Scala + Spark

Introduction to Scala + Spark for Programmers, based on the course from Sundog Education: https://www.udemy.com/course/apache-spark-with-scala-hands-on-with-big-data

- Scala
- Intro to Spark / RDDs
- DataFrames and Advanced Techniques

Scala

I'll start with a disclaimer:

- These are notes written by an experienced Java dev, thus some level of basic programming knowledge is required
- I cannot possibly cover every unique feature of Scala, only what's most important to me
- Awesome and free tutorials are available on the Scala docs or RocktheJVM

Also I'll be focusing on Scala 2, rather than the most recent release of Scala 3. There are big differences in the syntax but the functionality remains *mostly* the same. My reasoning for this is the Government dataset I use is frozen on Scala 2.12.15. More tech-fluent companies may be using Scala 3, but those of us in healthcare and government should get comfortable with v2.

Overview of Types

Scala has types, similar to most other JVM based languages. However, one thing that is specific to Scala is starting variable declaration with *val* or *var*.

```
val someValue: Int = 42 // Immutatble
var someBool = false // mutable, type inferred
```

val is similar to final in Java, the variable is now a constant, while var can be reassigned to the same type. Type is usually optional as it can be inferred, but it's often recommended to use the type for ease of reading.

It is always best practice to use immutable objects. When modifying an object, it's best to return a new object:

```
val word = "Hello World"
val reversedWord = word.reverse
```

Other basic types are pretty much the same as in Java. You got your String, Int, Float, Boolean, etc. However, there are several ways to represent nothingness in Scala:

- Nil - Used for representing empty lists, or collections of zero length. For sets you can use Set.empty
- None - One of two subclasses of the Optional type, the other being Some. Very well supported by the Scala collections
- Unit - Equivalent to Java's void which is used for functions that don't have a return type

- Nothing - A trait. It is a subtype of all other types, but supertype of nothing. It is like the leaf of a tree. No instances of Nothing
- Null - A trait. Not recommended to be used
- null - An instance of the trait, similarly used as the Java Null. Not recommended to be used

The best practice when coding that could return null, is to use the Optional type.

```
def methodWhichCouldReturnNull(): String = "Hello, Scala"
val anOption = Option(methodWhichCouldReturnNull())
```

Another important keyword is *lazy* which when applied to a variable makes it so it is not evaluated until the value is used.

Collections

A Scala Tuple is an immutable value that contains a fixed number of elements, which each has its own type.

Lists and Arrays must be all of the same type. A Scala List is an immutable data structure, ex. val sample: List[String] = List("type1", ...). While an Array is mutable, ex. val sample: Array[String]] ["type1", ...]. Lists are usually preferred as they take less memory. The Java equivalent is LinkedList vs ArrayList.

Another way to construct a list is to use the "element1 :: element2 :: ... :: Nil" operator in Scala:

```
// Lists
val oneList = 1 :: Nil // A list with only 1
val list2 = List(1,2,3)
val firstElemenet = list2.head
val lastElement = list2.tail
val appenededList = 0 :: list2 // List(0, 1, 2, 3)
val extendedList = 0 +: list2 +: 4 // List(0,1,2,3,4)

// Sequences - similar to list but you can access an element at any given index (starting at 0)
val aSequence: Seq[Int] = Seq(1,2,3)
val accessedElement = aSequence(1) // 2

// Vectors - very fast implementation of sequences for large datasets
val aVector = Vector(1,2,3,4)

// Sets - collections with no duplicates
val aSet = Set(1,1,2,3,4) // Set(1,2,3,4)
// useful set operations
```

```
val setHas5 = aSet.contains(5) // false
val addedSet = aSet + 5 // Set(1,2,3,4,5)
val removedSet = aSet - 3 // Set(1,2,4)

// range - doesn't actually contain every value but iterates like it does!
val aRange = 1 to 1000
val aRangeByTwo = aRange.map(x => 2 * 2).toList // List(2,4,6... 2000)

// tuples - groups of values under the same value
val tuple1 = ("Test1", 25) // inferred type is (String, Int)
// Access using the underscore
println(tuple1._1) // "Test1"
println(tuple2._2) // 25

// maps - two argument typles
val phoneBook: Map[String, Int] = Map(
	("Ari", 1234"),
    "bob" -> 5678 // Same thing
)
```

Symbols

The biggest complaint I have about Scala is the rampant use of symbols without a proper appendix explaining what they do. I found this blogpost to be more informative than the actual docs in this area. I'll break down the most popular symbols:

- _ is widely used as the wildcard and pattern matching unknown symbols. It can also be thought of as the current or default value. We'll see more of this later
- + is for adding a single element
- ++ is for concatenating two collections
- : is for declaring associativity (type)
- :: represents prepending a single element or tuple of elements to a List
- ::: represents prepending a list with another list

Exceptions

Exceptions work much the same as in Java, but the catch statement looks slightly different:

```
// code that could throw an error
try {
	val x: String = null
    x.length
```

```
} catch {
case e: Exception => "some fatal error message"
} finally {
// Execute some code no matter what
}
```

We can also use a Try object, which is a collection with a value if the code succeeded, or an exception if not.

```
def methodWhichCouldThrowException(): String = throw new RuntimeException
val aTry = Try(methodWhichCouldThrowException())
```

Functional Programming

Scala is a functional programming language. While in most languages we think about the flow of instructions in sequence, but in a functional language we want to think in terms of composing the values through expressions, and passing functions as arguments and as results.

For example, observe the if expression below:

```
val ifExpression = if (testValue > 42) "String1" else "String2" // if-expression
```

This is much more readable than other ternary operators in other languages, this allows for cleaner chaining:

```
val ifExpression = if (testValue > 42) "String1"
else if (testValue == 0) "String2"
    else "String3" // chained if-expression
```

In Scala a function can be defined as a single line or multiple line statement. Scala is very permissive with object names; You can define methods that start with ? or !. Methods that start with ! usually denote it communicates with actors Asynchronously (which I don't think I'll be covering as it cannot be used with Spark).

```
def myFunction(x: Int, y: String): String = s"$y is a string and $x is an int"


def myOtherFunction(x: Int): Unit = {
x + " is an int"
}


def factorial(n: Int): Int =
```

```
if (n<=1) 1
    else n + factorial(n-1)
```

The above code also demonstrates the following:

- The return statement is not needed, the last line of the function is automatically returned
- Unit is the equivalent of void in Java, that is to say the return value does not have a meaningful value
- The string with the s"" represents a formatted string, where we can reference variables or expressions with $ or ${}

The key difference in functional programming, is DO NOT USE LOOPS OR ITERATION. We use recursion and functional mapping. If you're writing code with a lot of for loops, you're not thinking functionally.

More on this after a brief overview of objects...

Object-Oriented Concepts

Scala is a Object-Oriented language. It has objects, abstraction, and inheritance just like Java. For example:

```
object MyClass extends App {

class Animal {
    val age: Int = 0
    def eat() = println("Munch Munch")
    }

    val animal = new Animal

    // inheritance
    class Dog(name: String) extends Animal // constructor definition
    val dog = new Dog("Fluffy")

    // Doesn't work, constructor arguements are NOT fields
    dog.name

    class Cat(val name: String) extends Animal
    val cat = new cat("kitty")
    // this works, it was declared
    cat.name
```

```
// subtype polymorphism
val someAnimal: Animal = new Dog("Spot")
someAnimal.eat()

abstract class WalkingAnimal {
def walk(): Unit // does not need to be initialized
}

// Interfaces are abstract types
trait Bird {
private val hasWings = true
    def eat(animal: Animal): Unit
}

// single-class inheritance, multi-trait "mixing"
class Dragon extends Animal with Bird {
override def eat(animal: Animal): Unit = println("I am eating animal!")

    def ?!(thought: String): Unit = println("I just thought " + thought) // perfectly valid method name
}

val aDragon = new Dragon
aDragon.eat(aDog)
aDragon eat aDog // This is infix notiation, only works for methods with 1 argument

aDragon ?! "about pizza"



// singleton object, only one instance exists
object MySingleton {
val someVal = 1234
    def someMethod(): Int = 5678
    def apply(x: Int): Int = x + 1 // the apply method is special
}

// Both of the below are equivalent!
MySingleton.apply(54)
MySingleton(54) // the presence of an apply method allows the singleton to be invoked like a function
```

```
    // Case classes are lightweight data structures with senisble equals, hash code, and serialization implementation
    case class Person(name: String, age: Int)
    // We don't need the new keyword because case class has a companion object with an apply method!
    val bob = new Person("bob", 54) // equivalent to Person.apply("Bob", 54)
}
```

- An object is similar to a static type in Java, we don't have to declare a *new* instance to call it's functions
- All fields are by default public, there is no public keyword. But there is a *private* or *protected* keyword for methods/vars we wish to keep hidden
- A class is similar to that of Java, we need to create a new instance through its constructors to use it
- *extends* can be used to inherit functions from the parent class
  - Note the *extends App* at the top level object makes the class runnable. It add a hidden "def main(args: Array[String])" method, equivalent to the java "public static void main(String[] args)"
- *abstract* classes and interfaces (*traits*) also exist, which do not need an implementation for every function

Generics

I'll briefly mention here that Scala can use Generics the same way that Java can. The T below could be any letter, and represents a type:

```
abstract class MyList[T] {
    def head: T
    def tail: MyList[T]
}
```

I will not go any further into this because in my experience these are seldomly used in data processing, but they have their place.

Applying Functions as Objects

As previously stated, in functional programming we want to think of functions like objects that can be passed as arguments and be returned by other functions.

```
class MyClass extends App {


    val simpleIncrementor = new Function1[Int, Int] { // an object that takes an int and returns an int
        override def apply(arg: Int): Int = arg + 1
    }
```

```
    // Remember singletons with an apply function can be called as a function
simpleIncrementor.apply(54)
    simpleIncrementor(54)

val stringConcatanator = new Function2[String, String, String] { // an object that takes 2 Strings and returns a String
    override def apply(arg1: String, arg2: String): String = arg1 + arg2
    }
    stringConcatanator("Scala", "is great")



    // Syntax Sugars - a shorthand way to do the same thing as above
    val doubler: Function1[Int, Int] = (x: Int) => 2 * x
    // here's another simplification, although the type can be implied
    val doubler: Int => Int = (x: Int) => 2 * x
    doubler(54)



    // higher-order functions: Take function as args and return function as results
    // One-to-one mapping
    val aMappedList: List[Int] = List(1,2,3).map(x => x + 1) // List(2,3,4)
    // One-to-many mapping
val aFlatmappedList = List(1,2,3).flatMap(x => List(x,2*x)) // List(List(1,2), List(2,4), List(3,6))
    // Filtering, using _ instead of x => x
    val aFilteredlist = List(1,2,3,4,5).filter(_ <= 3) // equivalent to x => x <= 3



    // Example - all pairs between lists 1,2,3 and a,b,c
    val allPairs = List(1,2,3).flatMap { number =>
    List('a','b','c').map(letter => s"$number-$letter")
    }
    // alternative with for loops
    val alternativePairs = for {
    number <- List(1,2,3)
        letters <- List('a', 'b', 'c')
    } yield {
    s"$number-$letter"
    }
}
```

The FunctionX (Function1, Function2, Function3.... Function22) objects above are actually built in Scala type, and you can only have up to 22. **ALL SCALA FUNCTIONS ARE INSTANCES OF SOME FUNCTIONX**

Pattern Matching

Let's take a look at the case statement in Scala:

```
val anInteger = 55
val order = anInteger match {
case 1 => "first"
    case 2 => "second"
    case _ => anInteger + "th (default)"
}
```

Pretty standard, but unlike other languages we can use case classes , Lists, or tuples for the case:

```
case class Person(name: String, age:Int)
val bob = Person("bob", 54)

val personGreeting = bob match {
case Person(n, a) => s"Hi my name is $n, I am $age years old"
    case _ => "Something else"
}

// Deconstructing tuples
val aTuple = ("The Beatles", 1964)
val bandDescription = aTuple match {
case (band, year) => s"$band started in $year"
    _ => "I don't know what you're talking about
}

// Decomposing lists
val aList = List(1,2,3)
val listDescription = aList match {
case List(_,2,_) => "List continaing 2 on its second position"
    case _ = "Other"
}
```

Also note that it will always match the patterns in sequence, so the first match will always be taken.

Contextual/Implicit Parameters

When a function parameter starts with *implicit*, Scala will look for a implicit variable that matches the type and will automatically fill them in if the function is used without said parameter.

```
def aMethodWithImplicitArg(implicit arg: int) = arg + 1
implicit val myImplicitInt = 54
println(aMethodWithImplicitArg) // Will use myImplicitInt
```

This concept can also be used for implicit conversions:

```
implicit class MyRichInteger(n: Int) {
  def isEven() = n % 2 == 2
}
// Int does not have a method isEven, but because MyRichInteger is implicit Scala will apply that class to 23
println(23.isEven()) // new MyRichInteger.isEven()
```

This is a cool feature, and useful when we don't have complete control over the codebase. However, it should be used carefully.

Intro to Spark / RDDs

Apache Spark

Spark is a fast and general engine for large-scale data processing. The user writes a Driver Program containing the script that tells spark what to do with your data, and Spark builds a Directed Acyclic Graph (DAG) to optimize workflow. With a massive dataset, you can process concurrently across multiple machines.

Let's take a second to discuss the components of spark:

- **Spark Core** - All functionalities are build on this layer (task scheduling, memory management, fault recovery, and storage systems). Also has the API that defines Resilient Distributed Datasets (RDDs), which we will discuss later. There are 4 modules on this layer:
  - **Spark Streaming** - Live data streaming of data, such as log files. These APIs are similar to RDD.
  - **MLib** - Scalable machine learning library
  - **Spark SQL** - Library for working with structured data. Supports Hive, parquet, json, csv, etc.
  - **GraphX** - API for graphs and graph parallel execution. Clustering, classification, traversal, searching, and path-finding is possible in the graph. We'll come back to this much later on.

Why Scala?

You can use Spark with many languages; primarily *Python, R, Java and Scala*. I like Scala because it's functional, type-safe and JVM-friendly language. Also, since Spark is written in Scala, there is a slight overhead on running scripts in any other language.

Besides a knowledge of programming, a familiarity with SQL with make Spark very easy to learn.

The Resilient Distributed Dataset

The RDD is the backbone of Spark Core. This is not the same as a DataFrame, that's a different level of the API. This is a dataset made up of rows of information, that can be divided (distributed)

on to many computers for parallel processing, and Spark makes sure we can find a way to get the data even if a node goes down during an operation (resilient).

Creation

We don't have to write the code to make sure we can handle al of the, the RDD is within the **SparkContext.** The first thing the Driver Program does is start the SparkContext automatically. And the RDD can come from any structured data stores; a database, file, in-memory object, or anything else.

Transformation

Once you have an RDD you'll probably want to transform it. There are many types of row-wise operations, such as:

- map - map an operation to each row, one-to-one
- flatmap - map an operation to create new rows from each row, one-to-many
- filter
- distinct
- sample
- Set operations - union, intersection, subtract, cartesian

Since Scala is a functional language, functions are objects. Meaning, there are many functions that also take functions. This is helpful in mapping operations to RDDs:

```
def sqaureIt(x: Int) {
	return x*x
}

rdd.map(sqaureIt)
```

Actions

Finally we want the results of our RDD, to do this we can summarize the data with functions such as:

- collect - view all the raw data
- count
- countByValue - how many rows exist for each given unique value
- take - sample the rows
- top - view the top rows
- reduce

> **Important: You can call a transformation on an RDD but until you call an action nothing will be executed! This is part of Spark's lazy evaluation.**

Key Values RDDs

The map function can return key-value pairs by returning a tuple, and the values in the tuple can be other tuples or a complex object. This is not unique to Scala

```
rdd.map(x => (x,1))
```

And there are unique functions that can be used with key-value RDDs:

- reduceByKey() - combine values by key
  - ex. ((x,y) => x + y) to add them
  - think of x like the current row and y as the next row
- groupByKey() - group keys with the same values
- sortByKey() - sort by keys
- keys(), values() - return rdd of keys or values
- SQL style joins are also possible, but in practice Dataframes or the SQL API are usually used for this
- mapValues() / flatmapValues() - apply operation only to the values

Map vs Flatmap

map is a fairly straightforward concept, for each element of an RDD a transformation is applied in a 1:1 manner

```
val lines = sc.textFile("words.txt")
val bigWords = lines.map(x => x.toUpperCase) // covert all words to uppercase
```

Flatmap removes that 1:1 restriction. We can create many new rows for each row:

```
val lines = sc.textFile("words.txt")
val words = lines.map(x => x.split(" ")) // split words on spaces
val wordCount = words.countByValue() // count occurances of each word
```

DataFrames and Advanced Techniques

A Spark DataSet is an extension of the RDD object. It has rows, can run queries, and has a schema (which leads to more efficient storage and optimization). A DataFrame is just a DataSet of Row Objects, and unlike a DataSet the schema is inferred at runtime rather than compile time. This also means DataSets can only be used in compiled languages (NOT Python).

There are some instances where an RDD might be preferred, but for the vast majority of operations **DataSets are king.** They are more efficient, simplify development and allow for interoperability with other libraries.