter(lambda x: x%7==1,range(1,100)))
#x = [1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **min** and **max** functions to find out the biggest/smallest element from an iterable list based on the lambda expression provided.

max (*iterableElement, [key]*) **max** (el_1, el_2, … *[key]*)	**min** (*iterableElement, [key]*) **min** (el_1, el_2, … *[key]*)

Python 3.x

```
x = [1,2,3,4,5]
y = max (x)                          #y = 5
y = max (1,3,2,7,9,3,5)              #y = 9
y = max (x,key = lambda i: i % 3)    #y = 2
```

- If you want to use a **key** for max and/or min function, be sure that you added with the parameter name decoration: key = <function>, and not just the key_function or a lambda.

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sum** to add all elements from an iterable object. Elements from the iterable objects should allow the possibility of addition with other elements.

sum (iterableElement, *[startValue]*)

- *startValue* represent the value from where to start summing the elements. Default is 0

Python 3.x

```
x = [1,2,3,4,5]
y = sum (x)                    #y = 15
y = sum (x,100)                #y = 115 (100+15)
```

```
x = [1,2,"3",4,5]
y = sum (x)                    #ERROR→ Can't add int and string
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sorted** to sort the element from a list (iterable object). The key in this case represents a compare function between two elements of the iterable object.

sorted (iterableElement, *[key],[reverse]*)

- The *reverse* parameter if not specified is considered to be False

Python 3.x

```
x = [2,1,4,3,5]
y = sorted (x)                                  #y = [1,2,3,4,5]
y = sorted (x,reverse=True)                     #y = [5,4,3,2,1]
y = sorted (x,key = lambda i: i%3)              #y = [3,1,4,2,5]
y = sorted (x,key = lambda i: i%3,reverse=True) #y = [2,5,1,4,3]
```

- Just like in the precedent case, you have to use the optional parameter with their name

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **reversed** to reverse the element from a list (iterable object).

Python 3.x

```
x = [2,1,4,3,5]
y = list (reversed(x))                    #y = [5,3,4,1,2]
```

- Use **any** and **all** to check if at least one or all elements from a list (iterable objects) can be evaluated to true.

Python 3.x

```
x = [2,1,0,3,5]
y = any(x)     #y = True, all numbers except 0 are evaluated to True
y = all(x)     #y = False, 0 is evaluated to False
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **zip** to group 2 or more iterable objects into one iterable object

Python 3.x

```
x = [1,2,3]
y = [10,20,30]
z = list(zip(x,y))   #z = [(1,10) , (2,20) , (3,30)]
```

- Use **zip** with * character to unzip such a list. The unzip variables are tuples

Python 3.x

```
x = [(1,2) , (3,4) , (5,6)]
a,b = zip(*x)          #a = (1,3,5) and b = (2,4,6)
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **del** to delete a list or a tuple

Python 3.x

```
x = [1,2,3]
del x
print (x)      #!!!ERROR!!! x no longer exists
```