map vs filter in Python 3

K. S. Ooi
Foundation in Science
Faculty of Health and Life Sciences
INTI International University
Persiaran Perdana BBN, Putra Nilai,
71800 Nilai, Negeri Sembilan, Malaysia
E-mail: kuansan.ooi@newinti.edu.my
dr.k.s.ooi@gmail.com

Abstract
Higher-order functions are essential tools for functional programmers, but majority of Python programmers are imperative. When the higher-order functions are used in imperative development, it usually has negative impact on readability of the code. With a small investment on understanding these higher-order functions, we may be able to alleviate some miscommunication between developers with different backgrounds in Python. In this article, two higher-order functions are examined: map and filter. Few examples are given in this article to illustrate their differences.

Keywords: Python 3, higher-order functions, map, filter
Date: Dec 25, 2020

1. Introduction

Functional programmers adore higher-order functions. These are functions that can accept functions as arguments and return functions as results. Hughes [1] highlights that higher-order functions (and lazy evaluation) can contribute to modularity of software construction. Programmers who have been grounded firmly on imperative paradigm would not agree easily with Hughes; one such individual is Guido van Rossum [2], the creator of Python. Python has a number of higher-order functions. In this article, I will discuss two of them, map() and filter(), which are built-in functions of Python 3.

Since map() is a built-in function, you will find documentation of it from Python 3 documentation site [3]:

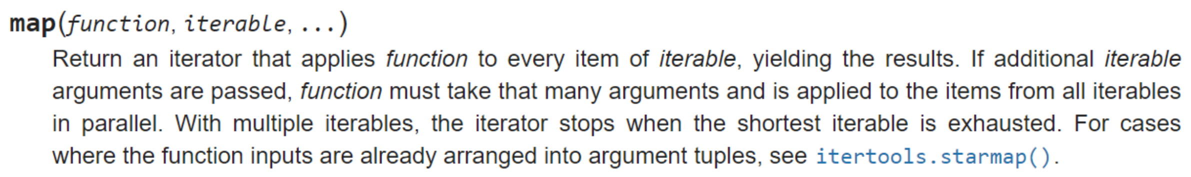
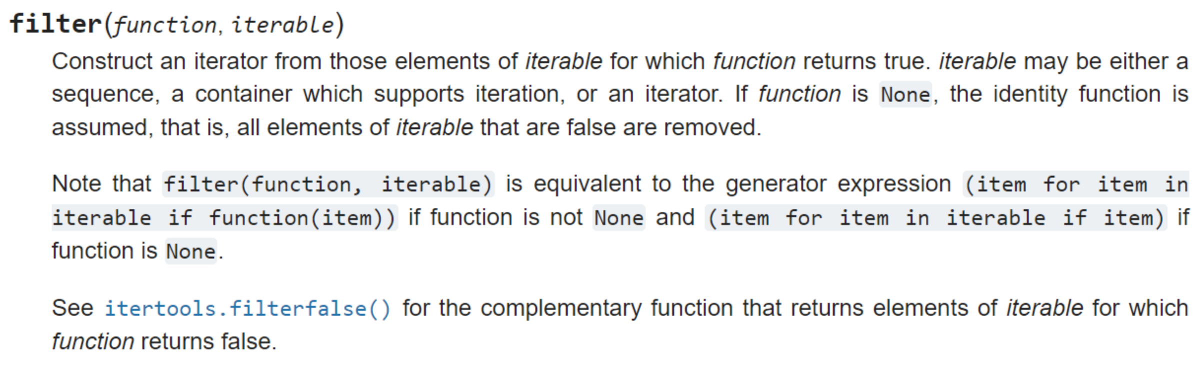

map(*function*, *iterable*, ...)
Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

Figure 1: The map() function [3].

Some of the terms used in the documentation require additional explanation. First, the *iterator* that the function returns. This is a *map* object, an iterator object that represents a stream of data. This map iterator is neither a sequence nor a collection iterable. However, you can iterate the map object using a for statement, just like you do with sequence inerables (list, string, tuple, range, or bytes). In this article, I assume a working knowledge of lambda, the anonymous function.

The filter() is another built-in, higher-order function. From the Python 3 documentation site [3]:



filter(*function*, *iterable*)

Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if function is not `None` and `(item for item in iterable if item)` if function is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* returns false.

Figure 2: The filter() function [3].

The iterator object returned is a filter object. All other terms should be clear to you. Again, throughout this article, the function will be anonymous function, the lambda.

2. First Problem

I receive five legit Python statements from John, Keith, Lamp, and Carl. I retrieve their statements from database as str objects and I made a list. I want to compute their statements and output the resulting values. What to use: map or filter?

The solution is given in Program 1.

Program 1

```
import math

from_db = ["3*5**2 # John" , "9 - 8*5 # Keith", "math.exp(5*3) # Lamp", "3 + 26 # Carl"]

a = map(eval, from_db)

print(list(a))
```

In this problem, I need to *apply eval() function to each element* of from_db. Obviously, map is the only choice here. You need to transform each individual statement of from_db to a numerical value; filter() would not do that.

3. Second Problem

I have three tuples, each contains a sequence of integers. The resulting value would be the multiplication of the first element of first tuple with the resulting exponential of the first element of the second tuple and the first element of the third tuple. Continue the computation until the tuples are exhausted.

The solution is in Program 2.

Program 2

```
from math import pow

f_tp = (3,2,7,-3)
s_tp = (4,6,3,5)
t_tp = (2,2,3,3,3)

a = map(lambda x, y, z: x*pow(x,z), f_tp, s_tp, t_tp)

print(list(a))
```

You have only one choice here: map. The problem requires you to process three iterables. Between map and filter, only map is capable of doing that. Furthermore, you need to process each element of the tuples, until you have exhausted the shortest one.

4. Third Problem

I have a list of people and their ages. In this computation, I want to find out how many people are qualified to be the first vaccinated. You must be younger than 10 or older than 60 to be qualified. So, I extract the data and put the ages in a list. How do I find out the number of people is to be vaccinated first?

The solution is given in Program 3. Of course, I use toy data here.

Program 3

```
# Counting visually, 7 is qualified
age = [45, 7, 80, 23, 56, 60, 10, 78, 101, 88, 92, 38, 3]

a = filter(lambda x: x < 10 or x > 60, age)

print(sum(1 for _ in a))
```

First, you need to construct a iterator that may have different length from the iterable you put in as argument. Only filter() is able to do that between the two higher-order function. So, there you have it.

5. Fourth Problem

I have a list of numbers. These numbers are money my company earn from individuals doing business with us. Out of these numbers, my company has to pay tax if an individual spent more than $500. The tax varies: more than $500 but less than $1000, 5% tax; otherwise, 7%. Count how much tax my company needs to pay for this series of money earn.

The solution is in Program 4. This is only one of the solutions.

Program 4

```
money_earn = [500, 600.6, 300.3, 460., 790.8, 900.2, 1200, 50, 1300., 800.8, 12, 1000]

# taxable
a = filter(lambda x: x > 500, money_earn)

# calculate tax for each taxable earning
b = map(lambda x: x*0.05 if x <= 1000 else x*0.07, a)

# print out the tax amount
print(sum(b))
```

In this solution, filter() creates a taxable iterator, and map() creates a iterator that contains the tax for each taxable money earn. In the end, we sum the taxes need to pay.

However, map alone can do the job. This is shown in Program 5.

Program 5

```
def the_tax(money):
   if money > 1000:
      return 0.07*money
   elif money > 500:
      return 0.05*money
   else:
      return 0.0

money_earn = [500, 600.6, 300.3, 460., 790.8, 900.2, 1200, 50, 1300., 800.8, 12, 1000]

a = map(the_tax, money_earn)

print(sum(a))
```

In Program 5, I do not *select*. I perform tax calculation for each element of money earn. So, the iterator *a* has the same dimension as the iterable *money_earn*. Here, I define the function *the_tax*. How able lambda function, you may ask? I can lambda, but you may not like it! See the following program.

Program 6
```
money_earn = [500, 600.6, 300.3, 460., 790.8, 900.2, 1200, 50, 1300., 800.8, 12, 1000]

a = map(lambda x: 0.07*x if x > 1000 else 0.05*x if x > 500 else 0, money_earn)

print(sum(a))
```

However, I am able to make the program a bit readable by providing a pair parenthesis.

Program 7
```
money_earn = [500, 600.6, 300.3, 460., 790.8, 900.2, 1200, 50, 1300., 800.8, 12, 1000]

a = map(lambda x: (0.07*x if x > 1000 else 0.05*x) if x > 500 else 0, money_earn)

print(sum(a))
```

6. Concluding Remarks

The difference between map() and filter() is rather obvious. I can summarize it in a 4x4 table.

map	**filter**
Perform computation on each element of the iterable	Select elements from the iterable that satisfy the conditions in the function
Able to work on more than one iterable	Only can select from one iterable

Table 1: map versus filter in 4x4.

Solving problems using map and filter can be summarized below:

- You must have an iterable to begin with.
- You do not want to use a for statement or a while statement, for reason that these control statements are not cool.
- You want to pick and select the elements out from the iterable or you want to transform all the elements of the iterable.

If you want to program, and do not want to look cool, for and while statements are the way forward. You should go for it.

References

1. John Hughes, *Why Functional Programming Matters*, from *Research Topics in Functional Programming*, edited by D. Turner, Addison-Wesley (1990)

2. Guido van Rossum, *The fate of reduce() in Python 3000*, In: *All Things Pythonic*. Available from: http://www.artima.com/weblogs/viewpost.jsp?thread=98196 (2005) (accessed Dec 25, 2020)

3. The Python Standard Library, *Built-in Functions*, at https://docs.python.org/3/library/functions.html (accessed Dec 26, 2020)