Python 3 Lambda

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

Python coders are traditionally imperative. They may program in the paradigms of structured or object-oriented programming, but seldom in functional program. The resurgence of functional programming in the past decade or so has sent Python coders into tailspin. Lambda functions appear in countless of code, not as pure functional but as hybrid to the traditional imperative. Various resources about lambda function add even more confusion. This article address lambda in the context of Python, with minimum jargon but with a number of illustrative examples.

Keywords: lambda, Python 3, higher-order functions, map, filter
Date: Dec 26, 2020

1. Anonymous

I recently made a foray into lambda function in a book chapter [1]. Now it is time to expand the section of that chapter about lambda function, address some misconceptions on it, and reduce jargons used addressing it. Let us begin by saying lambda function is *anonymous* function, function without name. Let us also begin by a problem that convert the mass of kg into pounds. The two units have fixed conversion factors:

1 kg	2.20462 lb
1 lb	0.453592 kg

If we have *x* kg, we can covert it to lb using the factor label method (dimensional analysis):

$$x\ \text{kg} \times \frac{2.20462\ \text{lb}}{1\ \text{kg}}$$

$$x\ \text{kg} \times \frac{1\ \text{lb}}{0.453592\ \text{kg}}$$

If you know a little math, you would prefer multiplication to division, and hence we choose the first conversion factor 1 kg = 2.20462 lb. The first program is created to convert a series of masses in kg into pounds. We define a function called *kg_to_lb*, and it is done. The intention of the program is clear.

Program 1

```
def kg_to_lb(kg):
    return 2.20462*kg

# Weights of students in kg
mass_kg = [45, 20, 50, 65, 96, 120]

print(list(map(kg_to_lb,mass_kg)))
```

If we use lambda instead of kg_to_lb, it is still ok provided you comment a little, because now the intention of the code may not be clear to many coders who do not have engineering or science degree or not familiar with that particular conversion factor. There are countless conversion factors out there in the open, readily available by googling; but if you keep the readers of the code in the dark, very likely they know where to get info, but not knowing what to look for. The comment in Program 2 is not perfect; imagine you take away those comments.

Program 2

```
# Weights of students in kg
mass_kg = [45, 20, 50, 65, 96, 120]

# 1 kg is 2.20462 lb: print mass in lb
print(list(map(lambda x: 2.20462*x, mass_kg)))
```

You can skip the comment by assigning the lambda function to an expressive variable. This is shown in the following program.

Program 3

```
# Weights of students in kg
mass_kg = [45, 20, 50, 65, 96, 120]

kg_to_lb = lambda x: 2.20462*x

print(list(map(kg_to_lb, mass_kg)))
```

If you are okay with it, I have nothing to say. Giving a name to a lambda function, in my opinion, is going against the anonymity of lambda. I would say Program 3 is okay, the intention is clear, the code readers can google up and verify the code; but yet I would not do it.

Let say we are tasked to convert a series of numbers, money in Great Britain Pound, into U. S. dollars. Using lambda, we have:

Program 4

```
# money in GBP
money = [57.9, 67.9, 72.6, 5.4]

print(list(map(lambda x: 1.36*x, money)))
```

Without comment, I have the slightest idea what the code tries to accomplish. Let say I figure it out after a few days coming out from my procrastination cocoon, the exchange rate might have changed!

2. One-Time Usage

Lambda function, in anonymous guise, is widely considered built for function that you compute only once (or even twice) in a piece of code. You may call this piece code repeatedly, elsewhere, as long as within that piece of code, the lambda function is used once or twice. To be more relaxed in software development, once bitten twice shy does not apply here. Let us be less fanatical about it. Yet, many Python coders have forgotten that they are not doing functional programming, but a little functional with lots of imperative. Caveat alert!

In the following program, we compute the discriminant of quadratic equation, and print out the number of roots the equation would have. Let us assume that throughout that piece of code, with many other lines of code not shown, discriminant will be computed once

Program 5

```
from math import sqrt

def number_of_roots(discriminant):
   if discriminant > 0:
      print("You have TWO distinct roots.\n")
   elif discriminant < 0:
      print("You have TWO complex roots.\n")
   else:
      print("You have ONE real roots.\n")

a = 4
b = 3
c = 2

discriminant = lambda a, b, c: b**2 - 4*a*c

number_of_roots(discriminant(a,b,c))
```

I am still disagreeing with you. Lambda function is specifically built for functional programming, working nicely with higher-order functions. With the mixed paradigm

programming shown in Program 5, it is truly unnecessary. Replace the last two lines of code with a single line, as shown below:

Program 6

```
# ..... some code here
# ..... some code here

number_of_roots(b**2 - 4*a*c)
```

3. Nearness

We promote less fanatical software development. So, if one of use has the habit of using lambda function, we tolerate to certain extent. You want to name a lambda function, fine to my, as long as it is near to that code that uses it. Climatology and meteorology applications use temperature conversions often, reporting back and forth temperatures in Celsius and Fahrenheit. In program 7 I show snippet of such conversions.

Program 7

```
# f is short for Fahrenheit
# c is short for Celsius

# You know what it means.
c_to_f = lambda c: 9.0*c/5.0 + 32.0

# ..... few lines of code follows ...

# You know what it means
f_to_c = lambda f: 5*(f - 32.0)/9.0

# ..... lines of code here

print(list(map(c_to_f, [-35, -25, -7, 0, 10., 120.])))

# ..... lines of code here

print(list(map(f_to_c, [-30.0, -12.5, 17.7, 33.0, 110., 220.])))
```

Climatologists and meteorologists are smart people. They know how to convert temperatures in these two scales. However, I would say they usually do not write the software they are using; the software developers usually do, and they are usually not proficient in temperature conversion. In many circumstances, they would not know c_to_f and f_to_c ready do. However, if these two functions are nearby, they can perform a quick check, but not if they are far away. For me, I am willing to relax anonymity and frequency of use for lambda functions provided they are near to the piece of code that uses them. Coding is not for extremists, but we still have threshold.

4. Higher-Order Functions

Higher-order functions are staple of functional programmers; in their mind, these functions contribute to modularity of software development [2]; Imperative-trained programmers may loathe these functions a little [3], difficult to say how much. Many times, higher-order functions reduce readability in hybrid development. You have better leave it alone.

If I have a list of marks from a test, I want to find out the number of students who obtain a B grade, whose marks between 70 and 80, I wound say it is fine to use higher-order function and lambda, with some comment.

Program 8

```
marks_test1 = [54.5, 90.0, 67.8, 75.0, 68.9, 79.0, 88.4, 79.9, 77.6, 84.5]

# find out the numbers of students who get B in test 1
print(len(list(filter(lambda x: x >= 70.0 and x < 80.0, marks_test1))))
```

The function filter() is higher-order and built-in, and hence it can take lambda function as its first argument. You cannot put lambda as argument in any function, even though you wish you could, as shown in the following program.

Program 9

```
marks_test1 = [54.5, 90.0, 67.8, 75.0, 68.9, 79.0, 88.4, 79.9, 77.6, 84.5]

# Wrong! Read the documentation
print(marks_test1.count(lambda x: x >= 70.0 and x < 80.0))
```

It is not wrong to use a for statement. For most of us who are trained imperative, it is godsend, and not pompous.

Program 10

```
marks_test1 = [54.5, 90.0, 67.8, 75.0, 68.9, 79.0, 88.4, 79.9, 77.6, 84.5]

num_of_B = 0
for marks in marks_test1:
   if marks >= 70.0 and marks < 80.0:
      num_of_B += 1

print(num_of_B)
```

You really need to get used to using higher-order function to use lambda. Sometimes, it can be too much to bear, as shown in the following Program.

Program 11

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

a = map(lambda x: 0.08*x if x > 1000 else 0.03*x if x >= 600 else 0, money)

print(sum(a))
```

In Program 11, It simply computes the sum of tax collected from a series of money people earned. People who earned less than 600 are exempted from tax. For people who are taxable, they have to pay 3% tax if their earning is not more than 1000; if more than 1000, 8% tax. Isn't clearer if I use a for statement?

Program 12

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

tax_collected = 0.0

for m in money:
  if m > 1000:
    tax_collected += 0.08*m
  elif m >= 600:
    tax_collected += 0.03*m

print(tax_collected)
```

5. Recursion

If I want to obtain the greatest common divisor (*gcd*) using the Euclid algorithm, even in recursive version, many of us is able to read it:

Program 13

```
def r_gcd(a, b):
    if b == 0:
            return a
    return r_gcd(b, a % b)

print(r_gcd(20,5))
```

The named lambda version is still readable:

Program 14

```
l_gcd = lambda a, b: a if b == 0 else l_gcd(b, a % b)

print(l_gcd(20,23))
```

But if the code spans multiple line, it will be quite difficult to read [1]:

Program 15

```
prime = lambda n, d = 3: (True if d*d > n else prime(n, d = d + 2))
if n % 2 and n > 3 and n % d else n == 2 or n == 3
```

Program 15 is the simplest primality test of a number. Imagine so many problems out there which complexity easily exceeds this simple algorithm many times over.

6. Closure

Putting a lambda in a function is not desirable in Python development. Consider the following example, which computes the collectable tax from people who earn more than 500 with the tax 5%:

Program 16

```
def a_func(money):
  return lambda rate, earning: rate*money if money > earning else 0.0

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

tax_coll = 0
for m in money:
  tax_coll += a_func(m)(0.05, 500)

print(tax_coll)
```

Even though I can go functional, I still find closure with lambda disturbing in Python. Up to this point, you should be able to write a much clearer imperative program equivalent to Program 16. It simply makes very little sense to do this in a Python's hybrid development. However, if you project is functional using languages like Haskell, this kind of way to structure your functions is inevitable. In Python is a nay.

7. Conclusion

The first requirement to become a better coder is to achieve perfect or near perfect understanding of the problem in hand. Sometimes it can be impossible to achieve full understanding, then go for the less perfect one is still better than understanding with tremendous ambiguity. If you still insist on using lambda, at least have the courtesy to comment the code to render it readable to others. Because majority of Python coders is imperative, and developing software in hybrid mode, the first that should come to mind is no lambda. If insistent lambda usage is high in a development environment, we can tolerate it provided it does not sacrifice correctness and readability. Lambda functions are used with higher-order functions. Some functions may look and sound like higher-order function and you wish they were, but they are not. Lambda function should not be recursed; you will see nightmarish code, even for the simplest problems. Lambda for closure can only complicate Python development. If one really wants to go functional, there are many other programming languages one can opt for: Haskell should be on top of one's list.

References

1. K. S. Ooi, *The Perils of Idiomatic Python*, Chapter 1, Current STEM. Volume 2, Edited by Maurice H. T. Ling, Nova Science Publishers (2019)

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

3. 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)