

# Chapter 17

## Curried Functions



### 17.1 Introduction

Currying is a technique which allows new functions to be created from existing functions by *binding* one or more parameters to a specific value. It is a major source of reuse of functions in Python which means that functionality can be written once, in one place and then reused in multiple other situations.

The name currying may seem obscure, but the technique is named after Haskell Curry (for whom the Haskell programming language is also named).

This chapter introduces the core ideas behind currying and explores how currying can be implemented in Python. The chapter also introduces the concept of closures and how they affect curried functions.

### 17.2 Currying Concepts

At an abstract level, consider having a function that takes two parameters. These two parameters  $x$  and  $y$  are used within the function body with the multiply operator in the form  $x * y$ . For example, we might have:

```
operation(x, y): return x * y
```

This function `operation()` might then be used as follows

```
total = operation(2, 5)
```

Which would result in 5 being multiplied by 2 to give 10. Or it could be used:

```
total = operation(10, 5)
```

Which would result in 5 being multiplied by 10 to give 50.

If we needed to double a number, we could thus reuse the `operation()` function many times, for example:

```
operation(2, 5)
operation(2, 10)
operation(2, 6)
operation(2, 151)
```

All of the above would double the second number. However, we have had to remember to provide the 2 so that the number can be doubled. However, the number 2 has not changed between any of the invocations of `operation()` function. What if we fixed the first parameter to always be 2, thus would mean that we could create a new function that apparently only takes one parameter (the number to double). For example, let us say we could write something like:

```
double = operation(2, *)
```

Such that we could now write:

```
double(5)
double(151)
```

In essence `double()` is an alias for `operation()`, but an alias that provides the value 2 for the first parameter and leaves the second parameter to be filled in by the future invocation of the `double` function.

## 17.3 Python and Curried Functions

A curried function in Python is a function where one or more of its parameters have been *applied or bound* to a value, resulting in the creation of a new function with one fewer parameters than the original. For example, let us create a function that multiplies two numbers together:

```
def multiply(a, b):
    return a * b
```

This is a general function that does exactly what it says; it multiplies any two numbers together. These numbers could be any two integers or floating-point numbers, etc.

We can thus invoke it in the normal manner:

```
print(multiply(2, 5))
```

The result of executing this statement is:

```
10
```

We could now define a new method that takes a function and a number and returns a new (anonymous) function that takes one *new* parameter and calls the function passed in with the number passed in and the new parameter:

```
def multby(func, num):
    return lambda y: func(num, y)
```

Look carefully at this function; it has used or *bound* the number passed into the `multby` function to the invocation of the function passed in, but it has also defined a new variable ‘y’ that will have to be provided when this new anonymous function is invoked. It then returns a reference to the anonymous function as the result of `multby`.

The `multby` function can now be used to bind the first parameter of the multiply function to anything we want. For example, we could bind it to 2 so that it will always double the second parameter and store the resulting function reference into a property `double`:

```
double = multby(multiply, 2)
```

We could also bind the value 3 to the first parameter of `multiply` to make a function that will triple any value:

```
triple = multby(multiply, 3)
```

Which means we can now write:

```
print(double(5))
print(triple(5))
```

which produces the output

```
10
15
```

You are not limited to just binding one parameter; you can bind any number of parameters in this way.

Curried functions are therefore very useful for creating new functions from existing functions.

## 17.4 Closures

One question that might well be on your mind now is what happens when a function references some data that is in scope where it is defined but is no longer available when it is evaluated? This question is answered by the implementation of a concept known as *closure*.

Within Computer Science (and programming languages in particular) a closure (or a lexical closure or function closure) is a function (or more strictly a reference to a function) together with a referencing environment. This referencing environment

records the context within which the function was originally defined and if necessary, a reference to each of the non-local variables used by that function. These non-local or free variables allow the function body to reference variables that are external to the function, but which are utilized by that function. This referencing environment is one of the distinguishing features between a functional language and a language that supports function pointers (such as C).

The general concept of a lexical closure was first developed during the 1960s but was first fully implemented in the language Scheme in the 1970s. It has since been used within many functional programming languages including LISP and Scala.

At the conceptual level, a closure allows a function to reference a variable available in the scope where the function was originally defined, but not available by default in the scope where it is executed.

For example, in the following simple programme, the variable `more` is defined outside the body of the function named `increase`. This is permissible as the variable is a *global* variable. Thus, the variable `more` is *within scope* at the point of definition.

```
more = 100

def increase(num):
    return num + more

print(increase(10))
more = 50
print(increase(10))
```

Within our program we then invoke the `increase` function by passing in the value 10. This is done twice with the variable `more` being reset to 50 between the two. The output from this program is shown below:

```
110
60
```

Note that it is the *current* value of `more` that is being used when the function executes and not the value of `more` present at the point that the function was defined. Hence the output is 110 and 60 that is  $100 + 10$  and then  $50 + 10$ .

This might seem obvious as the variable `more` is still in scope within the same function as the invocations of the function referenced by `increase`.

However, consider the following example:

```
def increment(num):
    return num + 1

def reset_function():
    global increment
    addition = 50
    increment = lambda num: num + addition

print(increment(5))
reset_function()
print(increment(5))
```

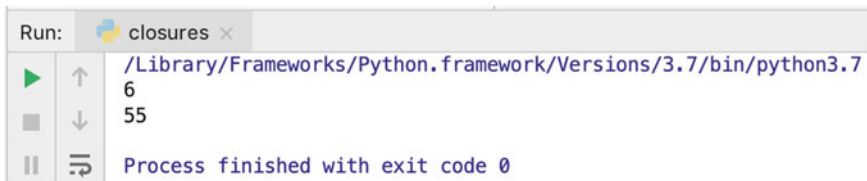
In the above listing the function `increment` initially adds 1 to whatever value has been passed to it. Then in the program this function is called with the value 5 and the result returned by the function is printed. This will be the value 6.

However, after this a second function, `reset_function()` is invoked. This function has a variable that is *local* to the function. That is, *normally* it would only be available within the function `reset_function`. This variable is called `addition` and has the value 50.

The variable `addition` is, however, used within the function body of a new anonymous function definition. This function takes a number and adds the value of `addition` to that number and returns this as the result of the function. This new function is then assigned to the name `increment`. Note that to ensure we reference the *global* name `increment` we must use the keyword `global` (otherwise we will create a local variable that just happens to have the same name as the function).

Now, when the second invocation of `increment` occurs, the `reset_function()` method has terminated and *normally* the variable `addition` would no longer even be in existence. However, when this program runs the value 55 is printed out from the second invocation of `increment`. That is the function being referenced by the name `increment`, when it is called the second time, is the one defined within `reset_function()` and which uses the variable `addition`.

The actual output is shown below:



```
Run: closures x
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3.7
6
55
Process finished with exit code 0
```

So, what has happened here? It should be noted that the value 50 was not copied into the second function body. Rather it is a concrete example of the use of a reference environment with the closure concept. Python ensures that the variable `addition` is available to the function, even if the invocation of the function is somewhere different to where it was defined by binding any free variables (those defined outside the scope of the function) and storing them so that they can be accessed by the function's context (in effect moving the variable from being a *local* variable to one which is available to the function anywhere; but only to the function).

## 17.5 Online Resources

Further information on currying see:

- <https://en.wikipedia.org/wiki/Currying> Wikipedia page on currying.
- <https://wiki.haskell.org/Currying> A page introducing currying (based on the Haskell language but still a useful reference).
- [https://www.python-course.eu/currying\\_in\\_python.php](https://www.python-course.eu/currying_in_python.php) A tutorial on currying in Python.

## 17.6 Exercises

This exercise is about creating a set of functions to perform currency conversions based on specified rates using currying to create those functions.

Write a function that will curry another function and a parameter in a similar manner to `multby` in this chapter—call this function `curry()`.

Now define a function that can be used to convert an amount into another amount based on a rate. The definition of this conversion function is very straight forward and just involves multiplying the number by the rate.

Now create a set of functions that can be used to convert a value in one currency into another currency based on a specific rate. We do not want to have to remember the rate, only the name of the function. For example:

```
dollars_to_sterling = curry(convert, 0.77)
print(dollars_to_sterling(5))

euro_to_sterling = curry(convert, 0.88)
print(euro_to_sterling(15))

sterling_to_dollars = curry(convert, 1.3)
print(sterling_to_dollars(7))

sterling_to_euro = curry(convert, 1.14)
print(sterling_to_euro(9))
```

If the above code is run the output would be:

```
3.85
13.2
9.1
10.26
```