# 15.3. Anonymous functions with lambda expressions

To further drive home the idea that we are passing a function object as a parameter to the sorted object, let's see an alternative notation for creating a function, a **lambda expression**. The syntax of a lambda expression is the word "lambda" followed by parameter names, separated by commas but not inside (parentheses), followed by a colon and then an expression. `lambda arguments: expression` yields a function object. This unnamed object behaves just like the function object constructed below.

```
def fname(arguments):
    return expression
```

```
def func(args):
    return ret_val
```

is equivalent to:

```
func = lambda args: ret_val
```

Consider the following code

Save & Run Original - 1 of 1 Show in CodeLens

```
1 def f(x):
2     return x - 1
3
4 print(f)
5 print(type(f))
6 print(f(3))
7
8 print(lambda x: x-2)
9 print(type(lambda x: x-2))
10 print((lambda x: x-2)(6))
11
```

```
<function f>
<class 'function'>
2
<function <lambda>>
<class 'function'>
4
```

Activity: 1 -- ActiveCode (ac15_3_1)

Note the paralells between the two. At line 4, f is bound to a function object. Its printed representation is "<function f>". At line 8, the lambda expression produces a function object. Because it is unnamed (anonymous), its printed representation doesn't include a name for it, "<function <lambda>>". Both are of type 'function'.

A function, whether named or anonymous, can be called by placing parentheses () after it. In this case, because there is one parameter, there is one value in parentheses. This works the same way for the named function and the anonymous function produced by the lambda expression. The lambda expression had to go in parentheses just for the purposes of grouping all its contents together. Without the extra parentheses around it on line 10, the interpreter would group things differently and make a function of x that returns x - 2(6).

Some students find it more natural to work with lambda expressions than to refer to a function by name. Others find the syntax of lambda expressions confusing. It's up to you which version you want to use though you will need to be able to read and understand lambda expressions that are written by others. In all the examples below, both ways of doing it will be illustrated.

Say we want to create a function that takes a string and returns the last character in that string. What might this look like with the functions you've used before?

Save & Run Original - 1 of 1 Show in CodeLens

```
1 def last_char(s):
2     return s[-1]
3
```

Activity: 2 -- ActiveCode (ac15_3_2)

To re-write this using lambda notation, we can do the following:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
1 last_char = (lambda s: s[-1])
2
```

Activity: 3 -- ActiveCode (ac15_3_3)

Note that neither function is actually invoked. Look at the parallels between the two structures. The parameters are defined in both functions with the variable `s`. In the typical function, we have to use the keyword `return` to send back the value. In a lambda function, that is not necessary - whatever is placed after the colon is what will be returned.

**Check Your Understanding**

advfuncs-3-1: If the input to this lambda function is a number, what is returned?

```
(lambda x: -x)
```

- A. A string with a - in front of the number.
- B. A number of the opposite sign (positive number becomes negative, negative becomes positive).
- C. Nothing is returned because there is no return statement.

Check me | Compare me

✔ Correct!

Activity: 4 -- Multiple Choice (question15_3_1)

You have attempted 5 of 4 activities on this page

15.2. Keyword Parameters">
eyword Parameters">
15.4. Programming With Style">

✔ Completed. Well Done!



| Back to top