Python Lambda Function

Lambda is one of the most useful, important and interesting features in Python. Unfortunately, they are easy to misunderstand and get wrong.

What is a Lambda Function?

A lambda is simply a way to define a function in Python. They are sometimes known as lambda operators or lambda functions.

By now you probably have defined your functions using the `def` **keyword, and it has worked well for you so far. So why is there another way to do the same thing?**

The difference is that lambda functions are anonymous. Meaning, they are functions that do not need to be named. They are used to create small one-line functions in cases where a normal function would be an overkill.

Basic Example

Before looking at a lambda function, let's look at a super basic function defined the "traditional" way: Here is a simple function that doubles the passed value.

```
def doubler(x):
    return x*2


print(doubler(2))
# Prints 4


print(doubler(5))
# Prints 10
```

Here's how it looks as a lambda function:

```
doubler = lambda x: x*2
```

```
print(doubler(2))
# Prints 4


print(doubler(5))
# Prints 10
```

In the above example, the lambda is constructed as:

```
lambda parameters: expression
```

Note that instead of using `def`, the keyword `lambda` is used. No parentheses are required. Anything after the lambda keyword is treated as a parameter. The colon is used to separate parameters and expression. In our case, the expression is x*2.

There's no need to use the `return` keyword, the lambda does this automatically for you.

Important characteristics

In particular, a lambda function has the following characteristics:

No Statements Allowed

A lambda function can not contain any statements in its body. Statements such as return, raise, pass, or assert in a lambda function will raise a SyntaxError. Here is an example of a lambda function containing assert:

```
doubler = lambda x: assert x*2
```

Single Expression Only

Unlike a normal function, a lambda function contains only a single expression.

Although, you can spread the expression over multiple lines using parentheses or a multiline string, but it should only remain as a single expression.

```
evenOdd = (lambda x:
          'odd' if x%2 else 'even')


print(evenOdd(2))
# Prints even


print(evenOdd(3))
# Prints odd
```

Immediately Invoked Function Expression (IIFE)

A lambda function can be immediately invoked. For this reason it is often referred to as an Immediately Invoked Function Expression (IIFE).

Here’s the same previously seen ‘doubler’ lambda function that is defined and then called immediately with 3 as an argument.

```
print((lambda x: x*2)(3))
# Prints 6
```

Multiple Arguments

You can send as many arguments as you like to a lambda function; just separate them with a comma ,.

Here’s how you’d create a lambda function with multiple arguments:

```
# A lambda function that multiplies two values
mul = lambda x, y: x*y
print(mul(2, 5))
# Prints 10

# A lambda function that adds three values
add = lambda x, y, z: x+y+z
```

```
print(add(2, 5, 10))
# Prints 17
```

Ways to Pass Arguments

Like a normal function, a lambda function supports all the different ways of passing arguments. This includes:

- **Positional arguments**
- **Keyword arguments**
- **Default argument**
- **Variable list of arguments (*args)**
- **Variable list of keyword arguments (**args)**

The following examples illustrate various options for passing arguments to the lambda function.

```
# Positional arguments
add = lambda x, y, z: x+y+z
print(add(2, 3, 4))
# Prints 9

# Keyword arguments
add = lambda x, y, z: x+y+z
print(add(2, z=3, y=4))
# Prints 9

# Default arguments
add = lambda x, y=3, z=4: x+y+z
print(add(2))
# Prints 9
# *args
add = lambda *args: sum(args)
print(add(2, 3, 4))
# Prints 9


# **args
add = lambda **kwargs: sum(kwargs.values())
print(add(x=2, y=3, z=4))
# Prints 9
```

Lambdas With Map, Filter, and Reduce

The Python core library has three methods called `map()`, `filter()`, and `reduce()`. These methods are possibly the best reasons to use lambda functions.

With map()

The `map()` function expects two arguments: a function and a list. It takes that function and applies it on every item of the list and returns the modified list.

Here's a map() function without a lambda:

```
# Double each item of the list
def doubler(x):
    return x*2


L = [1, 2, 3, 4, 5, 6]
mod_list = map(doubler, L)
print(list(mod_list))
# Prints [2, 4, 6, 8, 10, 12]
```

In above example the doubler function is passed in as an argument, but what if you don't want to create a new function every time you use the `map()`? You can use a lambda instead!

```
# Double each item of the list
L = [1, 2, 3, 4, 5, 6]
doubler = map(lambda x: x*2, L)
print(list(doubler))
# Prints [2, 4, 6, 8, 10, 12]
```

As you can see, the entire doubler function is no longer needed. Instead, the lambda function is used to create more concise code.

With filter()

The `filter()` function is similar to the `map()`. It takes a function and applies it to each item in the list to create a new list with only those items that cause the function to return True.

First, without a lambda:

```
# Filter the values above 18
def checkAge(age):
    if age > 18:
        return True
    else:
        return False

age = [5, 11, 16, 19, 24, 42]
adults = filter(checkAge, age)
print(list(adults))
# Prints [19, 24, 42]
```

Here's what the above code looks like, with the checkAge function replaced by a lambda:

```
# Filter the values above 18
age = [5, 11, 16, 19, 24, 42]
adults = filter(lambda x: x > 18, age)
print(list(adults))
```

```
# Prints [19, 24, 42]
```

With reduce()

`reduce()` is another Python function. It applies a rolling calculation to all items in a list. You can use it to calculate the total sum or to multiply all the numbers together.

Here's a `reduce()` function without a lambda:

```
# sum all items in a list
from functools import reduce


def summer(a, b):
    return a + b


L = [10, 20, 30, 40]
result = reduce(summer, L)
print(result)
# Prints 100
```

There is no need for a new function here either:

```
from functools import reduce


L = [10, 20, 30, 40]
result = reduce(lambda a, b: a + b, L)
print(result)
# Prints 100
```

Return Multiple Values

To return multiple values pack them in a tuple. Then use multiple assignment to unpack the parts of the returned tuple.

```
# Return multiple values by packing them in a tuple
```

```
findSquareCube = lambda num: (num**2, num**3)
x, y = findSquareCube(2)
print(x)
# Prints 4
print(y)
# Prints 8
```

if else in a Lambda

Generally `if else` statement is used to implement selection logic in a function. But as it is a statement, you cannot use it in a lambda function. You can use the if else ternary expression instead.

```
# A lambda function that returns the smallest item
findMin = lambda x, y: x if x < y else y

print(findMin(2, 4))
# Prints 2

print(findMin('a', 'x'))
# Prints a
```

List Comprehension in a Lambda

List comprehension is an expression, not a statement, so you can safely use it in a lambda function.

```
# Flatten a nested list with lambda
flatten = lambda l: [item for sublist in l for item in sublist]

L = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
print(flatten(L))
# Prints [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
L = [['a', 'b', 'c'], ['d', 'e']]

print(flatten(L))
# Prints ['a', 'b', 'c', 'd', 'e']
```

Jump Table Using a Lambda

The jump table is a list or dictionary of functions to be called on demand. Here’s how a lambda function is used to implement a jump table.

```
# dictionary of functions

exponent = {'square':lambda x: x ** 2,

            'cube':lambda x: x ** 3}


print(exponent['square'](3))

# Prints 9


print(exponent['cube'](3))
# Prints 27

# list of functions

exponent = [lambda x: x ** 2,

            lambda x: x ** 3]


print(exponent[0](3))

# Prints 9


print(exponent[1](3))
# Prints 27
```

Lambda Key Functions

In Python, key functions are higher-order functions that take another function (which can be a lambda function) as a key argument. This function directly changes the behavior of the key function itself. Here are some key functions:

- **List method: [sort()]**
- **Built-in functions: [sorted()], [min()], [max()]**
- **In the Heap queue algorithm module heapq: nlargest() and nsmallest()**

In the following example, a lambda is assigned to the key argument so that the list of students is sorted by their age rather than by name.

```
# Sort the list of taples by the age of students
L = [('Sam', 35),
    ('Max', 25),
    ('Bob', 30)]
x = sorted(L, key=lambda student: student[1])
print(x)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```