Lambda Functions with Practical Examples in Python

How, when to use, and when not to use Lambda functions

Susan Maina Jun 17 · 6 min read ★

Photo by Pixabay from Pexels

When I first came across lambda functions in python, I was very much intimidated and thought they were for advanced Pythonistas. Beginner python tutorials applaud the language for its readable syntax, but lambdas sure didn't seem user-friendly.

However, once I understood the general syntax and examined some simple use cases, using them was less scary.

Syntax

Simply put, a lambda function is just like any normal python function, except that it has no name when defining it, and it is contained in one line of code.

```
lambda argument(s): expression
```

A lambda function evaluates an expression for a given argument. You give the function a value (argument) and then provide the operation (expression). The keyword `lambda` must come first. A full colon (:) separates the argument and the expression.

In the example code below, x is the argument and x+x is the expression.

```
#Normal python function
def a_name(x):
    return x+x

#Lambda function
lambda x: x+x
```

Before we get into practical applications, let's mention some technicalities on what the python community thinks is good and bad with lambda functions.

Pros

- Good for simple logical operations that are easy to understand. This makes the code more readable too.
- Good when you want a function that you will use just one time.

Cons

- Bad for operations that would span more than one line in a normal `def` function (For example nested conditional operations). If you need a minute or two to understand the code, use a named function instead.
- Bad because you can't write a doc-string to explain all the inputs, operations, and outputs as you would in a normal `def` function.

At the end of this article, we'll look at commonly used code examples where Lambda functions are discouraged even though they seem legitimate.

But first, let's look at situations when to use lambda functions. Note that we use lambda functions a lot with python classes that take in a function as an argument, for example, map() and filter(). These are also called Higher-order functions.

1. Scalar values

This is when you execute a lambda function on a single value.

```
(lambda x: x*2)(12)

###Results
24
```

In the code above, the function was created and then immediately executed. This is an example of an immediately invoked function expression or IIFE.

2. Lists

Filter(). This is a Python inbuilt library that returns only those values that fit certain criteria. The syntax is `filter(function, iterable)`. The iterable can be any sequence such as a list, set, or series object (more below).

The example below filters a list for `even` numbers. Note that the filter function returns a 'Filter object' and you need to encapsulate it with a list to return the values.

```
list_1 = [1,2,3,4,5,6,7,8,9]
filter(lambda x: x%2==0, list_1)
```

```
list(filter(lambda x: x%2==0, list_1))

###Results
[2, 4, 6, 8]
```

Map(). This is another inbuilt python library with the syntax `map(function, iterable).`

This returns a modified list where every value in the original list has been changed based on a function. The example below cubes every number in the list.

```
list_1 = [1,2,3,4,5,6,7,8,9]
cubed = map(lambda x: pow(x,3), list_1)
list(cubed)

###Results
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

3. Series object

A Series object is a column in a data frame, or put another way, a sequence of values with corresponding indices. Lambda functions can be used to manipulate values inside a Pandas dataframe.

Let's create a dummy dataframe about members of a family.

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Luke','Gina','Sam','Emma'],
    'Status': ['Father', 'Mother', 'Son', 'Daughter'],
    'Birthyear': [1976, 1984, 2013, 2016],
})
```

	Name	Status	Birthyear
0	Luke	Father	1976
1	Gina	Mother	1984
2	Sam	Son	2013

Lambda with Apply() function by Pandas. This function applies an operation to every element of the column.

To get the current age of each member, we subtract their birth year from the current year. In the lambda function below, x refers to a value in the birthyear column, and the expression is `2021(current year) minus the value.`

```
df['age'] = df['Birthyear'].apply(lambda x: 2021-x)
```

	name	Status	Birthyear	age
0	Luke	Father	1976	45
1	Gina	Mother	1984	37
2	Sam	Son	2013	8
3	Emma	Daughter	2016	5

Lambda with Python's Filter() function. This takes 2 arguments; one is a lambda function with a condition expression, two an iterable which for us is a series object. It returns a list of values that satisfy the condition.

```
list(filter(lambda x: x>18, df['age']))

###Results
[45, 37]
```

Lambda with Map() function by Pandas. Map works very much like apply() in that it modifies values of a column based on the expression.

```
#Double the age of everyone

df['double_age'] =
df['age'].map(lambda x: x*2)
```

1	gina	mother	1984	37	74
2	sam	son	2013	8	16
3	emma	daughter	2016	5	10

We can also perform **conditional operations** that return different values based on certain criteria.

The code below returns 'Male' if the Status value is father or son, and returns 'Female' otherwise. Note that `apply` and `map` are interchangeable in this context.

```
#Conditional Lambda statement

df['Gender'] = df['Status'].map(lambda x: 'Male' if x=='father' or
x=='son' else 'Female')
```

	Name	Status	Birthyear	age	double_age	Gender
0	luke	father	1976	45	90	Male
1	gina	mother	1984	37	74	Female
2	sam	son	2013	8	16	Male
3	emma	daughter	2016	5	10	Female

4. Lambda on Dataframe object

I mostly use Lambda functions on specific columns (series object) rather than the entire data frame, unless I want to modify the entire data frame with one expression.

For example rounding all values to 1 decimal place, in which case all the columns have to be float or int datatypes because round() can't work on strings.

```
df2.apply(lambda x:round(x,1))

##Returns an error if some
##columns are not numeric
```

applied column-wise.

```
#convert to lower-case

df[['Name','Status']] =
df.apply(lambda x: x[['Name','Status']].str.lower(), axis=1)
```

	Name	Status	Birthyear	age
0	luke	father	1976	45
1	gina	mother	1984	37
2	sam	son	2013	8
3	emma	daughter	2016	5

Discouraged use cases

1. **Assigning a name to a Lambda function.** This is discouraged in the PEP8 python style guide because Lambda creates an anonymous function that's not meant to be stored. Instead, use a normal `def` function if you want to store the function for reuse.

```
#Bad
triple = lambda x: x*3

#Good
def triple(x):
     return x*3
```

2. **Passing functions inside Lambda functions.** Using functions like `abs` which only take one number- argument is unnecessary with Lambda because you can directly pass the function into map() or apply().

```
#Bad
map(lambda x:abs(x), list_3)
```

```
#Good
map(lambda x: pow(x, 2), float_nums)
```

Ideally, functions inside lambda functions should take two or more arguments. Examples are `pow(number,power)` and `round(number,ndigit).` You can experiment with various in-built python functions to see which ones need Lambda functions in this context. I've done so in this notebook.

3. **Using Lambda functions when multiple lines of code are more readable**. An example is when you are using if-else statements inside the lambda function. I used the example below earlier in this article.

```
#Conditional Lambda statement

df['Gender'] = df['Status'].map(lambda x: 'Male' if x=='father' or
x=='son' else 'Female')
```

The same results can be achieved with the code below. I prefer this way because you can have endless conditions and the code is simple enough to follow. More on vectorized conditions here.

```
df['Gender'] = ''
df.loc[(df['Status'] == 'father') | (df['Status'] == 'son'), 'Gender'] = 'Male'
df.loc[(df['Status'] == 'mother') | (df['Status'] == 'daughter'), 'Gender'] = 'Female'
```

Conclusion

Many programmers who don't like Lambdas usually argue that you can replace them with the more understandable list comprehensions, built-in functions, and standard libraries. Generator expressions (similar to list comprehensions) are also handy alternatives to the map() and filter() functions.

Whether or not you decide to embrace Lambda functions in your code, you need to understand what they are and how they are used because you will inevitably come across them in other peoples' code.

Check out the code used here in my GitHub. Thank you for reading!

- https://www.analyticsvidhya.com/blog/2020/03/what-are-lambda-functions-in-python/
- Overusing lambda expressions in Python