**What are higher order functions?**

Higher order functions are functions that have other functions as parameters or if the function returns another function

Basically functions that use other functions inside it are known as higher order functions.

Scala does support higher order functions, here are some simple example of the two types of higher order functions. One which uses a function as a parameter and the other that returns a function.

**Example with function as a parameter:**

We can start out by creating the function first which can take in a parameter. In this case, lets make a simple higher order function that can do some kind of math depending on what they pass.

Ex:

```
def arith(x: Int, y: Int, f(Int, Int) => Int): Int = f(x,y);
```

So the function above passes some arbitrary integers to x and y. Then, you can pass a function that takes in two integers. Those two integers will turn into another integer depending on what we choose to do to those values.

Let's try to do some multiplication of two integers using our arith function we created above and print that result:

Start by creating a main function:

```
def main(args: Array[String]) {
        val result = arith(10, 2, (x,y) => x*y)
        println(result) //returns 20
}
```

What we did here was used two integers for the first two values and then for our function, we decided that we would multiple those integers. So when we print the result, we get 10 times 2 which is 20. If we want to do something else with those integers, we can simply change the * symbol between x and y to any operator like "+" or "-".

Now we have learned how to use higher order function when we want to use function as parameters.

An example of a popular higher order function in Scala is the map function. Here's how it works: Map is a function used for list that can do something to each element in the list. Let's try to illustrate this idea with an example using map.

First let's initialize a list we can use

```
val mylist = List(1,2,3,4,5,6)
```

Then, lets use the map function in the main function to do something in this list.

```
def main(args: Array[String]) {
        println(mylist.map(x => x*2) //returns List[2,4,6,8,10,12]
        println(mylist.map(x => x + "foo") // List(1foo,2foo,3foo,4foo,5foo,6foo)
}
```

The first print statement multiplies every element in the list by 2. And the second print statement appends foo to every element in the list.

We are using map which "changes" a list by doing something to it. It is important to keep in mind that this doesn't actually change the original list, "mylist", but rather creates a new list and print

it. Since, map is a function and we put another function with it, this is a higher order function. This is one of the many example of higher order methods.

Python higher order functions:
https://docs.python.org/3/library/functools.html
https://www.geeksforgeeks.org/higher-order-functions-in-python/
Properties of higher-order functions

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()

print(shout('Hello'))

# Assigning function to a variable
yell = shout

print(yell('Hello'))
```

For example:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

Similarities between python higher order functions and scala higher order functions:
They both are functions that act or return other functions, but it's easier to visualize it with a coding language that most people are more familiar with. (compare and contrast examples of scala higher order functions vs. python higher order functions)

It is much easier to understand higher-order functions by showing the same implementation for filtering out even numbers in Python and Scala.

Python:

```python
def isEven(x):
    return x % 2 == 0

def main(args):

    collection = [1,2,3,4,5]
    evenNums = list(filter(isEven, collection))
    print(evenNums)
```

Scala:

```scala
object FilterEven {

    def isEven(x: Int): Int = {
        x % 2 == 0
    }

    def main(args: Array[String]) {

        val collection = List[1,2,3,4,5]
        val evenNums = collection.filter(isEven)
        println(evenNums)
    }
}
```

With both of these implementations, the filter function takes in an input predicate that applies some function to all of the elements of the collection and returns a new collection with all of the elements satisfying the predicate function. These are defined as higher-order as they take in a function as a parameter.

The only major difference between these 2 implementations is that the filter for Python takes the predicate input function as well as the collection that the predicate will be applied to. For Scala collections already have a filter method so the filter function can be applied directly to the collection with the predicate function that applies to the elements of the collection.

An example that may show some more minor differences is the reduce function. As with the previous example collections in Scala have reduce functions as a method to be applied whereas

with Python, the collection must be passed as a parameter to the higher-order reduce method. The reduce method is a higher-order function that takes all the elements in a collection (Array, List, etc) and combines them using a binary operation to produce a single value. It is necessary to make sure that operations are commutative and associative.

Python:

```
def main(args):

    from functools import reduce

    collection = [1,3,5,2,4]
    totalSum = reduce(lambda x,y: x + y, collection)
    print(totalSum)
```

With the python implementation it may be harder to understand exactly what it is doing but the higher-order part comes in with the “lambda x,y: x+y” translated to “for each (x,y) pair within the collection, compute the sum x+y and store that value to be the x for the iteration.” With both Python and Scala the initial x value is 0 and y is the first item in the collection but you may pass another parameter to the reduce function called the initializer which just sets the starting value of x rather than it being 0.

Scala:

```
object SumNumbers {

    def main(args: Array[String]) {

        val collection = List[1,3,5,2,4]
        val totalSum = collection.reduce((x, y) => x + y)
        println(totalSum)
    }
}
```

As we can see the Scala implementation is very similar with just minor syntax changes as well as there is no option to change the initial x value from 0. With both of these implementations, the steps taken would be as follows:

```
[1,3,5,2,4].reduceLeft((x, y) => x + y)    // initialize var acc
(((1 + 3) + 5) + 2) + 4    // take first value, acc = 1
((4 + 5) + 2) + 4    // acc = 1 + 3 = 5
(9 + 2) + 4  // acc = 4 + 5 = 9
11 + 4    // acc = 9 + 2 = 11
15    // acc = 11 + 4 = 15 returned upon end of collection
```

As shown, the implementation for reduce defaults to a “reduceLeft” method, starting at the smallest indexed element working to the highest indexed element. It will then add parenthesis to do the computation for each x,y pair until a result has been reached. A key difference with the reduce function is that in Python reduce will always do reduceLeft and there is no built-in function to perform reduceRight. With Scala, reduce will automatically perform reduceLeft but you can specify to perform reduceLeft or reduceRight which would perform the same arithmetic but in reverse order.

For the most part, Scala and Python are very similar in terms of higher-order functions and being able to pass functions as parameters or return a function from a function. The main differences lie in the syntax of how they are used. Intuitively I see scala as having a better syntax for higher-order functions that aren’t predefined for example “x => 2x” which will take each element in the collection and multiply it by 2 vs a predefined function being “def timesTwo(x: Int ): Int = { return 2x }”. With Python it may be more confusing to understand “lambda x,y: x+y” but both syntaxes do the exact same thing.

Sources:


https://www.geeksforgeeks.org/higher-order-functions-in-scala/

https://ongchinhwee.me/functional-programming-hof-datapipeline/

https://www.geeksforgeeks.org/scala-reduce-function/

https://realpython.com/python-reduce-function/