LINUX VOICE
TUTORIAL

CODE NINJA: LAMBDA FUNCTIONS

BEN EVERARD

Anonymous functions aren't just 4Chan meetups – they're also a way to create cleaner code.

WHY DO THIS?
- Write cleaner code.
- Understand one of the formal underpinnings of computation.
- Sound clever in conversations with other programmers.

If we were trying to come up with a name to make something sound excessively mathematical, we couldn't do better than lambda calculus. The phrase conjures up a picture of a stern-faced maths teacher peering over his glasses while wearing a tweed jacket with leather-patched elbows.

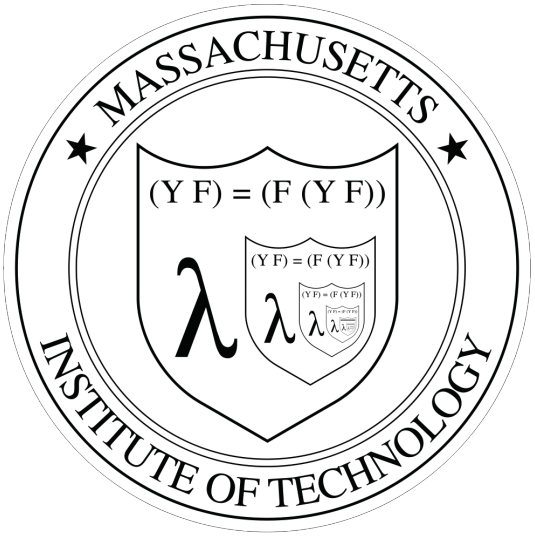
The reason it sounds so confusing is because it hails from a time when computing was little more than an obscure branch of maths that only interested academics and the military.

Lambda calculus was created as mathematicians were struggling to understand computation, and what its limits were. It's a very simple way of specifying programs, and because it's simple, it's easy to reason about mathematically.

Basically, lambda calculus is a way of forming computer programs out of functions with two restrictions. Firstly, the functions don't have a name, and secondly they can only take one argument. Functions that follow these rules are known as Lambda functions. Let's take a look at this in Python, which supports lambda functions with the lambda statement:

```
>>> add2 = lambda x: x+2
```

This creates a function that takes one argument (**x**) and returns the number **x+2**. Python imposes additional restrictions on lambda functions: they can only contain one statement and that statement must return a value (which not all statements do in Python).



The badge of the Knights of the Lambda Calculus – a band of Lisp programmers who wait for the day when a well-placed anonymous function will save the world.

Because they only contain one statement, they don't need the return keyword to specify what they return. Whatever is after the colon is the statement, and the function will return whatever it evaluates to.

In this case, we've assigned the function to a variable called **add2**. You don't have to assign the function to a variable and most of the time it's more useful not to (remember that we said functions don't

Church-Turing thesis

We've looked at lambda functions in Python where they're a convenient shorthand for creating functions to be used only once. However, the basic purpose of Lambda calculus wasn't to add syntactical simplicity to high level languages. It was to help understand computation.

One of the big problems in early computer science was working out what could be computed and what couldn't. Alonzo Church worked with lambda calculus as Alan Turing worked with Turing machines.

It's possible to show that anything computable using a Turing Machine is computable using lambda calculus and vice versa. It's also possible to prove that some things can't be computed using Turing Machines or lambda calculus. For example, the halting problem can't be computed. This means that it's impossible to write a program that takes another program as input and works out whether or not it will finish running, or not (eg whether it will get stuck in an infinite loop).

The Church-Turing thesis states that anything that can be computed by a computer can be computed using lambda calculus or a Turing Machine. However, this problem remains stubbornly a thesis and has never been formally proven. Since lambda calculus can implement anything that a Turing machine can, lambda calculus is known as Turing-complete. If the Church-Turing thesis is correct, any language that is Turing complete can compute anything that is computable. All general-purpose languages are Turing complete – as you would expect – but so are some languages that are quite restrictive. For example, **sed** is Turing complete (see **www.robertkotcher.com/sed.html** for proof). Some more powerful markup languages are also Turing complete, such as HTML5 + CSS3 (**https://github.com/elitheeli/stupid-machines**) and C++ templates (**http://ubietylab.net/ubigraph/content/Papers/pdf/CppTuring.pdf**).

The creativity of geeks knows no bounds, and it's become a challenge to prove ever more obscure things are Turing complete. *Minecraft* is Turing complete (**www.youtube.com/watch?v=1X21HQphy6I**) and so is an infinite version of *Minesweeper* (**http://web.mat.bham.ac.uk/R.W.Kaye/minesw/infmsw.pdf**), but the most bizarre thing we could find that is Turing complete is the *Magic: The Gathering* card game (**www.toothycat.net/~hologram/Turing/HowItWorks.html**). If the Church-Turing thesis is correct, this means that it's possible to port any computer program to run on the *Magic: The Gathering* card game. Weird, huh?

have names?), but we'll get onto that in a bit. You can run the function with:

```
>>> add2(1)
3
```

So far, this just looks like a slightly awkward way of creating functions. You could be forgiven for wondering why Python includes this slightly odd theoretical concept. One of the advantages of lambda functions in Python is that they can be a very convenient way of specifying a function that will only be used once. Typically, this when a function is needed as a parameter.

For example, take a look at the following function from the *XBMC* remote elsewhere in this issue's coding section:

```
def get_artists():
        data = xbmc.AudioLibrary.GetArtists()
        return sorted(data['result']['artists'], key=lambda k:
k['label'])
```

Here, the Python function **sorted()** can take an argument called **key** which specifies a function that is called on each element to be sorted that returns the value that the items should be sorted on. In this case, key is a lambda function that takes a dictionary as its parameter and outputs the particular item from that dictionary that we want to sort on. We could define a function in the usual Python way (by using **def** and giving it a name). However the lambda notation is clearer and simpler.

Hello again, Mr Turing!

Lambda calculus wasn't created as a convenient shorthand. It was created as a method of defining computation. Like Turing machines, lambda calculus is a computationally complete language. That means that anything that can be computed, can be defined using lambda calculus (not necessarily in Python's restricted version of it though).

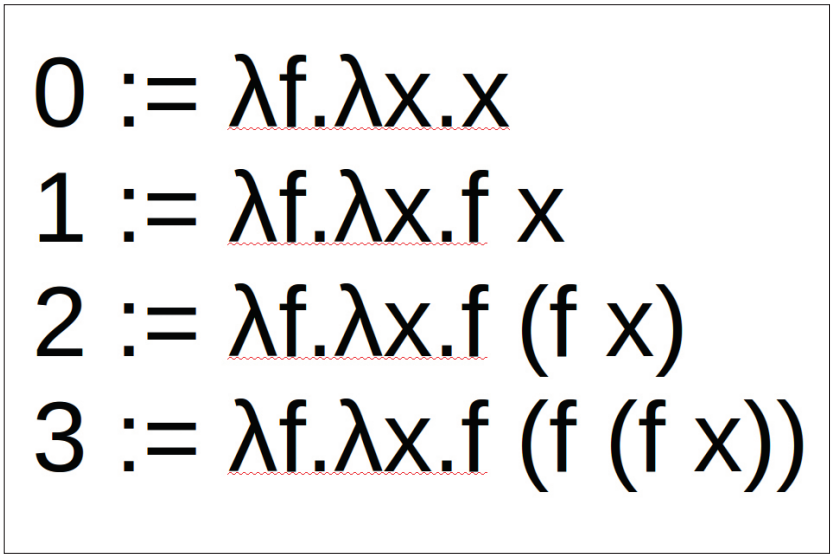
Obviously this isn't possible if each function can only operate on a single value. Lambda calculus also allows chaining of functions to build up more complex operations. For example, you could create a function to add two values together with:

```
>>> add = lambda x: lambda y: y+x
>>> add(3)(2)
5
```

Beyond Python

Most programming languages allow anonymous functions (you can argue about whether an anonymous function with more than one argument is really a lambda function). The only commonly used general purpose languages without them are C (though they are supported in Clang) and Fortran. No other common language has the single statement restriction of Python.

The syntax and terminology varies from language to language, but they're usually used for cases similar to those we've looked at here when functions need passing as arguments in other functions, particularly in callbacks (which we looked at in LV007).



Lambda calculus gets its name from the lower-case Greek letter lambda, which is used to denote anonymous functions. It's shown here calculating the Church numerals.

This chaining – also known as currying – enables you to build up functions of arbitrary complexity. It also enables you to build functions by fixing particular parameters in other lambda functions. For example (following on from the previous session):

```
>>> add10 = add(10)
>>> add10(1)
11
```

This is rarely used in Python, but it can be used in a few ways. For example, we could use it to create logging functions for system and application errors in Python 3:

```
>>> p_log = lambda er: lambda msg: print(er, msg)
>>> p_sys_err = p_log("System error:")
>>> p_app_err = p_log("Application error:")
>>> p_sys_err("operating system problem")
System error: operating system problem
>>> p_app_err("the application has crashed")
Application error: the application has crashed
```

You need to use Python3 because in previous versions of Python, **print()** didn't return a value, and so couldn't be used as a lambda statement (in Python3, **print()** is a function that returns **None**).

In Python, the restriction to only one statement means you can't loop through data, since there can't be any code blocks. However, you can still use **if** statements using a slightly different format:

```
x if <conditions> else y
```

For example, you could use this to return the lowest number in a pair using:

```
>>> min = lambda x: x[0] if x[0]<x[1] else x[1]
>>> min([3,5])
>>> 3
```

Python doesn't need lambda functions. Everything you do with them could also be achieved without them. However, there are several places where they can be used to make your code more readable. This is usually in places where a function object is passed (like in the **sort** example above).