

Python Fundamentals

day 13

Today's Agenda

- General use of lambda functions
- filter()
- reduce()
- map()
- Other examples



General use of lambda functions

Lambda functions are used whenever the following built-in functions are used

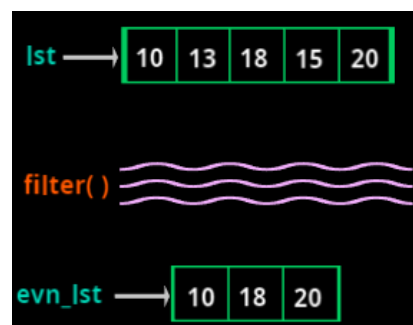
- ❖ filter()
- ❖ reduce()
- ❖ map()

filter()

The **filter()** method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax:

filter (function, sequence)



Let us try to understand with the normal function declaration using **def** and then compare with **lambda** function

```
test.py
lst = [10,13,18,15,20]

def fun(x):
    if x%2 == 0:
        return True
    else:
        return False

evn_lst = list(filter(fun,lst))
print(evn_lst)
```

```
Output Window
In [3]: runfile('C:/python/test.py', wdir='C:/python')
<filter object at 0x000002486F451D48>

In [4]: runfile('C:/python/test.py', wdir='C:/python')
[10, 18, 20]
```

To each element in the list **lst**, **fun** function is applied and checks if the result is true or false. All the true resulted elements are collected in a new list **evn_lst** and displayed. In the output we can see that **evn_lst** is a filter object, to get the list we have to type cast it to list. And the new output displays the filtered list **evn_lst**.

Let us now see how the code changes when we use **lambda** function

```
test.py
lst = [10,13,18,15,20]

'''def fun(x):
    if x%2 == 0:
        return True
    else:
        return False

evn_lst = List(filter(fun,lst))
print(evn_lst) '''

result = list(filter(lambda x: (x%2 == 0),lst))
print(result)
```

```
Output Window
In [7]: runfile('C:/python/test.py', wdir='C:/python')
[10, 18, 20]
```

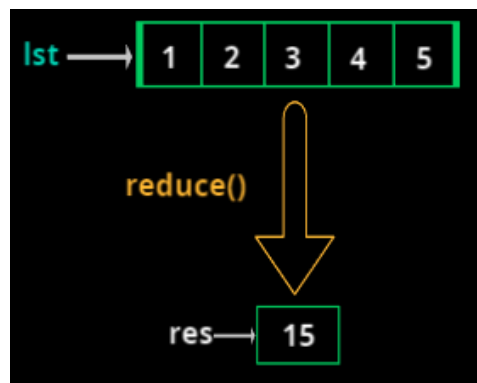
We can see how a six line code has been reduced to a single line code with less complexity.

reduce()

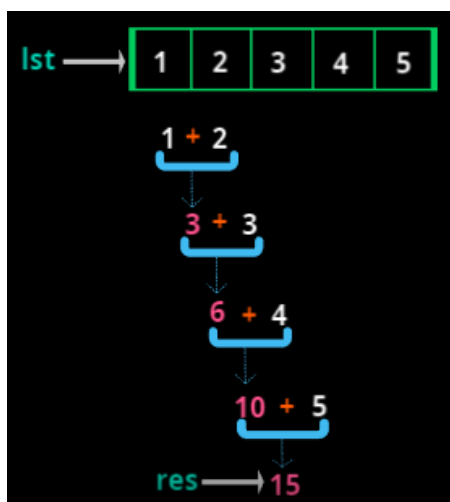
The `reduce()` function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “**functools**” module.

Syntax:

`reduce (function,sequence)`



In the above diagram we are trying to reduce the list `lst` by adding all the elements and storing the result in `res`.



Let us see how to use `reduce()` with normal `def` function before comparing with `lambda` function

```
test.py
from functools import reduce

lst = [1,2,3,4,5]

def fun(x,y):
    return x+y

res = reduce(fun,lst)
print(res)
```

lst →

1	2	3	4	5
---	---	---	---	---

reduce() ↓

res

15

Output Window
In [3]: runfile('C:/python/test.py', wdir='C:/python')
15

Using **def** function we had to write the logic in 4 lines let us see how the code changes with the use of **lambda** function

```
test.py
from functools import reduce

lst = [1,2,3,4,5]

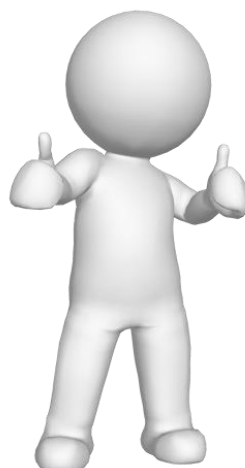
'''def fun(x,y):
    return x+y

res = reduce(fun,lst)
print(res) '''

result = reduce(lambda x,y: x+y,lst)
print(result) |
```

Output Window
In [3]: runfile('C:/python/test.py', wdir='C:/python')
15

We can see that the code reduced to a single line statement with less complexity.



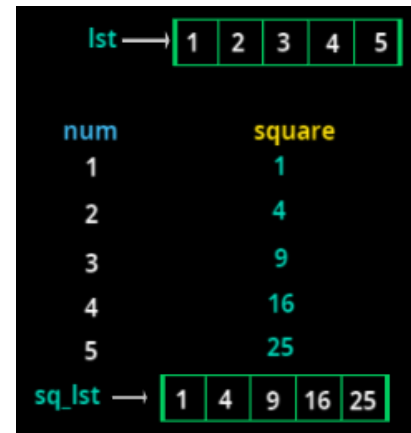
map()

map() function returns a **map** object of the results after applying the given function to each item of a given sequence (list, tuple etc.)

For example let's take a list with certain numbers and map it with its squared values.

Syntax:

map (function,sequence)

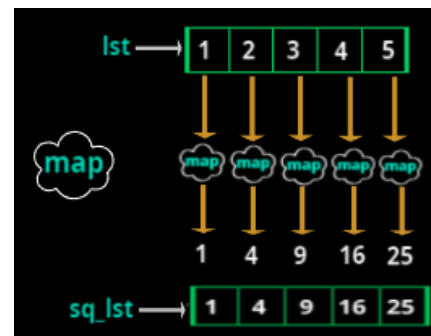


Let us try to code the above logic using **def** function first and then use **lambda** function

```
lst=[1,2,3,4,5]

def fun(x):
    return x**2

sq_lst=list(map(fun,lst))
print(sq_lst)
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
<map object at 0x0000021303701C08>

In [7]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[1, 4, 9, 16, 25]
```

In the above output we see that similar to filter, in mapping also map object is created and to see the result in list format we have to type cast it to list.

Let us see the same code with **lambda** function now

```
lst=[1,2,3,4,5]

'''def fun(x):
    return x**2

sq_lst=list(map(fun,lst))
print(sq_lst)'''
sq_lst=list(map(lambda x : x**2,lst))
print(sq_lst)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[1, 4, 9, 16, 25]
```



Certainly the code has again reduced with less complexity here as well.

Other examples

In python a function can not only be given as input to a function. If you choose to, you can design a function in such a way that about calling a function it would return a function as output to you.

Let us see how

```
def fun1(num):
    return lambda x : x*num

result=fun1(2)(5)  #(num)(x)
print(result)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
```

But what if we want to call it multiple times?? Let us see the solution

```
def fun1(num):
    return lambda x : x*num

'''result=fun1(2)(5)  #(num)(x)
print(result)'''

fun2=fun1(2) #first function call
print(fun2(5)) #call to lambda function
```



Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
```

Like this we can call lambda function multiple times.

Now let us see using **fun1()** how can we create a **mathematical table of any given number**

```
def fun1(num):
    return lambda x : x*num

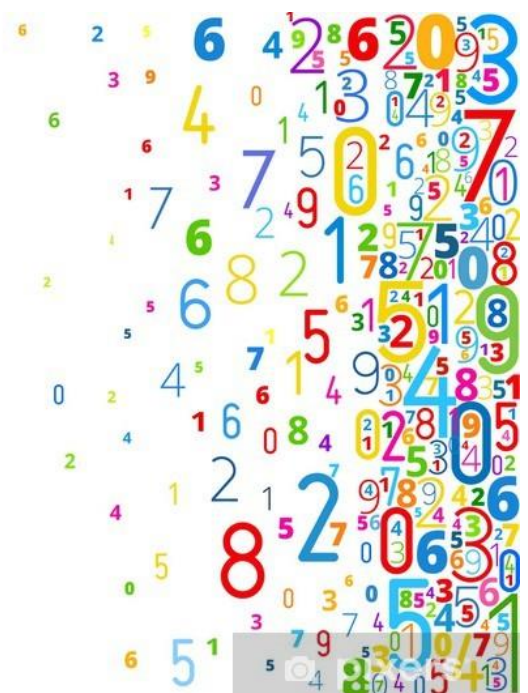
n= int(input("Enter a number\n"))
math_table=fun1(n)

for i in range(1,11):
    print(n,"X",i,"=",math_table(i))
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter a number
7
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70
```




```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

Enter a number

12

12 X 1 = 12

12 X 2 = 24

12 X 3 = 36

12 X 4 = 48

12 X 5 = 60

12 X 6 = 72

12 X 7 = 84

12 X 8 = 96

12 X 9 = 108

12 X 10 = 120

Similarly we can give any number and generate its mathematical table. Why not try yourself and verify

Multiplication Table

1	2	3	4	5	6
1 x 1 = 1	2 x 1 = 2	3 x 1 = 3	4 x 1 = 4	5 x 1 = 5	6 x 1 = 6
1 x 2 = 2	2 x 2 = 4	3 x 2 = 6	4 x 2 = 8	5 x 2 = 10	6 x 2 = 12
1 x 3 = 3	2 x 3 = 6	3 x 3 = 9	4 x 3 = 12	5 x 3 = 15	6 x 3 = 18
1 x 4 = 4	2 x 4 = 8	3 x 4 = 12	4 x 4 = 16	5 x 4 = 20	6 x 4 = 24
1 x 5 = 5	2 x 5 = 10	3 x 5 = 15	4 x 5 = 20	5 x 5 = 25	6 x 5 = 30
1 x 6 = 6	2 x 6 = 12	3 x 6 = 18	4 x 6 = 24	5 x 6 = 30	6 x 6 = 36
1 x 7 = 7	2 x 7 = 14	3 x 7 = 21	4 x 7 = 28	5 x 7 = 35	6 x 7 = 42
1 x 8 = 8	2 x 8 = 16	3 x 8 = 24	4 x 8 = 32	5 x 8 = 40	6 x 8 = 48
1 x 9 = 9	2 x 9 = 18	3 x 9 = 27	4 x 9 = 36	5 x 9 = 45	6 x 9 = 54
1 x 10 = 10	2 x 10 = 20	3 x 10 = 30	4 x 10 = 40	5 x 10 = 50	6 x 10 = 60
1 x 11 = 11	2 x 11 = 22	3 x 11 = 33	4 x 11 = 44	5 x 11 = 55	6 x 11 = 66
1 x 12 = 12	2 x 12 = 24	3 x 12 = 36	4 x 12 = 48	5 x 12 = 60	6 x 12 = 72
7	8	9	10	11	12
7 x 1 = 7	8 x 1 = 8	9 x 1 = 9	10 x 1 = 10	11 x 1 = 11	12 x 1 = 12
7 x 2 = 14	8 x 2 = 16	9 x 2 = 18	10 x 2 = 20	11 x 2 = 22	12 x 2 = 24
7 x 3 = 21	8 x 3 = 24	9 x 3 = 27	10 x 3 = 30	11 x 3 = 33	12 x 3 = 36
7 x 4 = 28	8 x 4 = 32	9 x 4 = 36	10 x 4 = 40	11 x 4 = 44	12 x 4 = 48
7 x 5 = 35	8 x 5 = 40	9 x 5 = 45	10 x 5 = 50	11 x 5 = 55	12 x 5 = 60
7 x 6 = 42	8 x 6 = 48	9 x 6 = 54	10 x 6 = 60	11 x 6 = 66	12 x 6 = 72
7 x 7 = 49	8 x 7 = 56	9 x 7 = 63	10 x 7 = 70	11 x 7 = 77	12 x 7 = 84
7 x 8 = 56	8 x 8 = 64	9 x 8 = 72	10 x 8 = 80	11 x 8 = 88	12 x 8 = 96
7 x 9 = 63	8 x 9 = 72	9 x 9 = 81	10 x 9 = 90	11 x 9 = 99	12 x 9 = 108
7 x 10 = 70	8 x 10 = 80	9 x 10 = 90	10 x 10 = 100	11 x 10 = 110	12 x 10 = 120
7 x 11 = 77	8 x 11 = 88	9 x 11 = 99	10 x 11 = 110	11 x 11 = 121	12 x 11 = 132
7 x 12 = 84	8 x 12 = 96	9 x 12 = 108	10 x 12 = 120	11 x 12 = 132	12 x 12 = 144