Introduction to Scala

Spark

* Similar to Map/Reduce, but it was actually designed by database scientists
  + The code is easier to write
  + Much more versatility
  + Kind of like its own language for Hadoop jobs, so these jobs can be done in much fewer lines
* All intermediate data is stored in main memory – not disk

Scala

* Scala’s like Java – it even runs Java code and compiles into Java binary code
* Kind of like a scripting language so you’ll write much less code than Java.
* You can pass functions as parameters
* Anonymous Functions
  + You can use the ‘=>’ operator to make lambda functions
  + x => x+1 is an anonymous function that increments the input by one
* Scala is meant to be scalable – where you can write as little code as possible.

Example Program:

import scala.io.\_

object App {

def main(args: Array[String]): Unit = {

println(“Hello World!”); // ‘;’ is optional

}

}

* Instead of “static” you have objects. Here, this one is called “App”
* Unit means that this function does not return anything
* Array here is an object.

Variables:

* Two Types of variables
  + var = variable, it can change
    - CANT USE UNARY OPERATOR ‘++’
    - You can explicitly define the time by using :
      * var x: int = 5;
  + val = value, it doesn’t change, similar to a constant
* There are no primitive types. It is ENTIRELY OBJECT ORIENTED

Functions:

def max(x: Int; y: Int): Int = {

if (x > y){

x;

}

y;

}

* Scala will actually look for the last thing in the run and return that value.
* So the above code is the same as:

if (x > y)

Return x;

else

Return y;

* Return keyword is optional
* Here we are literally making max = to some function
* If the method does not take any parameters, it does not need the empty () – you can omit them

Functions with many of the same Parameter:

// Very Java-Looking Code

def sum (a: Int\*) : Int = {

var sum = 0;

for (el <- a){

Sum = sum + a;

}

return sum;

}

println(sum(2, 3, 4, 5, 6));

// Very Scala-looking Code

def sum (a: Int\*): Int = {

a.reduce ({(x,y) => x+y})

}

println(sum(2, 3, 4, 5, 6));

* You can specify multiple inputs of the same type using \*
* And the for each loops looks different
* The method reduce takes a function as an argument.
  + In this case the function takes two numbers, and returns their sum
  + Since reduce only takes one function as an argument, the bolded parentheses are optional

Functions with Several Lists of Arguments:

def myFunction(x: Int, a: Int) (y: Int) (z: Int) : Int = {

x+y+z+a // no return needed

}

println(myFunction(2,3)(5)(10));

* You can take multiple sets of arguments
* You just call the function with different sets of parentheses

“Primitive” Types:

* Types:
  + Int
  + Long
  + Double
  + String
  + Boolean
  + Char
* These all behave like the Java primitives (pass by value) – but these are technically objects in Scala
* All these classes inherit from AnyVal, which inherits from Any (the root class)

Lists:

* List = ArrayList in Java + some other features

val fruits = List(“apples”, “oranges”, “pears”)l

fruits.foreach(println(\_));

* No need for “new” to initialize a new object
* .foreach() takes a function as input and does this function for each entry in the list
  + Can also be written as:
    - foreach(x => println(x)) // this is more code
* We can only use the underscore in this situation because the equivalent function only does one thing for each entry in the list
* You can access an entry in the list by using listName(INDEX).
  + E.g. fruits(2) => element at index 2
  + This calls a method in fruits that gives you the element
  + Square brackets are reserved for when we are using “Genetic/Generic” classes
* The contents of a list must all be of the same type
* Lists are immutable; YOU CANNOT MODIFY THEM

:: and ::: Syntax

* The commas can be exchanged for the :: syntax – but you must add Nil to the very end
* ::: help concatenate lists
  + keep in mind that the end of the list may have Nil, so you can only concatenate in the beginning in this situation

Other Methods:

* drop(x: Int) – drops the first x things in the list
* dropRight(x: Int) – drops the first x things in the list from the right
* head() – the entry in the first spot of the list; doesn’t take parameters, so you don’t need the parentheses
* tail() – returns a list of the entries excluding the head
* reverse() – reverses the entries in the list – this actually creates a new list
* sort(FUNCTION) – sorts the list based on the provided function
  + .sort{ (x: Int, y: Int) => (x<y)}
  + Orders list in ascending order
* mkString(delim) – makes a String out of the entries in the list delimited by the delim

Arrays:

var fruits = new Array[String](3)

fruits(0) = “apples”;

fruits(1) = “oranges”;

fruits(2) = “pears”;

fruits.foreach(println);

* You need to use the “new” keyword
* Use () instead of [] to reference an element of the array
* Alternative declaration syntax:

var fruits = Array(“apples”, “oranges”, “pears”);

* You can also do a 2D array:

var fruits = Array.ofDim[String](3,3);

**Map/Filter Keywords:**

* **Map**: Returns a new list by applying a function to each element of the list.

Var fruits = List(“apples”, “oranges”, “pears”)

Fruits.map(x=> x+“!”).foreach(println)

* **Filter:** returns a new list that contains only the elements that pass the condition.
* an array of numbers:

val numbers = (1 to 10 by 2).toList **// Could also use an array**

numbers.map(x=>x+1).filter(x=>x%3==0).foreach(println)

* \_ shorthand

val numbers = (1 to 10).toList

numbers.map(\_+1).filter(\_%3==0).foreach(println)