spark_scala_tutorial

January 6, 2021

Variables

[1]:
```
"hello  ".trim
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://192.168.8.105:4041
SparkContext available as 'sc' (version = 2.2.1, master = local[*], app id =␣
↪local-1573925835332)
SparkSession available as 'spark'
```

[1]:
```
res0: String = hello
```

[2]:
```
"hello".toUpperCase
```

[2]:
```
res1: String = HELLO
```

[3]:
```
"HELLO".toLowerCase
```

[3]:
```
res2: String = hello
```

[4]:
```
"hello".take(2)
```

[4]:
```
res3: String = he
```

[5]:
```
"hello world".length
```

[5]:
```
res4: Int = 11
```

[6]:
```
"hello world".split(" ")
```

[6]:
```
res5: Array[String] = Array(hello, world)
```

[7]:
```
"hello world".replace("o", "0")
```

[7]:
```
res6: String = hell0 w0rld
```

[8]:
```
"h m".replaceAll("(.*?)[\\s]+(.*?)", "$1&$2")
```

[8]:
```
res7: String = h&m
```

[9]:
```
" hello ".toUpperCase.trim.take(2)
```

[9]:
```
res8: String = HE
```

[10]:
```
("hello" + " " + "world").length
```

[10]:
```
res9: Int = 11
```

[11]:
```
"user1".endsWith("1")
```

[11]:
```
res10: Boolean = true
```

[12]:
```
"user1".endsWith("a")
```

[12]:
```
res11: Boolean = false
```

Casting primitive types

[13]:
```
val a = "12"
val b = a.toInt
```

[13]:
```
a: String = 12
b: Int = 12
```

Collections

List

- immutable

[14]:
```
val list = List(1, 2, 3)
```

[14]:
```
list: List[Int] = List(1, 2, 3)
```

[15]:
```
list.length
```

[15]:
```
res12: Int = 3
```

[16]:
```
list(0)
```

[16]:
```
res13: Int = 1
```

[17]:
```
list.head
```

[17]:
```
res14: Int = 1
```

[18]:
```
list.tail
```

[18]:
```
res15: List[Int] = List(2, 3)
```

[19]:
```
val extended = 0 :: list
```

[19]:
```
extended: List[Int] = List(0, 1, 2, 3)
```

[20]:
```
val extended = list :+ 4
```

[20]:
```
extended: List[Int] = List(1, 2, 3, 4)
```

[21]:
```
list.indexOf(1)
```

[21]:
```
res16: Int = 0
```

[22]:
```
list.slice(0, 2)
```

[22]:
```
res17: List[Int] = List(1, 2)
```

[23]:
```
list.sum
```

[23]:
```
res18: Int = 6
```

[24]:
```
list.max
```

[24]:
```
res19: Int = 3
```

[25]:
```
list.mkString(",")
```

[25]:
```
res20: String = 1,2,3
```

[26]:
```
list.sorted
```

[26]:
```
res21: List[Int] = List(1, 2, 3)
```

Mutable List

[27]:
```
import scala.collection.mutable.ListBuffer

val list = new ListBuffer[Int]()
```

[27]:
```
import scala.collection.mutable.ListBuffer
list: scala.collection.mutable.ListBuffer[Int] = ListBuffer()
```

[28]:
```
list.append(1)
```

[29]:
```
list
```

[29]:
```
res23: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1)
```

[30]:
```
list += 2
```

[30]:
```
res24: list.type = ListBuffer(1, 2)
```

[31]:
```
list.appendAll(List(0, 2))
```

[32]:
```
list
```

[32]:
```
res26: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 0, 2)
```

[33]:
```
list.prepend(-1)
```

[34]:
```
list
```

[34]:
```
res28: scala.collection.mutable.ListBuffer[Int] = ListBuffer(-1, 1, 2, 0, 2)
```

[35]:
```
list.distinct
```

[35]:
```
res29: scala.collection.mutable.ListBuffer[Int] = ListBuffer(-1, 1, 2, 0)
```

[36]:
```
list.remove(0)
```

[36]:
```
res30: Int = -1
```

[37]:
```
list
```

[37]:
```
res31: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 0, 2)
```

Map

- immutable

[38]:
```
val map = Map[String, Int]()
```

[38]:
```
map: scala.collection.immutable.Map[String,Int] = Map()
```

[39]:
```
var m = Map[String, Int]("id" -> 1)
```

[39]:
```
m: scala.collection.immutable.Map[String,Int] = Map(id -> 1)
```

[40]:
```
m += ("age" -> 4)
```

[41]:
```
m
```

[41]:
```
res33: scala.collection.immutable.Map[String,Int] = Map(id -> 1, age -> 4)
```

[42]:
```
m.get("id")
```

[42]:
```
res34: Option[Int] = Some(1)
```

[43]:
```
val id = m.getOrElse("id", 0)
```

[43]:
```
id: Int = 1
```

[44]:
```
m -= "age"
```

[45]:
```
m.keys
```

[45]:
```
res36: Iterable[String] = Set(id)
```

Mutable Map

[46]:
```
import scala.collection.mutable

val map = mutable.Map[String, AnyVal]()
```

[46]:
```
import scala.collection.mutable
map: scala.collection.mutable.Map[String,AnyVal] = Map()
```

[47]:
```
map("id") = 1
```

[48]:
```
map += ("age" -> 4)
```

[48]:
```
res38: map.type = Map(age -> 4, id -> 1)
```

[49]:
```
map.put("active", true)
```

[49]:
```
res39: Option[AnyVal] = None
```

[50]:
```
map.getOrElse("active", null)
```

[50]:
```
res40: Any = true
```

[51]:
```
map.isEmpty
```

[51]:
```
res41: Boolean = false
```

[52]:
```
map.keys
```

[52]:
```
res42: Iterable[String] = Set(active, age, id)
```

[53]:
```
map contains "id"
```

[53]:
```
res43: Boolean = true
```

[54]:
```
map.size
```

[54]:
```
res44: Int = 3
```

[55]:
```
map.values
```

[55]:
```
res45: Iterable[AnyVal] = HashMap(true, 4, 1)
```

[56]:
```
map.clear
```

[57]:
```
map
```

[57]:
```
res47: scala.collection.mutable.Map[String,AnyVal] = Map()
```

Lambda Expressions Map Method - Map method is a for kind iterator function that takes variable(s), transform each o parameter in each iteration and return the result as stream list

[58]:
```
val l = List[Int](1, 2, 3, 4, 5)
```

[58]:
```
l: List[Int] = List(1, 2, 3, 4, 5)
```

[59]:
```
l.map(k => k * 2).map(i => i - 1)
```

[59]:
```
res48: List[Int] = List(1, 3, 5, 7, 9)
```

[60]:
```
val m = Map("id" -> 1, "age" -> 12)
```

[60]:
```
m: scala.collection.immutable.Map[String,Int] = Map(id -> 1, age -> 12)
```

[61]:
```
m
```

[61]:
```
res49: scala.collection.immutable.Map[String,Int] = Map(id -> 1, age -> 12)
```

[62]:
```
m.map(i => i._1)
```

[62]:
```
res50: scala.collection.immutable.Iterable[String] = List(id, age)
```

[63]:
```
m.map(i => i._2)
```

[63]:
```
res51: scala.collection.immutable.Iterable[Int] = List(1, 12)
```

Filter Method - Filter method is a for kind iterator function that takes variable(s), apply boolean filter on each parameter and return the response as subset of input list

[64]:
```
val l = List[Int](1, 2, 3, 4, 5)
```

[64]:
```
l: List[Int] = List(1, 2, 3, 4, 5)
```

[65]:
```
l.filter(k => k > 2)
```

[65]:
```
res52: List[Int] = List(3, 4, 5)
```

[66]:
```
val m = Map("id" -> 1, "age" -> 12)
```

[66]:
```
m: scala.collection.immutable.Map[String,Int] = Map(id -> 1, age -> 12)
```

[67]:
```
m.filter(i => i._2 != 1)
```

[67]:
```
res53: scala.collection.immutable.Map[String,Int] = Map(age -> 12)
```

Reduce Method - Reduce method is a for kind iterator function that takes variable(s), apply aggregation operations which defined in method and return the result as variable

[68]:
```
val l = List[Int](1, 2, 3, 4, 5)
```

[68]:
```
l: List[Int] = List(1, 2, 3, 4, 5)
```

[69]:
```
l.reduce((l, r) => l max r)
```

[69]:
```
res54: Int = 5
```

[70]:
```
l.reduce((l, r) => l + r)
```

[70]:
```
res55: Int = 15
```

Chaining - You can combine lambda expressions and create a chain function

[71]:
```
val l = List[Int](1, 2, 3, 4, 5, 6, 7)
```

[71]:
```
l: List[Int] = List(1, 2, 3, 4, 5, 6, 7)
```

[72]:
```
val max = l.filter(i => i % 2 != 0)
   .map(k => k * 2)
   .reduce((l, r) => l max r)
```

[72]:
```
max: Int = 14
```

[73]:
```
val d = List[Double](.1, .2, .3, .4, 1)
```

[73]:
```
d: List[Double] = List(0.1, 0.2, 0.3, 0.4, 1.0)
```

[74]:
```
d.map(i => i + .1).map(a => a * .2).toSeq
```

[74]:
```
res56: scala.collection.immutable.Seq[Double] = List(0.04000000000000001,
0.06000000000000001, 0.08000000000000002, 0.1, 0.22000000000000003)
```

Method Definition

[75]:
```
def sum(a: Int, b: Int): Int = a + b
```

[75]:
```
sum: (a: Int, b: Int)Int
```

[76]:
```
sum(3, 4)
```

[76]:
```
res57: Int = 7
```

[77]:
```
def pow(a: Int, b: Int = 2): Double = Math.pow(a, b)
```

[77]:
```
pow: (a: Int, b: Int)Double
```

[78]:
```
pow(3)
```

[78]:
```
res58: Double = 9.0
```

[79]:
```
implicit val number: Int = 3
```

[79]:
```
number: Int = 3
```

[80]:
```
def sum(a: Int, b: Int)(implicit n:Int): Int = a + b + n
```

[80]:
```
sum: (a: Int, b: Int)(implicit n: Int)Int
```

[81]:
```
sum(4, 5)
```

[81]:
```
res59: Int = 12
```

[82]:
```
def size(a: Int*): Int = a.length
```

[82]:
```
size: (a: Int*)Int
```

[83]:
```
size(0, 1, 2, 3)
```

[83]:
```
res60: Int = 4
```

Loops

[1]:
```
for (i <- 0 to 10) {
    println(i)
}
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://192.168.8.105:4042
SparkContext available as 'sc' (version = 2.2.1, master = local[*], app id =␣
↪local-1573930403810)
SparkSession available as 'spark'


0
1
2
3
4
5
6
7
8
9
10
```

[5]:
```
var items = List(1, 2, 3, 4, 5);

for(item <- items) {
    println(item)
}
```

```
1
2
3
4
5
```

[5]:
```
items: List[Int] = List(1, 2, 3, 4, 5)
```

Classes

Trait Traits are used to share interfaces and fields between classes. They are similar to Java 8's interfaces. Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters. - Similar to interfaces - Supports multiple inheritance - Able to use fields - Can be generic - Able to use abstract methods or non-abstract methods - Cannot be initialized

[7]:
```
trait Color {
   def name(): String

   def fill()
}
```

[7]:
```
defined trait Color
```

[10]:
```
import scala.util.Random

trait User {
   def age(): Int

   def randId(): Int = {
       Random.nextInt();
   }
}
```

[10]:
```
import scala.util.Random
defined trait User
```

Abstract class The working of the Scala abstract class is similar to Java abstract class. In Scala, an abstract class is constructed using the abstract keyword. It contains both abstract and

non-abstract methods and cannot support multiple inheritances.

- Cannot use fields
- Can be generic
- Does not support multiple inheritance
- Able to use constructor parameter
- Able to use abstract methods or define methods
- Cannot be initialized

[11]:
```
abstract class Car(serialCode: String) {
    def model(): String

    def numberOfWheel(): Int = 4
}
```

[11]:
```
defined class Car
```

Class A Scala class is a template for Scala objects. That means, that a class defines what information objects of that class holds, and what behaviour (methods) it exposes. - Cannot use fields - Can be generic - Able to use constructor parameter - Cannot define abstract method - Can be initialized

[17]:
```
class Activity(name: String = "page_view") {
    private[this] val PI: Double = 3.14

    def emit(record: Map[String, String]): Unit = {
        println(s"Emitted ${name}")
    }
}
```

[17]:
```
defined class Activity
```

Object It is a basic unit of Object Oriented Programming and represents the real-life entities. A typical Scala program creates many objects, which as you know, interact by invoking methods. - Able to use fields - Cannot be generic - Cannot use constructor parameter - Cannot define abstract method - Cannot be initialized

[20]:
```
class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
```

```
   }
}

object Demo {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      pt.move(10, 10);
   }
}
```

[20]:
```
defined class Point
defined object Demo
```

RDD This is the most basic data abstraction in Spark, short for resilient distributed dataset. It is a fault-tolerant collection of elements that can be operated on in parallel. - Most basic data structure in Spark - Fault tolerant - Parallel - Immutable - Low level transformation API (does not recommend)

[1]:
```
val r = sc.parallelize(Array(1, 2, 3, 4, 5))
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://ytuegemtincimbp:4040
SparkContext available as 'sc' (version = 2.4.4, master = local[*], app id =␣
↪local-1574019237303)
SparkSession available as 'spark'
```

[1]:
```
r: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:25
```

[2]:
```
val filteredList = r.filter(k => k > 3).collect.toList
```

[2]:
```
filteredList: List[Int] = List(4, 5)
```

DataFrame It is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame but with lot more stuff under the hood. - Structured data structure - Written top of RDD API - Faster than RDD - Easy to use - High level transformation API - Support SQL API

[3]:
```
val r = sc.parallelize(Array(1, 2, 3, 4, 5))
val df = r.toDF
val filteredDF = df.where(col("value") > 4)
```

[3]:
```
r: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at
<console>:27
df: org.apache.spark.sql.DataFrame = [value: int]
filteredDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [value:
int]
```

[4]:
```
filteredDF.show
```

```
+-----+
|value|
+-----+
|    5|
+-----+
```

[5]:
```
val df = spark.read.option("header", "True").csv("file:///tmp/example")
```

[5]:
```
df: org.apache.spark.sql.DataFrame = [cluster: string, number: string]
```

[7]:
```
df.show(5)
```

```
+-------+------+
|cluster|number|
+-------+------+
|      c|   862|
|      d|   225|
|      e|   524|
|      d|   643|
|      b|   628|
+-------+------+
only showing top 5 rows
```

[8]:
```
val g = df.groupBy(col("cluster")).agg(max(col("number")).alias("m"))
```

[8]:
```
g: org.apache.spark.sql.DataFrame = [cluster: string, m: string]
```

[10]:
```
g.show
```

```
+-------+---+
|cluster|  m|
+-------+---+
|      e|999|
|      d|999|
|      c|999|
```

```
|      b|999|
|      a|999|
+-------+---+
```

[11]:
```
g.repartition(1).write.csv("file:///tmp/output")
```

Join in Spark

[25]:
```
val a = spark.read.option("header", "true").csv("file:///tmp/adata/").as("a")
val b = spark.read.option("header", "true").csv("file:///tmp/bdata/").as("b")
```

[25]:
```
a: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: string, name:
string]
b: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: string, age:
string]
```

[26]:
```
val c = a.join(b, col("a.id") === col("b.id")).select("a.id", "name", "age")
```

[26]:
```
c: org.apache.spark.sql.DataFrame = [id: string, name: string … 1 more field]
```

[27]:
```
c.show(5)
```

```
+---+----+---+
| id|name|age|
+---+----+---+
|752| Max| 39|
|752| Max| 37|
|736|John| 36|
|677|Ivan| 30|
|677|Ivan| 36|
+---+----+---+
only showing top 5 rows
```

[30]:
```
val crossA = a.crossJoin(b)
```

[30]:
```
crossA: org.apache.spark.sql.DataFrame = [id: string, name: string … 2 more
fields]
```

[31]:
```
crossA.show(5)
```

```
+---+----+---+---+
| id|name| id|age|
+---+----+---+---+
```

```
|752| Max|752| 37|
|752| Max|736| 36|
|752| Max|677| 36|
|752| Max| 14| 44|
|752| Max|657| 29|
+---+----+---+---+
only showing top 5 rows
```

There are several join types in Spark, - Broadcast hash join - Sort-merge join - Shuffle join

Broadcast hash join, - Provides best performance - Suitable if both dataset is fit in the memory - Create hashtable for the key lookup - spark.sql.autoBroadcastJoinThreshold is the threshold for memory fit

Sort-merge join, - May slower than hash join - Suitable if both dataset is cannot fit in the memory - Sort data before joining - spark.sql.join.preferSortMergeJoin is true by default

Shuffle join, - May slower than hash join - May faster than sort-merge join - Partitioned broadcast join - spark.sql.join.preferSortMergeJoin=false and spark.sql.autoBroadcastJoinThreshold may need some tweak

Benchmark,

Join example 2 tables,

```
+--------+-------------+---------+
| table  | records     | columns |
+--------+-------------+---------+
| users  | 1 million   | 4       |
+--------+-------------+---------+
| orders | 10 millions | 4       |
+--------+-------------+---------+
```

Here is result,

```
+----------------+-------------+-------------+
| type           | time (secs) | peak memory |
+----------------+-------------+-------------+
| Broadcast Hash | 20 seconds  | 424 MB      |
+----------------+-------------+-------------+
| Sort-Merge     | 8 seconds   | 4.2 MB      |
+----------------+-------------+-------------+
| Shuffle Hash   | 34 seconds  | 850 MB      |
+----------------+-------------+-------------+
```

[3]:
```
import org.apache.spark.sql.functions.udf
```

[3]:
```
import org.apache.spark.sql.functions.udf
```

UDF

[1]:
```
val dataset = Seq((0, "hello"), (1, "world")).toDF("id", "text")
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://192.168.8.105:4040
SparkContext available as 'sc' (version = 2.4.4, master = local[*], app id =␣
↪local-1574599439018)
SparkSession available as 'spark'
```

[1]:
```
dataset: org.apache.spark.sql.DataFrame = [id: int, text: string]
```

[2]:
```
val upper: String => String = _.toUpperCase
```

[2]:
```
upper: String => String = <function1>
```

[4]:
```
val upperUDF = udf(upper)
```

[4]:
```
upperUDF: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function1>,StringType,Some(List(StringType)))
```

[5]:
```
dataset.withColumn("upper", upperUDF(col("text"))).show
```

```
+---+-----+-----+
| id| text|upper|
+---+-----+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+-----+
```

[7]:
```
// You could have also defined the UDF this way
// val upperUDF = udf { s: String => s.toUpperCase }

// or even this way
// val upperUDF = udf[String, String](_.toUpperCase)
```

[7]:
```
upperUDF2: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function1>,StringType,Some(List(StringType)))
upperUDF3: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function1>,StringType,Some(List(StringType)))
```