**Lambda Functions in Python: What Are They Good For?**

The lambda keyword in Python provides a shortcut for declaring small anonymous functions. Lambda functions behave just like regular functions declared with the def keyword. They can be used whenever function objects are required.

For example, this is how you’d define a simple lambda function carrying out an addition:

>>> add = lambda x, y: x + y

>>> add(5, 3)

8

You could declare the same add function with the def keyword:

>>> def add(x, y):

... return x + y

>>> add(5, 3)

8

Now you might be wondering: **Why the big fuss about lambdas?** If they’re just a slightly more terse version of declaring functions with def, what’s the big deal?

Take a look at the following example and keep the words function expression in your head while you do that:

>>> (lambda x, y: x + y)(5, 3)

8

Okay, what happened here? I just used lambda to define an “add” function inline and then immediately called it with the arguments 5 and 3.

Conceptually the lambda expression lambda x, y: x + y is the same as declaring a function with def, just written inline. The difference is I didn’t bind it to a name like add before I used it. I simply stated the expression I wanted to compute and then immediately evaluated it by calling it like a regular function.

Before you move on, you might want to play with the previous code example a little to really let the meaning of it sink in. I still remember this took me a while to wrap my head around. So don’t worry about spending a few minutes in an interpreter session.

There’s another syntactic difference between lambdas and regular function definitions: Lambda functions are restricted to a single expression. This means a lambda function can’t use statements or annotations—not even a return statement.

How do you return values from lambdas then? Executing a lambda function evaluates its expression and then automatically returns its result. So there’s always an implicit return statement. That’s why some people refer to lambdas as single expression functions.

### Lambdas You Can Use

When should you use lambda functions in your code? Technically, any time you’re expected to supply a function object you can use a lambda expression. And because a lambda expression can be anonymous, you don’t even need to assign it to a name.

This can provide a handy and “unbureaucratic” shortcut to defining a function in Python. My most frequent use case for lambdas is writing short and concise key funcs for [sorting iterables by an alternate key](https://dbader.org/blog/python-min-max-and-nested-lists):

>>> sorted(range(-5, 6), key=lambda x: x \*\* 2)

[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]

Like regular nested functions, lambdas also work as lexical closures.

What’s a lexical closure? Just a fancy name for a function that remembers the values from the enclosing lexical scope even when the program flow is no longer in that scope. Here’s a (fairly academic) example to illustrate the idea:

>>> def make\_adder(n):

... return lambda x: x + n

>>> plus\_3 = make\_adder(3)

>>> plus\_5 = make\_adder(5)

>>> plus\_3(4)

7

>>> plus\_5(4)

9

In the above example the x + n lambda can still access the value of n even though it was defined in the make\_adder function (the enclosing scope).

Sometimes, using a lambda function instead of a nested function declared with def can express one’s intent more clearly. But to be honest this isn’t a common occurrence—at least in the kind of code that I like to write.

### But Maybe You Shouldn’t…

Now on the one hand I’m hoping this article got you interested in exploring Python’s lambda functions. On the other hand I feel like it’s time to put up another caveat: Lambda functions should be used sparingly and with extraordinary care.

I know I wrote my fair share of code using lambdas that looked “cool” but was actually a liability for me and my coworkers. If you’re tempted to use a lambda spend a few seconds (or minutes) to think if this is really the cleanest and most maintainable way to achieve the desired result.

For example, doing something like this to save two lines of code is just silly. Sure, it technically works and it’s a nice enough “trick”. But it’s also going to confuse the next gal or guy having to ship a bugfix under a tight deadline:

# Harmful:

>>> class Car:

... rev = lambda self: print('Wroom!')

... crash = lambda self: print('Boom!')

>>> my\_car = Car()

>>> my\_car.crash()

'Boom!'

I feel similarly about complicated map() or filter() constructs using lambdas. Usually it’s much cleaner to go with a list comprehension or generator expression:

# Harmful:

>>> list(filter(lambda x: x % 2 == 0, range(16)))

[0, 2, 4, 6, 8, 10, 12, 14]

# Better:

>>> [x for x in range(16) if x % 2 == 0]

[0, 2, 4, 6, 8, 10, 12, 14]

If you find yourself doing anything remotely complex with a lambda expression, consider defining a real function with a proper name instead.

Saving a few keystrokes won’t matter in the long run. Your colleagues (and your future self) will appreciate clean and readable code more than terse wizardry.

### Things to Remember

* Lambda functions are single-expression functions that are not necessarily bound to a name (anonymous).
* Lambda functions can’t use regular Python statements and always include an implicit return statement.
* Always ask yourself: Would using a regular (named) function or a list/generator expression offer more clarity?