What is Python lambda?

Before trying to understand what, a Python lambda is, let's first try to understand what a Python function is at a much deeper level.

This will require a little bit of a paradigm shift of how you think about functions.

As you already know, everything in Python is an **object**.

For example, when we run this simple line of code.

```
x = 5
```

What happens is we're creating a Python object of type int that stores the value 5.

x is essentially a symbol that is referring to that object.

Now let's check the type of x and the address it is referring to.

We can easily do that using the type and the id built-in functions

```
>>> type(x)
<class 'int'>
>>> id(x)
4308964832
```

As you can see, x refers to an object of type **int** and this object lives in the address returned by the id

Pretty straightforward stuff.

Now what happens when we define a function like this one:

```
>>> def f(x):
...   return x * x
...
```

Let's repeat the same exercise from above and inspect the **type** of **f** and its **id**.

```
>>> def f(x):
...   return x * x
...
>>> type(f)
<class 'function'>
>>> id(f)
4316798080
```

Very interesting.

So, it turns out there is a **function** class in Python and the function f that we just defined is an instance of that class.

Exactly like how x was an instance of the **integer** class.

In other words, you can literally think about functions the same way you think about variables.

The only difference is that a variable stores data whereas a function stores code.

That also means you can pass functions as arguments to other functions, or even have a function be the return value of another function.

let's look at a simple example where you can pass the above function f to another function.

```
def f(x):
    return x * x

def modify_list(L, fn):
    for idx, v in enumerate(L):
        L[idx] = fn(v)

L = [1, 3, 2]
modify_list(L, f)
print(L)

#output: [1, 9, 4]
```

Give yourself a minute and try to understand what this code does before you read on…

As you can see, **modify_list** is a function that takes a list L and a function fn as arguments.

It then iterates over the list item-by-item and applies the function **fn** on each.

This is a very generic way of modifying the items of a list as it allows you to pass in the function that is responsible for the modification which can be very useful as you will see later.

So, for example when we pass the function f to **modify_list**, the result will be that each item in the list will be squared.

We could pass any other custom function that can modify the list in any arbitrary way.

That's powerful stuff right there!

Alright now that I have laid down some foundations, let's talk about lambdas.

A ***Python lambda*** is just another method to define a *function*.

The general syntax of a Python lambda is:

```
lambda arguments: expression
```

Lambda functions can accept **zero** or **more** arguments but **only one** expression.

The return value of the lambda function is the value that this expression is evaluated to.

For example, if we want to define the same function **f** that we defined before using lambda syntax, this is how it will look like:

```
>>> f = lambda x: x * x
>>> type(f)
<class 'function'>
```

But you might be asking yourself why the need for lambdas in the first place when we can just define functions the *traditional way?*

Good question!

Lambdas are only useful when you want to define a one-off function.

In other words, a function that will be used only once in your program. These functions are called *anonymous functions.*

As you will see later, there are many situations where anonymous functions can be useful.

Lambdas with multiple arguments

As you saw earlier, it was easy to define a lambda function with one argument.

```
>>> f = lambda x: x * x
>>> f(5)
25
```

But if you want to define a lambda function that accepts more than one argument, you can separate the input arguments by commas.

For example, say we want to define a lambda that takes two integer arguments and returns their product.

```
>>> f = lambda x, y: x * y
>>> f(5, 2)
10
```

Nice!

How about if you want to have a lambda that accepts no arguments whatsoever?

Lambdas with no arguments

Say you want to define a lambda function that takes no arguments and returns **True**.

You can achieve this with the following code.

```
>>> f = lambda: True
>>> f()
True
```

Multiline lambdas

Yes, at some point in your life you will be wondering if you can have a lambda function with multiple lines.

And the answer is:

No, you can't 🙂

Python lambda functions accept only one and only one expression.

If your function has multiple expressions/statements, you are better off defining a function the traditional way instead of using lambdas.