Programming in Python

GAVRILUT DRAGOS

COURSE 2

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 2.x / 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) )
```

In this case fnDiv2 and fnDiv7 are dynamically generated.

This programming paradigm is called **closure**.

Output

```
14 True True
```

Sequences

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

Python 2.x / 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 2.x / 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 2.x / 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
```

Lists and functional programming

A list can also be build using functional programming.

- A list of numbers from 1 to 9

Python 2.x / 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 2.x / 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 2.x / 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 2.x / 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 2.x / 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

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 2.x / 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 2.x / 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.

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

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

Python 2.x/3.x

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

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 2.x/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 a list in Python 2.x and an iterable parameter in Python 3.x

Python

```
x = [1,2,3]
y = map(lambda element: element*element,x)
#y = [1,4,9]              →Python 2.x
#y = iterable object      →Python 3.x
```

- **map** function returns a list in Python 2.x and an iterable parameter in Python 3.x

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
                                         #!!! ERROR on→ Python 2.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:

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

Python 2.x/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]
```

- Python 2.x had another function (**reduce**) that was removed from 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 **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 2.x/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 **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 2.x/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 2.x/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 2.x/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 2.x/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 2.x/3.x

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