Summation with Currying

In this lesson, we will rewrite our sum function using the currying method.

We'll cover the following

So far, we have been using specific summation functions along with a `sum` function to perform summation operations. Is it possible to remove the need for the summation functions and use the `sum` function directly?

Currying will help us in doing just that. But for this, we will still need helper functions. For a recap, the helper functions `cube`, `factorial`, and `id` are defined below.

```
def id(x: Int): Int = x

def cube(x: Int): Int = x*x*x

def factorial(x: Int): Int = if (x==0) 1 else x * factorial(x-1)
```

Let's now look at how we should define our `sum` function using currying.

Breaking it Down

Firstly, we need a general function that can be used to sum the numbers between two integers. This means that the function should take two integers as parameters. Something similar to this:

```
sum(Int: integer1, Int: integer2)
```

Next, the function should be able to sum the integers in any form. In other words, we should be able to specify to the function that *'compute the sum of the squares of*

the given integers' or *'compute the sum of the halves of the given integers.'* Any

operation can be applied to the integers. This means that along with the list of parameters for the two integers, our function should also take another function as a parameter, a helper function, that would be applied on the integer parameters. Something similar to this:

```
sum(function)(Int: integer1, Int: integer2)
```

Finally, the helper function should be such that it takes one integer, performs some operation on the integer and returns the modified integer as a result.

```
f: Int => Int
```

Implementation

By breaking down the problem, we were able to figure out that our `sum` function should have two parameter lists. The first list will contain a single argument and the second list will contain two arguments.

```
def sum(f: Int => Int)(a: Int, b: Int): Int ={
  if(a>b) 0
  else f(a) + sum(f)(a+1,b)
}
```

Example: Sum of cubes

Let's try our `sum` function. We want to calculate the sum of the cube of integers from **1** to **5**.

This code requires the following environment variables to execute:

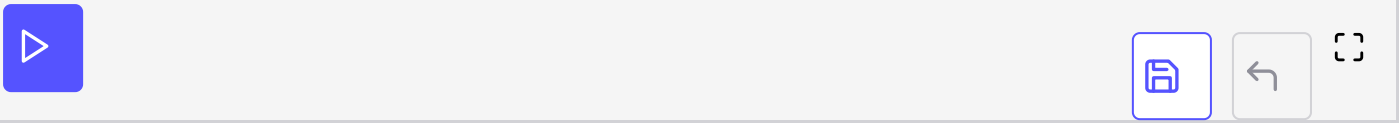
LANG	C.UTF-8

```
def cube(x: Int): Int = x*x*x

def sum(f: Int => Int)(a: Int, b: Int): Int ={
  if(a>b) 0
  else f(a) + sum(f)(a+1,b)
}

// Driver Code
val result = sum(cube)(1,5)
```

```
println(result)
```



`sum(cube)` is now equivalent to `sumOfCube()`, the summation function we defined previously to sum the cubes of integers.

`sum(cube)` is executed first and then this basically returns another function which is applied to the second parameter list `(1,5)`. Just as we did in the previous lesson.

> Association of functions is towards the left. Hence, `sub(cube)(1,5) == (sum(cube))(1,5)`

Why currying?

You might have noticed how the new curried `sum` function isn't all too different from our very first `sum` function which was taking a single list of parameters with three arguments. So, what is the point of currying?

Our curried `sum` function is much more flexible than the non-curried version and gives us a more generic function which we can use in multiple different ways. For instance, we can simply pass the cube function to `sum` without passing the second parameter list `(1,5)`. `sum(cube)` is valid on its own as it is a valid function call and returns another function. The second parameter of two integers can now be passed whenever required later on in the program.

Let's wrap up our discussion in the next lesson by taking a closer look at how multiple parameter lists expand.