Why is lambda so confusing?

From: https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/

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 for *that* question, I think a few simple rules of thumb will be sufficient.

- 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.