Spark in Scala

Spark Functions, Anonymous and Otherwise

Simple operations in Scala

Try typing the following lines at the spark-shell:

```
1 + 1                       // Should be Int = 2
1.0 + 1                     // Should be Double = 2.0
"This is a string"          // String = This is a string
```

Allocate a mutable variable using `var`

```
var x = 2 + 2
println(x)
x = 3 + 3
println(x)
```

However, an immutable, which is created using val, can't be changed once it is created.

```
val x = 2 + 2
println(x)
x = 3 + 3
println(x)
```

You should see

```
<console>:21: error: reassignment to val
       x = 3 + 3
         ^
```

Function Definitions In Spark

In this little exercise, we'll define a simple function in the most straightforward, obvious way and then show how Scala's ability to infer context and types allows us to be more succinct.

Let's start by defining a simple add function and testing it.

```
def add(x:Int, y:Int):Int = {     // Takes two Int arguments and
                                  // returns an Int
  return x + y
}
println(add(42,13))
```

You should see 55 as your result.

Now let's shorten things a bit. Let's use implicit typing.

```
// Implicit typing and return
def add(x:Int, y:Int) = {        //result type is inferred
  x + y                          //"return" keyword is optional
}
```

Further, curly braces are optional on single line blocks, so our function now just can be:

```
def add(x:Int, y:Int) = x + y
```

Not only is it sorter, but it's more readable too.

Anonymous Functions or Function Literals

First, create a named function that simply prepends "Hello " to a name. Then apply that greeting to a list of names using the `map` function.

```
scala> def greeting(x: String) = "Hello " + x
greeting: (x: String)String

scala> val names = List("Joe", "Mary", "Barbara")
names: List[String] = List(Joe, Mary, Barbara)

scala> names.map(greeting)
res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Because we can infer the types, we can use the underscore shortcut for this function definition.

```
scala> names.map("Hello " + _)
res7: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Now let's get rid of the name greeting by using an anonymous function.

```
scala> names.map((x: String) => "Hello " + x)
res3: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Define maximize to compute the maximum of two integers using a function literal instead of def. Function literals are also referred to as anonymous or lambda functions

```
scala> val maximize = (a: Int, b: Int) => if (a > b) a else b
maximize: (Int, Int) => Int = <function2>

scala> maximize(5, 3)
res4: Int = 5
```

Define doubler as an immutable variable whose value is a function. Note we aren't using def for our function definition.

```
scala> val doubler = (x: Int) => x * 2  // This assigns a function literal to doubler
doubler: Int => Int = <function1>

scala> doubler(4)
res5: Int = 8
```

OK, now use the _ placeholder for the doubler function argument. Read this as "doubler is defined as a

function literal that takes an integer and returns and integer; the function definition value multiplies its argument times 2.

```
val doubler: (Int) => Int = _ * 2
doubler(4)
```

Doubler's value is a function literal. We can now pass this value to other functions that take a function as an argument

Many ways exist to define functions and methods. Here are several.

```
val even = (i: Int) => { i % 2 == 0 } // explicit long form
val even: (Int) => Boolean = i => { i % 2 == 0 }
val even: Int => Boolean = i => ( i % 2 == 0 }
val even: Int => Boolean = i => i % 2 == 0
val even: Int => Boolean = _ % 2 == 0 // _ means first argument

// implicit result approach
val add = (x: Int, y: Int) => { x + y }
val add = (x: Int, y: Int) => x + y

// explicit result approach
val add: (Int, Int) => Int = (x,y) => { x + y }
val add: (Int, Int) => Int = (x,y) => x + y
```

We can also use _ as the argument placeholder for our even number tester.

```
scala> val nums = Array(1, 2, 3, 4, 5)
nums: Array[Int] = Array(1, 2, 3, 4, 5)

scala> nums.map(_ % 2 == 0)
res8: Array[Boolean] = Array(false, true, false, true, false)
```

This may be harder to understand; it's a reducing function similar to that used in MapReduce. This anonymous function adds its two arguments together. The net result is that it sums all the elements of the immutable `nums`.

```
scala> nums.reduceLeft(_ + _)  // first plus second argument
res9: Int = 15
```

These literal defintions are incredibly valuable in Spark because many of the Spark operations take functional arguments. Most of these will never be assigned to a variable. For example the following finds list entries that have an "f" in them using an anonymous function/function literal

```
scala> val mylist = List("foo", "bar", "fight")
mylist: List[String] = List(foo, bar, fight)

scala> mylist.filter(_.contains("f"))
res10: List[String] = List(foo, fight)
```

We could have written that as follows, but it's longer and less clear

```
scala> val mylist = List("foo", "bar", "fight")
mylist: List[String] = List(foo, bar, fight)

scala> val ffilter = (s: String) => s.contains("f")      // define a named function to search
ffilter: String => Boolean = <function1>

scala> mylist.filter(ffilter)                            // will generate the same result as p
res11: List[String] = List(foo, fight)
```

First full example: Textsearch

Load error messages from a log into memory, then interactively search for various patterns.

The file `log.txt` has the following 5 lines:

```
ERROR       php: dying for unknown reasons
WARN        dave, are you angry at me?
ERROR       did mysql just barf?
WARN        xylons approaching
ERROR       mysql cluster: replace with spark cluster
```

Our objective is to count all the error messages (not warnings) that have reference *mysql* or *php*.

```
val lines = sc.textFile("hdfs:///data/logs/log.txt")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Now Wordcount

The periods in this dataflow pipeline are at the ends of lines so that we can execute this code in the interactive spark-shell without modification.

Type this into spark-shell and see how it works for you.

```
val textFile = sc.textFile("hdfs:///data/shakespeare/input")
val counts = textFile.flatMap(line => line.split(" ")).
                map(word => (word, 1)).
                reduceByKey(_ + _)
counts.saveAsTextFile("hdfs:///tmp/shakespeare-wc-scala")
```

Estimate Pi in Spark

This program generates 100,000 x and y variables between 0 and 1. It then counts the ratio of those that fall within a unit circle over the number of total samples; that value should approximate pi/4.

Try various values of NUM_SAMPLES to see how the computed value and runtimes vary.

```
val NUM_SAMPLES = 100000
val count = sc.parallelize(1 to NUM_SAMPLES).map{i =>
  val x = Math.random()
  val y = Math.random()
  if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)
```