# lambda

December 13, 2017

## 1 This is the write up for the 'lambda' function in python

The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without name, at run time. These functions are throw-away functions i.e. they are just needed where they have been created. Lambda fucntion are mainly used in combination with the functions filter(), map() and reduce(). The lambda features was added to the Python due to the demand from Lisp programmer.

The general syntax of a lambda function is given below: lambda arg1, arg2, ...argN : expression using arguments The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.

```
In [2]: #this function finds the square of the given number 'x':

        square = lambda x: x**2
        print ("answer:",square(10))
```

```
('answer:', 100)
```

Like def, the lambda creates a function to be called later. But it returns the function instead of assigning it to a name. This is why lambdas are sometimes known as anonymous functions. In practice, they are used as a way to inline a function definition, or to defer execution of a code.

The following code shows the difference between a normal function definition, func and a lambda function, lamb:

```
In [13]: #function definition
         def func(x):
             return x**2
         print("function:",func(2))

         #lambda function
         l = lambda x: x**2
         print("lambda:", l(2))
```

```
('function:', 4)
('lambda:', 4)
```

As we can see, func() and lamb() do exactly the same and can be used in the same ways. Note that the lambda definition does not include a return statement, it always contains an expression which is returned. Also note that we can put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all.

## 2 Why lambda?

The lambdas can be used as a function shorthand that allows us to embed a function within the code. For instance, callback handlers are frequently coded as inline lambda expressions embedded directly in a registration call's arguments list. Instead of being define with a def elsewhere in a file and referenced by name, lambdas are also commonly used to code jump tables which are lists or dictionaries of actions to be performed on demand.

```
In [14]: container = [lambda x: x ** 2,
                      lambda x: x ** 3,
                      lambda x: x ** 4]
         for f in container:
             print("Value:",f(3))

('Value:', 9)
('Value:', 27)
('Value:', 81)
```

In the example above, a list of three functions was built up by embedding lambda expressions inside a list. A def won't work inside a list literal like this because it is a statement, not an expression. If we really want to use def for the same result, we need temporary function names and definitions outside:

```
In [15]: def f1(x): return x ** 2

         def f2(x): return x ** 3

         def f3(x): return x ** 4

         # Reference by name
         container = [f1, f2, f3]
         for f in container:
             print("value:",f(3))

('value:', 9)
('value:', 27)
('value:', 81)
```

This works but our defs may be far away in our file. The code proximity that lambda provide is useful for functions that will only be used in a single context. Especially, if the three functions are not going to be used anywhere else, it makes sense to embed them within the dictionary as

lambdas. Also, the def requires more names for these title functions that may cause name clash with other names in this file.

If we need to perform loops within a lambda, we can also embed things like map calls and list comprehension expressions.

```
In [8]: import sys
        fullname = lambda x: list(map(sys.stdout.write,x))
        f = fullname(['Chanda ', 'Rai ', 'Kelam'])
```

```
Chanda Rai Kelam
```

Here is the description of map built-in function. map(function, iterable, ...) Return an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

So, in the above example, sys.stdout.write is an argument for function, and the x is an iterable item, list, in the example.

In the following example, the lambda appears inside a def and so can access the value that the name x has in the function's scope at the time that the enclosing function was called:

```
In [10]: def action(x):
             # Make and return function, remember x
             return (lambda newx: x + newx)

         ans = action(99)
         print ans
```

```
<function <lambda> at 0x7f1380461848>
```

```
In [11]: ans(100)
```

```
Out[11]: 199
```

In the above example the value passed to the function is remembered and used by the lambda function.