reduce() 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

Abstract

Python, the most popular language at this moment, is the language of choice to write readable code. As coding instructors, it is our virtue to teach students to write code that is easy to read and understand. However, Python was older than Java. There are functional programming components that survive, and many students find them too cool to ignore. I address one such higher-order function, reduce(), in this article. Over-reduce() is the first issue I discuss. If we understand the problem well and do not shoehorn our problem into reduce(), there are alternatives readily available. In this article, I gives few examples that we should not reduce().

Keywords: Higher-order function, reduce(), lambda, Python 3
Date: Dec 25, 2020

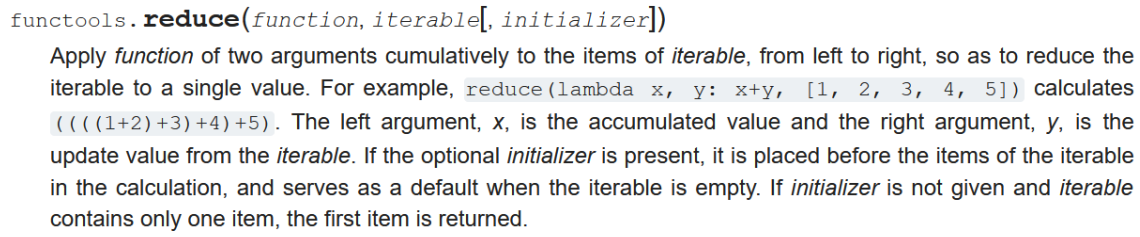
1. Introduction

Teaching programming languages to college students is apparently dictated by popular demand, which is tied to market demand. When I first started teaching programming at a college in 1997, Java was beginning to supplant C/C++ as the preferred programming language to teach. In the last ten years, we have a new favorite: Python. Popularity of programming languages can be measured by various means [1], and all the published indices put Python as the top programming language to learn, in the next few years, at least. The revival of machine learning and artificial intelligence, and the surging demands of big data experts, propel Python to be the language of choice for college students [2], who expected to be paid handsomely [3]; learning Python is a well-worth investment return.

For language creators and teachers, other concerns, besides popularity, factor in. With Python code and snippets available over all places, particularly the internet, as a programming language instructor, I am bombarded with all sorts of questions about coding choices, and code the students have difficulty to follow. According to van Rossum [4], Python was created, first and foremost, to write readable code. This is exactly the intent to teach Python to our students. But there are functional programming remnants that can be found in today's Python. Obviously, van Rossum *was* not a fan of functional programming back in 2005; he planned to get rid of lambda, map(), filter(), and reduce() from Python 3000, the project name of Python 3 at that time. The higher-order function reduce() was singled out to be cut, and van Rossum reasoned that they are better replacements available in Py3k [4]. However, it turns out that as of today, lambda is part of Python, map() and filter() are built-in functions, and reduce() is available in *functools* [5], the standard library that houses selected functional programming

modules. The functional programming remnants can make our teaching job much more complicated.

Before I begin, the definition of reduce() function in *functools* documentation [5] has been screen-captured in Figure 1 for your convenience.


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

Apply *function* of two arguments cumulatively to the items of *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)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initializer* is not given and *iterable* contains only one item, the first item is returned.

Figure 1: The higher-order function reduce() from *functools* documentation [5].

2. What is reduce()? The First Example

The higher order function reduce() is a staple of functional programmers. Like it or otherwise, it is also part of Python, despite repeated call for its demise. Iterables are objects in Python that you can iterate and manipulate using a loop; examples are lists, tuples, dictionaries, etc. The two essential arguments of reduce() are *function* and *iterable*. In essence, reduce applies the *function* to the first two elements of the *iterable*, and then applies the resulting value to the next element, and so on, until the last element. The resulting value is returned in the end. This may sound confusing on first read, but if you look at the first program, you should have much clearer picture.

In the following program, we first import reduce() from *functools* library. Next, we define a function call *add*, which, as the name implies, takes two arguments and returns the sum of them. Then, we have a sequence of Fibonacci numbers in a list called *fib_seq*. The *add* function and *fib_seq* are the arguments of reduce(). The program will print 609 on the screen.

Program 1

```
from functools import reduce

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

fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(add,fib_seq))
```

Let us use the notation => to denote the resulting value of reduce(). We can trace what reduce() does in Program 1 as follows:

Step 1: add(0,1) => 1 # Add the first two elements of fib_seq

Step 2: add(1,1) => 2 # Use the result of step 1 as first argument, add it to the 3rd element

Step 3: add(2,2) => 4 # Use the result of step 2 as first argument, add it to the 4th element

Step 4: add(4,3) => 7 # Use the result of step 3 as first argument, add it to the 5th element

…

Step 12: add(232,144) =>376 # Use step 11's result as first argument, add it to the 13th element

Step 13: add(376,233) => **609** # and .. we are done

If you are able to tell me that the output of Program 2 is 720, you got it.

Program 2

```
from functools import reduce

def multiply(x,y):
   return x * y

# list(range(1,7)) = [1, 2, 3, 4, 5, 6]
print(reduce(multiply,range(1,7)))
```

3. Are We Over-reduce()?

We can use lambda function instead of defining an *add* function. This is shown in Program 1b.

Program 1b

```
from functools import reduce

fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(lambda x,y: x+y, fib_seq))
```

Or you can use add function from the *operator* library.

Program 1c

```
from functools import reduce
import operator

fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(operator.add, fib_seq))
```

I have done three version of Program 1, but it cannot hide the fact that we are over-reduce() our problem. Program 1 sums up fib_seq. You may as well write Program 1 as follows.

Program 1d*

```
fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(sum(fib_seq))
```

Program 2 is also over-reduce(). Using factorial function from math library we can do the job. The solution is given in Program 2b.

Program 2b*

```
import math

print(math.factorial(6))
```

Let us expand the problem of Program 1. Let say we are required to sum only the odd number in the list. We may modify Program 1b as follows.

Program 3

```
from functools import reduce

fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(lambda x,y: x+y if y % 2 == 1 else x + 0, fib_seq))
```

Program 3 work, because the first element of fib_seq is 0. Let say we have a list [2,3,5,8,13,21,34,55,89,144,233] Program 3 will fail because check of oddity is not performed for the first element of the list. One solution to overcome this is by inserting 0 as the first element to the list. Both wrong and correct summation of odd numbers in the list are shown in the following program.

Program 4

```
from functools import reduce

a_seq = [2,3,5,8,13,21,34,55,89,144,233]

# Give wrong answer => 421
print(reduce(lambda x,y: x+y if y % 2 == 1 else x + 0, a_seq))

# Give the correct answer => 419
a_seq.insert(0,0)
print(reduce(lambda x,y: x+y if y % 2 == 1 else x + 0, a_seq))
```

Over-reduce() is a specific form of over-thinking. Over the internet, you will find people keep suggesting list comprehension to solve this and that problem. Let us apply list comprehension to Program 4. But we need help of the sum to get the result.

Program 4b

```
a_seq = [2,3,5,8,13,21,34,55,89,144,233]

print(sum([i for i in a_seq if i % 2 == 1]))
```

However, if you are still insisting on using reduce()/lambda, for whatever reason you may have, we need the help of filter() function. This is shown in the following program.

Program 4c

```
from functools import reduce

a_seq = [2,3,5,8,13,21,34,55,89,144,233]

print(reduce(lambda x,y: x+y, filter(lambda x: x % 2 == 1, a_seq)))
```

Do not brush off the good old for-loop. For newbies who have just completed the first programming course using Python, this should be the most comfortable construct he or she can fall back on. The code is in multiple lines, which we can easily debug and trace.

Program 4d

```
a_seq = [2,3,5,8,13,21,34,55,89,144,233]

# the_sum will be the sum of all odd number from a_seq
the_sum = 0
for i in a_seq:
    if i % 2 == 1:
        the_sum += i

print(the_sum)
```

Program 4d, albeit not that cool, can be modified easily and should be the version you should stashed away, should you anticipate, for example, new requirements to sum the list to arise in the near future. However, recursive version is over the top, as shown in Program 4e.

Program 4e

```
a_seq = [2,3,5,8,13,21,34,55,89,144,233]

def recur_sum(a):
   if not a:
      return 0
   else:
      if a[0] % 2 == 1:
         return a[0] + recur_sum(a[1:])
      else:
         return 0 + recur_sum(a[1:])

print(recur_sum(a_seq))
```

4. Should We Not-reduce()?

When program Python, you should put your readiness to reduce() aside. However, if your team manager requires you to do all things functional, you have no choice. Even in that scenario, look for a simpler solution.

In program 5, I give you two versions that produce the same result.

Program 5

```
fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(lambda x,y: x and y, [True if i > 0 else False for i in fib_seq]))

print(reduce(lambda x,y: x and y, map(lambda x: True if x > 0 else False, fib_seq)))
```

A better solution is Program 5a.

Program 5a*

```
fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(all(x > 0 for x in fib_seq))
```

Now you see my point. How about Program 6? Again, two equivalent versions are given.

Program 6

```
fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(reduce(lambda x,y: x or y, [True if i > 0 else False for i in fib_seq]))
print(reduce(lambda x,y: x or y, map(lambda x: True if x > 0 else False, fib_seq)))
```

It has to be Program 6a.

Program 6a*

```
fib_seq = [0,1,1,2,3,5,8,13,21,34,55,89,144,233]

print(any(x > 0 for x in fib_seq))
```

5. Concluding Remarks

Python is a multi-paradigm programming language. It is also an inclusive language. Since its inception, the ex-Lisp and ex-Scheme folks were active participants in the Python community. Despite Guido's effort to get rid of reduce() [4], reduce() survives to Python 3. But functional programming paradigm has its Renaissance in recent years. In my opinion, at this moment Guido might have changed his mind about the functional remnants in Python 3. Having said that, we should not see problems involving iterables as reduce() problems. If we do not shoehorn our iterable problems into reduce(), we might come out having better solutions. As a programming instructor, I would want my students to look for solutions which they can read and understand. This article provides some resources you can use in your class.

References

1. Wikipedia, *Measuring Programming Language Popularity*, at https://en.wikipedia.org/wiki/Measuring_programming_language_popularity (accessed Dec 25, 2020)

2. Sruthi Veeraraghavan, *Best Programming Languages to Learn in 2021*, at https://www.simplilearn.com/best-programming-languages-start-learning-today-article (accessed Dec 25, 2020)

3. glassdoor, *Python Developer Salaries*, at https://www.glassdoor.com/Salaries/python-developer-salary-SRCH_KO0,16.htm (accessed Dec 25, 2020)

4. 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)

5. Python Standard Library, *functools - Higher-order functions and operations on callable objects*, at https://docs.python.org/3/library/functools.html (accessed Dec 25, 2020)