Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #92

Functions – filter and reduce

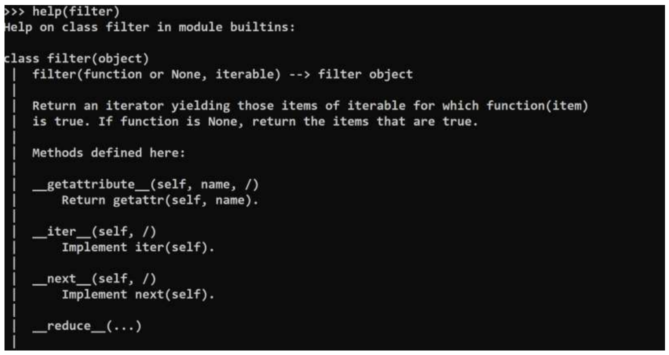
By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU
&
Dr.Manju More E
Associate Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M,(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

The function – filter()

Returns a filter object (which is an iterator) where the items are filtered through a function to test if the item is accepted or not. If there is a requirement to remove a few elements of the input iterable, then we make use of the filter function. This function filters the given iterable with the help of a function that tests each element is following some condition or not resulting in a Boolean value. The object created by filter function is a lazy object. Means**, unless the map object is iterated, you will not be able to access any of the elements from it**.

We can force an **iteration of the filter object** by passing the filter object as an argument for the **list() or set() or tuple().** The **for loop** of the client iterates through the filter object and displays the elements.



```
>>> help(filter)
Help on class filter in module builtins:

class filter(object)
 |  filter(function or None, iterable) --> filter object
 |
 |  Return an iterator yielding those items of iterable for which function(item)
 |  is true. If function is None, return the items that are true.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |
```

Note: You can pass one or more iterable to the filter() function.

Characteristics:

- Input : An iterable of some number of elements (say n)
- Output: A lazy iterator of 0 to n elements(between 0 and n)
- Elements in the output: Apply the callback on each element of the iterable – if the function returns True, select the element from the iterable and otherwise ignore the element.

Think about this!

- **Can you pass more than one iterable to filter function?**
- **Can you use lambda as a callback in the first argument of filter() ?**

Let us see a few example codes.

Example_code_1: Given a list, create a new list by removing negative numbers.

Input: Combination of positive and negative numbers

Expected output: Only Positive numbers

```
numbers = [ 1, -2, 3, -4, 5, -6 ]
result = list(filter(lambda x: x >= 0, numbers))
print(result)
```

```
[1, 3, 5]
```

Example_code_2: Create a list of words from the given set of words which has the letter ‘r’ within it.

```
a = {'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita'}
print(list(filter(lambda s : 'r' in s , a)))
```

```
['dasharatha', 'rama', 'balarama', 'krishna']
```

Example_code_3: Pickup all words from list whose length exceeds 4 and create a new list

```
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita']
print(list(filter(lambda s : len(s) > 4 , a)))
```

```
['krishna', 'balarama', 'lakshmana', 'dasharatha']
```

Think about this! -> What happens if we replace filter word with map in each of the above programs? What will be stored in the newly created map object?

There are many cases where we may want to do some mapping and filtering both to do the requirement**. It is always a good habit to filter first and then map.** Otherwise map will have to do processing of some elements which eventually be filtered out.

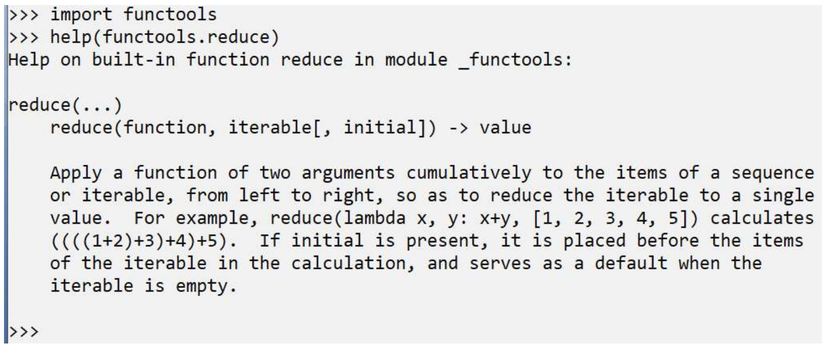
Example_code_4: Create a list by converting all strings in the list to uppercase and find all strings whose length exceeds 4.

```
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
#print(list(filter(lambda s : len(s) > 4 , map(str.upper , a))))  #Terrible code
print(list(map(str.upper, filter(lambda s : len(s) > 4, a))))  #good code
```

```
['KRISHNA', 'BALARAMA', 'LAKSHMANA', 'DASHARATHA']
```

The function - reduce()

Available in **“functools” module**. This function applies a given function to the elements of an iterable, reducing them to a single value.



```
>>> import functools
>>> help(functools.reduce)
Help on built-in function reduce in module _functools:

reduce(...)
    reduce(function, iterable[, initial]) -> value

    Apply a function of two arguments cumulatively to the items of a sequence
    or iterable, from left to right, so as to reduce the iterable to a single
    value.  For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
    ((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
    of the iterable in the calculation, and serves as a default when the
    iterable is empty.

>>>
```

Syntax: **functools.reduce(function, iterable[, initializer])**

- The function argument is the name of the function that takes two arguments and returns a single value. The first argument is the accumulated value, and the second argument is the current value from the iterable.
- The iterable argument is the iterable to be reduced.
- The optional initializer argument is used to provide an initial value. If no initializer is specified, the first element of the iterable is used as the initial value.

Characteristics:

- Input: An iterable of some number of elements (say n)
- Output: A single element
- Processing: Requires a callback which takes two elements. The callback function is called repeatedly on the iterable and reduce the output to a single element.

Working :

First two elements of sequence are picked and the result is obtained. Then the same function to the previously attained result and the number just succeeding the second element. Again the result is obtained and this process continues till no more elements are left in the container. The final returned result is returned.

Let us consider an example code to find the product of all numbers in a list.

```
# input: iterable  , output: single element. Usage of reduce is appropriate.
#Writing a user defined function
def get_product(x,y):
	#print(x,y)  # to understand better, this line is added. Uncomment and check the output
	return x*y
li = [2,4,1,6,7]
import functools
p = functools.reduce(get_product,li)
print(p)
```

```
2 4
8 1
8 6
48 7
336
```

There will be n – 1 calls where n is the number of elements in the input iterable. If we provide the initial value,i.e., the third argument for reduce function, the first argument of callback receives the initial value.

```
def get_product(x,y):
	#print(x,y)  # to understand better, this line is added. Uncomment and check the output
	return x*y
li = [2,4,1,6,7]
import functools
p = functools.reduce(get_product,li,1)
#Change 1 to 0 and observe the output
print(p)
```

```
1 2
2 4
8 1
8 6
48 7
336
```

Think about the number of times the callback function is called now in the above code?

Now, we shall use lambda rather than using user defined function?

```
li = [2,4,1,6,7]
import functools
p = functools.reduce(lambda x,y: x*y,li)
print(p)
```

```
336
```

Example_code_5: li = ["sindhu", "r", "pai", "sagara"]
#Expected output: srps

```
import functools
li = ["sindhu", "r", "pai", "sagara"]
def compress(x,y):
  return x + y[0]
print(functools.reduce(compress, li, ""))
#print(functools.reduce(lambda x,y : x + y[0], li, ""))  #using lambda
```

```
srps
```

Few points to think:

- **If we do not pass the empty string as third argument to reduce, what happens in the above code?**
- **Can we pass more than 2 parameters to callable of reduce function?**

-END-