Yet Another Lambda Tutorial

Posted on **2011/08/29**

There are a lot of tutorials[1] for Python's *lambda* out there. A very helpful one is Mike Driscoll's discussion of lambda on the Mouse vs Python blog. Mike's discussion is excellent: clear, straight-forward, with useful illustrative examples. It helped me — finally — to grok lambda, and led me to write **yet another** lambda tutorial.

Lambda is a tool for building functions

Lambda is a tool for building functions, or more precisely, for building function objects. That means that Python has two tools for building functions: *def* and *lambda*.

Here's an example. You can build a function in the normal way, using *def*, like this:

```
def square_root(x): return math.sqrt(x)
```

or you can use *lambda*:

```
square_root = lambda x: math.sqrt(x)
```

Here are a few other interesting examples of lambda:

```
sum = lambda x, y:   x + y   #  def sum(x,y): return x + y

out = lambda   *x:   sys.stdout.write(" ".join(map(str,x)))

lambda event, name=button8.getLabel(): self.onButton(event, name)
```

What is lambda good for? Why do we need lambda?

Actually, we don't absolutely need lambda; we could get along without it. But there are certain situations where it makes writing code a bit easier, and the written code a bit cleaner. What kind of situations? … Situations in which (a) the function is fairly simple, and (b) it is going to be used only once.

Normally, functions are created for one of two purposes: (a) to reduce code duplication, or (b) to modularize code.

- If your application contains duplicate chunks of code in various places, then you can put one copy of that code into a function, give the function a name, and then — using that function name — call it from various places in your code.
- If you have a chunk of code that performs one well-defined operation — but is really long and gnarly and interrupts the otherwise readable flow of your program — then you can pull that long gnarly code out and put it into a function all by itself.

But suppose you need to create a function that is going to be used only once — called from *only one* place in your application. Well, first of all, you don't need to give the function a name. It can be "anonymous". And you can just define it right in the place where you want to use it. That's where lambda is useful.

But, but, but… you say.

- First of all — Why would you want a function that is called only once? That eliminates reason (a) for making a function.
- And the body of a lambda can contain only a single expression. That means that lambdas must be short. So that eliminates reason (b) for making a function.

What possible reason could I have for wanting to create a short, anonymous function?

Well, consider this snippet of code that uses lambda to define the behavior of buttons in a Tkinter GUI interface. (This example is from Mike's tutorial.)

```
1  frame = tk.Frame(parent)
2  frame.pack()
3
4  btn22 = tk.Button(frame,
5          text="22", command=lambda: self.printNum(22))
6  btn22.pack(side=tk.LEFT)
7
8  btn44 = tk.Button(frame,
9          text="44", command=lambda: self.printNum(44))
10 btn44.pack(side=tk.LEFT)
```

The thing to remember here is that a *tk.Button* expects a **function object** as an argument to the *command* parameter. That function object will be the function that the button calls when it (the button) is clicked. Basically, that function specifies what the GUI will do when the button is clicked.

So we must pass a function object in to a button via the *command* parameter. And note that — since different buttons do different things — we need a different function object for each button object. Each function will be used only once, by the particular button to which it is being supplied.

So, although we *could* code (say)

```
1  def __init__(self, parent):
2      """Constructor"""
```

```
        frame = tk.Frame(parent)
        frame.pack()

        btn22 = tk.Button(frame,
            text="22", command=self.buttonCmd22)
        btn22.pack(side=tk.LEFT)

        btn44 = tk.Button(frame,
            text="44", command=self.buttonCmd44)
        btn44.pack(side=tk.LEFT)

    def buttonCmd22(self):
        self.printNum(22)

    def buttonCmd44(self):
        self.printNum(44)
```

it is much easier (and clearer) to code

```
    def __init__(self, parent):
        """Constructor"""
        frame = tk.Frame(parent)
        frame.pack()

        btn22 = tk.Button(frame,
            text="22", command=lambda: self.printNum(22))
        btn22.pack(side=tk.LEFT)

        btn44 = tk.Button(frame,
            text="44", command=lambda: self.printNum(44))
        btn44.pack(side=tk.LEFT)
```

When a GUI program has this kind of code, the button object is said to “call back” to the function object that was supplied to it as its *command*. So we can say that one of the most frequent uses of lambda is in coding “callbacks” to GUI frameworks such as Tkinter and wxPython.

This all seems pretty straight-forward. So…

Why is lambda so confusing?

There are four reasons that I can think of.

First *Lambda is confusing because:* the requirement that a lambda can take only a single *expression* raises the question: *What is an expression?*

A lot of people would like to know the answer to that one. If you Google around a bit, you will see a lot of posts from people asking “In Python, what’s the difference between an expression and a statement?”

One good answer is that an expression returns (or evaluates to) a value, whereas a statement does not. Unfortunately, the situation is muddled by the fact that in Python an expression can also be a statement. And we can always throw a red herring into the mix — assigment statements like `a = b = 0` suggest that Python supports *chained assignments*, and that assignment statements return values. (They do not. Python isn’t C.)[2]

In many cases when people ask this question, what they really want to know is: *What kind of things can I, and can I not, put into a lambda?* And the answer to *that* question is basically—

- If it doesn't return a value, it isn't an expression and can't be put into a lambda.
- If you can imagine it in an assignment statement, on the right-hand side of the equals sign, it is an expression and can be put into a lambda.

Using these rules means that:

1. Assignment statements cannot be used in lambda. In Python, assignment statements don't return anything, not even None (null).
2. Simple things such as mathematical operations, string operations, list comprehensions, etc. are OK in a lambda.
3. Function calls are expressions. It is OK to put a function call in a lambda, and to pass arguments to that function. Doing this wraps the function call (arguments and all) inside a new, anonymous function.
4. In Python 3, *print* became a function, so in Python 3+, *print(…)* can be used in a lambda.
5. Even functions that return None, like the *print* function in Python 3, can be used in a lambda.
6. Conditional expressions, which were introduced in Python 2.5, are expressions (and not merely a different syntax for an *if/else* statement). They return a value, and can be used in a lambda.

```
lambda: a if some_condition() else b
lambda x: 'big' if x > 100 else 'small'
```

Second *Lambda is confusing because:* the specification that a lambda can take only a *single* expression raises the question: *Why? Why only one expression? Why not multiple expressions? Why not statements?*

For some developers, this question means simply *Why is the Python lambda syntax so weird?* For others, especially those with a Lisp background, the question means *Why is Python's lambda so crippled? Why isn't it as powerful as Lisp's lambda?*

The answer is complicated, and it involves the "pythonicity" of Python's syntax. Lambda was a relatively late addition to Python. By the time that it was added, Python syntax had become well established. Under the circumstances, the syntax for lambda had to be shoe-horned into the established Python syntax in a "pythonic" way. And that placed certain limitations on the kinds of things that could be done in lambdas. Frankly, I still think the syntax for lambda looks a little weird. Be that as it may, Guido has explained why lambda's syntax is not going to change. Python will not become Lisp.[3]

Third *Lambda is confusing because:* lambda is usually described as a tool for creating functions, but a lambda specification does not contain a *return* statement.

The *return* statement is, in a sense, implicit in a lambda. Since a lambda specification must contain only a single expression, and that expression must return a value, an anonymous function created by lambda implicitly returns the value returned by the expression. This makes perfect sense. Still—the lack of an explicit *return* statement is, I think, part of what makes it hard to grok lambda, or at least, hard to grok it quickly.

Fourth *Lambda is confusing because:* tutorials on lambda typically introduce lambda as a tool for creating anonymous *functions*, when in fact the most common use of lambda is for creating anonymous *procedures*.

Back in the High Old Times, we recognized two different kinds of subroutines: procedures and functions. Procedures were for *doing* stuff, and did not return anything. Functions were for calculating and *returning values*. The difference between functions and procedures was even built into some programming languages. In Pascal, for instance, *procedure* and *function* were different keywords.

In most modern languages, the difference between procedures and functions is no longer enshrined in the language syntax. A Python function, for instance, can act like a procedure, a function, or both. The (not altogether desirable) result is that a Python function is always referred to as a "function", even when it is essentially acting as a procedure.

Although the distinction between a procedure and a function has essentially vanished as a language construct, we still often use it when thinking about how a program works. For example, when I'm reading the source code of a program and see some function F, I try to figure out what F does. And I often can categorize it as a procedure or a function — "the purpose of F is to do so-and-so" I will say to myself, or "the purpose of F is to calculate and return such-and-such".

So now I think we can see why many explanations of lambda are confusing.

First of all, the Python language itself masks the distinction between a function and a procedure.

Second, most tutorials introduce lambda as a tool for creating anonymous *functions*, things whose primary purpose is to calculate and return a result. The very first example that you see in most tutorials (this one included) shows how to write a lambda to return, say, the square root of x.

But this is not the way that lambda is most commonly used, and is not what most programmers are looking for when they Google "python lambda tutorial". The most common use for lambda is to create anonymous *procedures* for use in GUI callbacks. In those use cases, we don't care about what the lambda *returns*, we care about what it *does*.

This explains why most explanations of lambda are confusing for the typical Python programmer. He's trying to learn how to write code for some GUI framework: Tkinter, say, or wxPython. He runs across examples that use lambda, and wants to understand what he's seeing. He Googles for "python lambda tutorial". And he finds tutorials that start with examples that are entirely inappropriate for his purposes.

So, if you are such a programmer — this tutorial is for you. I hope it helps. I'm sorry that we got to this point at the end of the tutorial, rather than at the beginning. Let's hope that someday, someone will write a lambda tutorial that, instead of beginning this way

Lambda is a tool for building anonymous functions.

begins something like this

Lambda is a tool for building callback handlers.

So there you have it. Yet another lambda tutorial.

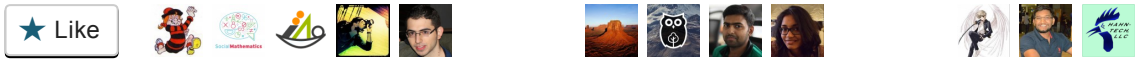
Footnotes

[1] Some lambda tutorials:

- Mike's discussion of lambda on the Mouse vs Python blog is useful. Check out the *Further Reading* links at the bottom of the post. The comments describe a couple of other interesting use cases for lambda.
- Python tutorial discussion of lambda
- Stupid lambda tricks
- Bogotobogo tutorial discusses some interesting uses of lambda (e.g. in coding **jump tables**) in the *Why lambda?* section

[2] In some programming languages, such as C, an assignment statement returns the assigned value. This allows *chained assignments* such as `x = y = a`, in which the assignment statement `y = a` returns the value of `a`, which is then assigned to `x`. In Python, assignment statements do **not** return a value. Chained assignment (or more precisely, code that *looks like* chained assignment statements) is recognized and supported as a special case of the assignment statement.

[3] Python developers who are familiar with Lisp have argued for increasing the power of Python's lambda, moving it closer to the power of lambda in Lisp. There have been a number of proposals for a syntax for "multi-line lambda", and so on. Guido has rejected these proposals and blogged about some of his thinking about "pythonicity" and language features as a user interface. This led to an interesting discussion on *Lambda the Ultimate, the programming languages weblog* about lambda, and about the idea that programming languages have personalities.



19 bloggers like this.

RELATED

A Globals Class pattern for Python
In "Python Globals"

Python Decorators
In "Decorators"

Gotcha — Mutable default arguments
In "Python gotchas"

This entry was posted in **Python features** by **Steve Ferg**. Bookmark the **permalink [https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/]** .

17 THOUGHTS ON “YET ANOTHER LAMBDA TUTORIAL”

Alex Clark (@aclark4life)
on **2011/08/29 at 8:18 pm** said:

Thanks! This was really helpful.

Roger Matelot
on **2011/09/04 at 3:04 am** said:

need examples !

Roger Matelot
on **2011/09/04 at 12:27 pm** said:

per your example under “So, although we could code (say)”

…….
command=self.buttonCmd22
……..

can’t one code it like this ? :
“command=self.buttonCmd(22)”….”command=self.buttonCmd(44)”

Steve Ferg
on **2011/09/06 at 1:48 am** said:

Q:
although we could code (say)

command=self.buttonCmd22

can’t one code it like this?

command=self.buttonCmd(22)

A:
No. This

command=self.buttonCmd22

assigns the self.buttonCmd22 method to command. Basically what is assigned to command is a function object.

On the other hand, this

command=self.buttonCmd(22)

calls the self.buttonCmd method, passing it 22 as an argument. The call will probably crash because self.buttonCmd doesn’t accept any arguments. But

assuming that it doesn't crash, the call returns something (in this case, probably None) as a result, and that result is what is assigned to command.

This is one way in which Python differs from (for example) Ruby. In Python, when dealing with methods and functions, parentheses matter.

Suppose we have a function foobar:

```
def foobar():
    return 5
```

Now this

$x = foobar$

binds the name x to the foobar function. Whereas this

$x = foobar()$

calls foobar(), and binds x to the result that foobar returns.

After the first statement, "x" and "foobar" are names that are bound to the same value, which is a function object.

After the second statement, the value of x is 5, the value returned by a call to foobar().

Anonymous
on **2011/09/14 at 11:26 am** said:

Hi,

Thank you for the good post. I would like to point one issue though,

>> "In Python, statements don't return anything"

This is kind of misleading, because it gives the wrong impression about Expressions and Statements being mutually exclusive. Expressions in python are just a sub-set of statements. In fact they are referred to as "Expression Statements"

Link: http://docs.python.org/reference/simple_stmts.html#expression-statements

Steve Ferg
on **2011/10/11 at 9:52 pm** said:

Thanks. There was a typo in the line, which should have read “In Python, **assignment** statements don’t return anything”. Fixed now.

driscollis
on **2011/09/20 at 9:27 pm** said:

Thanks for your readership. I’m glad you enjoyed my explanation of lambda on MvP. It’s always nice to know who likes what.

youngungjeong
on **2012/01/11 at 10:44 pm** said:

Very interesting and helpful explanation, it was. Thanks for the scratch for my long-been itch.

agentultra
on **2012/01/12 at 10:47 am** said:

An entire programming language can be specified entirely with lambda (providing the implementation is sound… Python’s is rather limiting). See the Y combinator for an example.

While lambda can be confusing for beginning programmers, especially those without a mathematics background, is there any risk in exposing them to the bigger picture in an

introductory tutorial? Maybe a footnote?

I just love lambda and wish it was better supported in Python (even though it will probably never be).

Nice article! 🙂

Steve Ferg
on **2012/03/08 at 11:55 pm** said:

James wrote

>> But lambda is so much more than just a keyword in Python for creating anonymous function objects!

I agree. I focused on *lambda-the-Python-progamming-language-keyword* because that (and its documentation) was the source of the problems that I wanted to address. In footnote 3, I made a nod to situating that topic in a larger context. But, well, Alonzo Church was definitely out-of-scope.

But James is right. And **"lambda — more than a Python keyword"** would be a great topic for a post of its own. Unfortunately, I'm not competent to write it.

I know that the word "lambda" can refer to a lot of different things.

- lambda-in-Python, i.e. lambda-the-Python-progamming-language-keyword
- lambda-in-Lisp
- lambda-the-programming-language-concept
- lambda-the-logical-concept
- … *etc. etc.* …

And I've found a few articles:

- In wikipedia, what you really want to look at is the article on anonymous functions. And maybe the article on first-class functions.
- Surprisingly, the Wikipedia article on lambda in programming languages is very weak.
- Wikipedia has an article on the lambda calculus that will likely be too far afield to be interesting or helpful for most programmers.
- For the really hardy, there is an article on the Y combinator.

But those are encyclopedia articles. Pretty dry. What I'd **really** like to learn more about is what makes programmers like James, who use lambda on a regular basis, love lambda.

So… are there any readers out there who might have recommendations and/or links to books, articles, blog postings, tutorials, or web sites that would help those of us (the ones familiar with lambda only in the context of Python) to see lambda in a larger context? If so, would you be willing to share them? You can code raw HTML into a comment, so you can code an HTML link by doing something that looks like this:

```
There is a great introduction to lambda for programmers at <a
href="....">Lambdas R Us</a>
```

agentultra
on **2012/04/25 at 11:01 am** said:

My favourite introduction was in The Little Schemer. It provides a very logical progression from fundamentals into lambda's and ends with the Y-Combinator… in an introductory text that, IMO, is really easy to follow (and fun!).

I think Python's lambda is probably somewhat confusing because of its limitations. For example, you can write the Y Combinator in Python but the limitations of lambda in Python make that a very difficult bit of code to read and understand. In languages with better lambda support (like Javascript, Scheme, Lisp, etc) such concepts are pretty natural.

Lambda is pretty useful for encapsulating procedures related to a single function without exposing them in the namespace of the module. Python's conventions for working around this limitation are pretty ugly, IMO. One either prefixes such supporting "functions" with an underscore or they exclude them from the __all__ special module-level attribute (otherwise they create more classes or just let such trivial bits of code float freely in their module name space).

Lambda is also pretty useful for expressing alternate control flows irrc, but I think that's something you can read about in the Seasoned Schemer.

Dac
on **2012/01/12 at 12:50 pm** said:

For me, the most common use case for lambda is when using python's function programming tools, such as map, reduce, etc.

```
class Fraction:
     def __init__(self, num,denom):
        self.num = num
        self.denom = denom

fractions = [Fraction(1,2), Fraction(2,3), Fraction (5,7)]
sum_num = reduce(fractions, lambda x,y:x.num+y.num, 0)
```

Of course this is a contrived example (why would you need to sum numerators?), but this style of usage is most common for me.

Wouter

on **2012/01/13 at 6:49 am** said:

I think that a lot of applications of lambda, including your callback examples, are situations in which you want to specify an extra argument to an existing "real" function.

command=lambda: self.printNum(44)) # calls self.printNum with an extra argument (44)

For this, functools provides a function that provides the same functionality: partial

command=partial(self.printNum, 44)

This is less general than lambda: any partial can be rewritten as a lambda statement. partial(func, *args, **kargs) is simply lambda *args, **kargs: func(*args, **kargs). However, partial does not depend on a special syntax that is considered difficult to understand.

IMHO, the remaining use cases for lambda are so limited that 'they' might as well remove it from the language.

The alternative would be changing function (and class?) definitions to become proper expressions, eg rewriting def function(args) as function = def(args), but that would have very serious implications for the whole language and might make it more difficult to read/learn rather than easier.

Anonymous

on **2012/07/07 at 2:29 pm** said:

This article is a most perfect example of how meta data is needed when explaining new concepts. After reading this once I understand lambdas having never worked with one before.

> Anonymous
> on **2012/07/07 at 2:31 pm** said:
>
> Although, I was confused when the author advised lambdas are about returning things implicitly, then going on to say how lambas are like procedures, which just do work and don't return anything.

lefönk
on **2012/07/13 at 12:19 pm** said:

"And he finds tutorials that start with examples that are entirely inappropriate for his purposes." –> Your just so right!
"So, if you are such a programmer — this tutorial is for you." –> Thanks! you nailed it!

Sanjay Kannan
on **2012/08/11 at 3:44 pm** said:

I've used some hack-ish techniques to create a multi-line lambda (https://github.com/whaatt/Mu) in Python, for those who actually need true anonymous functions. Great tutorial though; I agree that lambda is often misconstrued.

Comments are closed.