# Python: Lambda Functions

> Note: Lines beginning with ">>>" and "..." indicate input to Python (these are the default prompts of the interactive interpreter). Everything else is output from Python.

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda". This is not exactly the same as lambda in functional programming languages, but it is a very powerful concept that's well integrated into Python and is often used in conjunction with typical functional concepts like `filter()`, `map()` and `reduce()`.

This piece of code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
>>> def f (x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>>
>>> print g(8)
64
```

As you can see, f() and g() 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 you can put a lambda definition anywhere a function is expected, and you don't have to assign it to a variable at all.

The following code fragments demonstrate the use of lambda functions. Note that you should have Python 2.2 or newer, in order to have support for nested scopes (in older versions you have to pass "n" through a default argument to make this example work).

```
>>> def make_incrementor (n): return lambda x: x + n
>>>
>>> f = make_incrementor(2)
>>> g = make_incrementor(6)
>>>
>>> print f(42), g(42)
44 48
>>>
>>> print make_incrementor(22)(33)
55
```

The above code defines a function "`make_inrementor`" that creates an anonymous function on the fly and returns it. The returned function increments its argument by the value that was specified when it was created.

You can now create multiple different incrementor functions and assign them to variables, then use them independent from each other. As the last statement demonstrates, you don't even have to assign the function anywhere -- you can just use it instantly and forget it when it's not needed anymore.

---

The following takes this a step further.

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>>
>>> print filter(lambda x: x % 3 == 0, foo)
[18, 9, 24, 12, 27]
>>>
>>> print map(lambda x: x * 2 + 10, foo)
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>>
>>> print reduce(lambda x, y: x + y, foo)
139
```

First we define a simple list of integer values, then we use the standard functions `filter()`, `map()` and `reduce()` to do various things with that list. All of the three functions expect two arguments: A function and a list.

Of course, we could define a separate function somewhere else and then use that function's name as an argument to `filter()` etc., and in fact that's probably a good idea if we're going to use that function several times, or if the function is too complex for writing in a single line. However, if we need it only once and it's quite simple (i.e. it contains just one expression, like in the above examples), it's more convenient to use a lambda construct to generate a (temporary) anonymous function and pass it to `filter()` immediately. This creates very compact, yet readable code.

In the first example, `filter()` calls our lambda function for each element of the list, and returns a new list that contains only those elements for which the function returned "True". In this case, we get a list of all elements that are multiples of 3. The expression `x % 3 == 0` computes the remainder of x divided by 3 and compares the result with 0 (which is true if x is evenly divisible by 3).

In the second example, `map()` is used to convert our list. The given function is called for every element in the original list, and a new list is created which contains the return values from our lambda function. In this case, it computes 2 * x + 10 for every element.

Finally, `reduce()` is somewhat special. The "worker function" for this one must accept two arguments (we've called them x and y here), not just one. The function is called with the first two elements from the list, then with the result of that call and the third element, and so on, until all of the list elements have been handled. This means that our function is called n-1 times if the list contains n elements. The return value of the last call is the result of the `reduce()` construct. In the above example, it simply adds the arguments, so we get the sum of all elements. (Note: since Python 2.3 there's a built-in function `sum()`that does the same thing more efficiently.)

---

The following example is one way to compute prime numbers in Python (not the most efficient one, though):

```
>>> nums = range(2, 50)
>>> for i in range(2, 8):
...     nums = filter(lambda x: x == i or x % i, nums)
...
>>> print nums
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

How does it work? First, we put all numbers from 2 to 49 into a list called "nums". Then we have a "for" loop that iterates over all possible divisors, i.e. the value of "i" goes from 2 to 7. Naturally, all numbers that are multiples of those divisors cannot be prime numbers, so we use a filter function to remove them from the list. (This algorithm is called "the sieve of Eratosthenes".)

In the above case, the filter function simply says: "Leave the element in the list if it is equal to i, or if it leaves a non-zero remainder when divided by i. Otherwise remove it from the list." After the filtering loop finishes, only prime numbers are left, of course. I am not aware of a language in which you can do the same thing with built-in features as compact and as readable as in Python (except for functional programming languages).

Note: The range function simply returns a list containing the numbers from x to y-1. For example, `range(5, 10)` returns the list `[5, 6, 7, 8, 9]`.

---

In the following example, a sentence is split up into a list of words, then a list is created that contains the length of each word.

```
>>> sentence = 'It is raining cats and dogs'
>>> words = sentence.split()
>>> print words
['It', 'is', 'raining', 'cats', 'and', 'dogs']
>>>
>>> lengths = map(lambda word: len(word), words)
>>> print lengths
[2, 2, 7, 4, 3, 4]
```

I think it doesn't need any further explanation, the code is practically self-documenting.

Of course, it could all be written in one single statement. Admittedly, this is somewhat less readable (not much, though).

```
>>> print map(lambda w: len(w), 'It is raining cats and dogs'.split())
[2, 2, 7, 4, 3, 4]
```

---

Here's an example from the UNIX scripting world: We want to find all mount points in our file system. To do that, we execute the external "mount" command and parse the output.

```
>>> import commands
>>>
>>> mount = commands.getoutput('mount -v')
>>> lines = mount.splitlines()
>>> points = map(lambda line: line.split()[2], lines)
>>>
>>> print points
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```

The `getoutput` function from the `commands` module (which is part of the Python standard library) runs the given command and returns its output as a single string. Therefore, we split it up into separate lines first. Finally we use "map" with a lambda function that splits each line (on whitespace, which is the default) and returns just the third element of the result, which is the mountpoint.

Again, we could write all of that in one single statement, which increases compactness but reduces readability:

```
print map(lambda x: x.split()[2], commands.getoutput('mount -v').splitlines())
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```

When writing "real-world" scripts, it is recommended to split up complex statements so that it is easier to see what it does. Also, it is easier to make changes.

However, the task of splitting up the output of a command into a list of lines is very common. You need it all the time when parsing the output of external commands. Therefore, it is common practice to include the split operation on the getoutput line, but do the rest separately. This is a good trade-off between compactness and readability:

```
>>> lines = commands.getoutput('mount -v').splitlines()
>>>
>>> points = map(lambda line: line.split()[2], lines)
>>> print points
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```

An even better idea is probably to write a small function for that task, which encapsulates the job of running the command and splitting the output.

On a related note, you can also use so-called list comprehensions to construct lists from other lists. Sometimes this is preferable because of efficiency or readability. The previous example could very well be rewritten using a list comprehension:

```
>>> lines = commands.getoutput('mount -v').splitlines()
>>>
>>> points = [line.split()[2] for line in lines]
>>> print points
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```

In many cases, you can use list comprehensions instead of map() or filter(). It depends on the situation which one should be preferred.

Note: The `commands` module is deprecated in newer versions of Python (though it still works in all 2.x versions). Instead, the `subprocess` module should be used which is available since Python 2.4. The `commands.getoutput()` function can be replaced by `subprocess.check_output()`. Please refer to the documentation for details.

---