1.

) $ git init

2.) $ git cd D:\FileFolderName

3.) $ git status

4.) If needed to switch the git branch, use this command :

$ git checkout -b DesiredBranch

5.) $ git add .

6.) $ git commit -m "added a new folder"

7)git remote add origin https://github.com/yuvraj777/GDriveDemo.git

8.) $ git push -u origin master

Eclipse git guide

http://wiki.eclipse.org/EGit/User\_Guide#

**Kafka in a nutshell**

<https://sookocheff.com/post/kafka/kafka-in-a-nutshell/>

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Kafka stream is the processing in kafka where data in one topic will be processed and writes to the other topic.

**Some useful Github scala links**

<https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples>

<https://github.com/rathboma/hadoop-framework-examples/tree/master/sparkscala/src/test/scala/com/matthewrathbone/spark>

<http://blog.madhukaraphatak.com/categories/spark-two/>

<https://github.com/RamkSwamy/sparkmlpipeline/tree/master/src/main/scala/com/ram/spark/ml>

<https://github.com/malli3131/HadoopTutorial>

<http://blog.madhukaraphatak.com/>

<https://github.com/databricks/>

Usefule repositories

<https://github.com/PacktPublishing/> // filter with programming in scala

<https://github.com/deanwampler/spark-scala-tutorial/tree/master/src/main/scala/sparkworkshop>  
<https://github.com/andfanilo/spark-project-template/blob/master/pom.xml> -- spark project template  
<https://github.com/knoldus/pure-config-examples> // Config in scala

<https://github.com/pureconfig/>

<https://github.com/grzegorzgajda/spark-examples/tree/master/spark-examples/src/main/scala/examples>

<https://github.com/erikerlandson/spark-kafka-sink> //Spark-kafka-sink

<https://github.com/karande/kafka-producer-file> ///kafka producer

<https://github.com/koeninger/> //Kafka-examples

https://github.com/dibbhatt/kafka-spark-consumer/

https://github.com/tresata

<https://github.com/knoldus/Sparkathon>

https://github.com/Re1tReddy/Spark

https://github.com/LIBBLE/LIBBLE-Spark

https://github.com/XichengDong/sparktraining

https://github.com/xubo245/SparkLearning

https://github.com/mkuthan/example-spark //Good for unit testing strategies.

https://github.com/skrusche63

[https://github.com/spirom](https://github.com/spirom%20%20%20)  //Good for spark example and spark integration with kafka

<https://github.com/Intel-bigdata/spark-streamingsql> //check for spark streaming example

<https://github.com/groupon/sparklint> // to tuning and monitoring spark jobs

<https://github.com/sryza/spark-timeseries> //spark time-series example

[https://github.com/adornes/spark\_scala\_ml\_examples //](https://github.com/adornes/spark_scala_ml_examples%20//) ML example

<https://github.com/TalkingData/Fregata/tree/master/core/src/main/scala/fregata>

<https://github.com/activewizardslab/spark_scala>

<https://github.com/fstasz/spark-cassandra>

<https://github.com/leviathan88/scala-repo>

<https://github.com/LotteryOne/ScalaSpark>

<https://github.com/Chaitanya993/spark-scala> //Elastic search and kafka-spark example

[https://github.com/akashsethi24/Distributed\_OnlineLDA //](https://github.com/akashsethi24/Distributed_OnlineLDA%20//) Must see

<https://github.com/Nitro?language=scala> V-IMP Read  
<https://github.com/anindya-saha/ScalaSpark/>  
<https://github.com/sachindeshpande/SparkScalaModules>  
<https://github.com/prashant-chopra/spark-scala/>

<https://github.com/epfromer>

<https://github.com/kbisko/profiling_code>

<https://github.com/steven-prgm>

<https://github.com/red7blue8pj9/ML_Spark_1.5>

https://github.com/EatMedicineFeelLovely/Spark

https://github.com/AbdullahAli/spark-mysql-scala

Dataset-dataframe-RDD difference

<https://www.linkedin.com/pulse/apache-spark-rdd-vs-dataframe-dataset-chandan-prakash>

Refer below link for good scala programs:-

<https://github.com/azakordonets/fabricator>

Wrapper scripts to call spark-submit jobs:-

<https://github.com/deanwampler/spark-scala-tutorial/blob/master/scripts/hadoop.sh>

sending a file to kafka topic

kafka-console-producer.sh --broker-list localhost:9092 --topic my\_topic < my\_file.txt

Testing Scripts:-

import org.scalatest.FlatSpec

import org.scalatest.{FlatSpec, GivenWhenThen, Matchers}

import com.holdenkarau.spark.testing.SharedSparkContext

import org.scalatest.FunSuite

import com.holdenkarau.spark.testing.{RDDComparisons, RDDGenerator, SharedSparkContext}

import org.scalacheck.Arbitrary

import org.scalacheck.Prop.\_

import org.scalatest.prop.Checkers

import org.scalatest.{FlatSpec, Matchers}

import org.scalatest.Assertions

import org.junit.Test

import org.scalatest.FunSuite

import org.scalatest.matchers.ShouldMatchers

import org.junit.runner.RunWith

import org.scalatest.junit.JUnitRunner

class SampleRddTest extends FunSuite with SharedSparkContext {

logger.info("=> jobName \"" + jobName + "\"")

logger.info("=> pathToFiles \"" + pathToFiles + "\"")

object SparkStructType extends LogHelper {

def main(args: Array[String]) {

logger.info("SparkStructType.main()")

val today = Calendar.getInstance().getTime

println(s"Script executed on $today")

Scala Program 1:

Defining function on the fly,main method , variable declaration

Entry point of scala applications :-

1. **Using the main method:-**
2. **Define an object which extends the App trait:-**

object Hello extends App {

println("Hello, world")

}

The code in the body of the object is automatically run, just as if it were inside a main method.

When using this approach, any command-line arguments to your application are implicitly available through an args object, which is inherited from the App trait. The args object is an instance of Array[String], just as if you had declared a main method yourself. The following code demonstrates how to use the args object:

object Hello extends App {

if (args.length == 1)

println(s"Hello, ${args(0)}")

else

println("I didn't get your name.")

}

**CLASS:-**

class Rational(n: Int, d: Int) {

println("Created "+ n +"/"+ d)

}

Given this code, the Scala compiler would place the call to println into Rational’s primary constructor. The println call will, therefore, print its debug message whenever you create a new Rational instance:

scala> new Rational(1, 2)

Created 1/2

res0: Rational = Rational@90110a

**Important:**

This both will generate the same output:

 a.   val list = List("Rahul Dravid","Sourav","Sachin","Sehwag","Harbhajan"  )

       val val1 = list.flatMap(\_.split(" "))

       println(val1)

 b.    val list = List("Rahul Dravid","Sourav","Sachin","Sehwag","Harbhajan"  )

        val val1 = list.flatMap(x => x.split(" "))

        println(val1)

Map and flatMap explained

cat test.md

This is the first line;

This is the second line;

This is the last line.

scala> val textFile = sc.textFile("test.md")

scala> textFile.map(line => line.split(" ")).count()

res2: Long = 3

scala> textFile.flatMap(line => line.split(" ")).count()

res3: Long = 15

scala> textFile.map(line => line.split(" ")).collect()

res0: Array[Array[String]] = Array(Array(This, is, the, first, line;), Array(This, is, the, second, line;), Array(This, is, the, last, line.))

scala> textFile.flatMap(line => line.split(" ")).collect()

res1: Array[String] = Array(This, is, the, first, line;, This, is, the, second, line;, This, is, the, last, line.)

**Apply Method:-**

 class Multiplier(factor: Int) {

        def apply(input: Int) = input \* factor

      }

object HelloObjectPlusClass {

  println("In object")

  def main(args: Array[String]) {

    println("In main")

      val tripleMe = new Multiplier(3)

      val tripled = tripleMe.apply(10)

      val tripled2 = tripleMe(10)

       println("tripled: " + tripled + "  tripled2: " + tripled2 )

  }

}

**=>**

When examining this code, it helps to think of the => symbol as a transformer, because the expression transforms the parameter list on the left side of the symbol into a new result using the algorithm on the right side of the symbol

val evens = x.filter((i: Int) => i % 2 == 0)

val evens = x.filter(i => i % 2 == 0)

val evens = x.filter(\_ % 2 == 0)

x.foreach((i:Int) => println(i))

x.foreach((i) => println(i))

x.foreach(i => println(i))

x.foreach(println(\_))

x.foreach(println)

Apply method is being called without calling it explicitly

import java.io.\_

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);

// Move to a new location

pt.move(10, 10);

val inc = (test : Int) => test + 1

inc(1)

println (inc(1))

val inc2 = List(1, 2, 3).map((x: Int) => x + 1)

println (inc2)

}

}

Output:-

sh-4.3$ scalac Demo.scala

sh-4.3$ scala Demo

Point x location : 20

Point y location : 30

2

List(2, 3, 4)

**Object keyword explained**

class HelloClass {

  def inClass() {

    println("In class")

  }

}

object HelloObjectPlusClass {

  println("In object")

  def main(args: Array[String]) {

    println("In main")

    val c = new HelloClass

    c.inClass()

  }

}

dave% scalac HelloWithClass.scala

dave% scala HelloObjectPlusClass

In object

In main

In class

Note :- The object **may not** have the same name as the class!.

If you put main function in HelloClass and run the HelloClass, It will throw a runtime error as

**java.lang.NoSuchMethodException: HelloClass.main is not static**

scala HelloObjectPlusClass will work because the object keyword is static.

Objects and classes are not completely decoupled. An object can extend another class, making its fields and methods available in a global instance. The reverse is not true, however, because an object cannot itself be extended.

object Demo {

def main(args: Array[String]) {

println ("Apply method : " + apply("Zara", "gmail.com"));

println ("Unapply method : " + unapply("Zara@gmail.com"));

println ("Unapply method : " + unapply("Zara Ali"));

}

// The injection method (optional)

def apply(user: String, domain: String) = {

user +"@"+ domain

}

// The extraction method (mandatory)

def unapply(str: String): Option[(String, String)] = {

val parts = str split "@"

if (parts.length == 2){

Some(parts(0), parts(1))

}else{

None

}

}

}

Outout :

sh-4.3$ scala Demo

Apply method : Zara@gmail.com

Unapply method : Some((Zara,gmail.com))

Unapply method : None

sh-4.3$

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      val sum = List(1,2,3,4,5).reduceLeft(\_+\_)

      val sum1 = List(1,2,3,4,5).reduceLeft((a, b) => a + b)

      val nums = List(1,2,3,4,5,6,7,8,9,10).filter(\_ % 2 == 0)

      val num2 = ("sumeet",1,"Ajay")

      println(num2.\_2)

      println(sum)

      println(nums)

      println(sum1)

   }

}

Output:-

sh-4.3$ scala HelloWorld

Hello, world!

15

List(2, 4, 6, 8, 10)

15

sh-4.3$ ^C

**Operators:- :: and ::: difference (primarily used for concat)**

val list1 = List(1,2)

val list2 = List(3,4)

list1::list2 returns:

List[Any] = List(List(1, 2), 3, 4)

list1:::list2 returns:

List[Int] = List(1, 2, 3, 4)

**Prgm 2**

Here the Input data was in the format of **(("Key1", 1),("Key2", 2),("Key3",3),("Key2",4),("Key1",4),("Key2",1),("Key3",1)**

To convert it to a MAP with most recent values of the keys, call the toMap method

To group all the values with the same keys

val val1 = list.groupBy(\_.\_1).mapValues(x => x.map(\_.\_2))

and to sum all the values for a particular key call the function:-

   val val3 = val1.map({case (key,count) => (key)->count.reduceLeft(\_+\_)})

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

    val list = List(("Key1", 1),("Key2", 2),("Key3",3),("Key2",4),("Key1",4),("Key2",1),("Key3",1)  )

    val val1 = list.groupBy(\_.\_1).mapValues(x => x.map(\_.\_2))

     println(val1)

   val val3 = val1.map({case (key,count) => (key)->count.reduceLeft(\_+\_)})

    /\* val val2 = val1.keys.foreach{ i =>

                           print( "Key = " + i )

                           println(" Value = " + val1(i).reduceLeft(\_+\_) )} \*/

     print(val3.count())

   // val list1 = list.toMap

   }

}

Output:-

3sh-4.3$ scala HelloWorld.scala

Hello, world!

Map(Key3 -> List(3, 1), Key2 -> List(2, 4, 1), Key1 -> List(1, 4))

Map(Key3 -> 4, Key2 -> 7, Key1 -> 5)

**Prgrm 3 : Per key average**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

    val list = List(("Key1", 1),("Key2", 2),("Key3",3),("Key2",4),("Key1",4),("Key2",1),("Key3",1)  )

    val val1 = list.groupBy(\_.\_1).mapValues(x => x.map(\_.\_2).map(\_.toDouble))

 val val3 = val1.map({case (key,count) => (key)->(count.reduceLeft(\_+\_)/count.size)})

/\* val val2 = val1.keys.foreach{ i =>

                           print( "Key = " + i )

                           println(" Value = " + val1(i).size )}

\*/

     println(val3)

    }

}

mapValues : A new RDD is formed  by applying a function on each value of the source RDD. Keys are retained without a change, which implies that any key based partitioning of the source is also retained. There is one to one correspondence between input and output elements.

val list = sc.sparkContext.parallelize(Seq(("Key1", 1),("Key2", 2),("Key3",3),("Key2",4),("Key1",4),("Key2",1),("Key3",1) ))

val val2 = list.mapValues(x=> (x,1))

val2.foreach(println)

(Key1,(1,1))

(Key2,(2,1))

(Key3,(3,1))

(Key2,(4,1))

(Key1,(4,1))

(Key2,(1,1))

(Key3,(1,1))

flatMapValues:-Same as above except there is a one to many correspondence between input and output elements if f returns a seq with more than one element

sh-4.3$ scala HelloWorld.scala

Hello, world!

Map(Key3 -> 2.0, Key2 -> 2.3333333333333335, Key1 > 2.5)

**Splitting a list and change it to upper case**

object hello {

def main(args:Array[String]) {

//val cities6chars = cities.map(p => p).filter(\_.name == "Warsaw").map(\_.name.toUpperCase)

val cities = List("Hello", "warSaw", "Ji")

val cit1 = cities.flatMap(p => p.split(",")).foreach { x => println(x.toUpperCase()) }

// val cit2 = cit1.

// cities.Foreach(println)

}

}

**How to create a pair RDD’s**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      val a = List (1, 2, 1, 3)

      val b = a.map ((\_, "b"))

           println(c)

   }

}

Output:-

sh-4.3$ scala HelloWorld

Hello, world!

List((1,b), (2,b), (1,b), (3,b))

**Partition Example**

object HelloWorld {

   def main(args: Array[String]) {

      //println("Hello, world!")

     // val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

     // println(numbers.partition(\_ % 2 == 0))

      val s = List("Hello World India republic day World Hello Python tree")

   //   println(s.flat (x => "." + x))

      val k = s.flatMap(x => x.split(" "))

      println(k)

      val counts = k.partition(\_.length() % 2 == 0)

      println(counts)

   }

}

We can actually implement word count even faster by using the countByValue() function on the first RDD: input.flatMap(x

=> x.split(" ")).countByValue().

Output

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

List(Hello, World, India, republic, day, World, Hello, Python, tree)

(List(republic, Python, tree),List(Hello, World, India, day, World, Hello))

**Case Class Example1:-**

case class Person(firstName: String, lastName: String)

val me = Person("Daniel", "Spiewak")

val first = me.firstName

val last = me.lastName

if (me == Person(first, last)) {

  println("Found myself!")

  println(me)

}

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Found myself!

Person(Daniel,Spiewak)

**Case Class Example2**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      case class Person(lastname: String, firstname: String, birthYear: Int)

      val p = Person("Lacava", "Alessandro", 1976)

sealed trait Maybe[+T]

case class Value[T](value: T) extends Maybe[T]

case object NoValue extends Maybe[Nothing]

val v: Maybe[Int] = Value(42)

val v\_1: Maybe[Int] = NoValue

def logValue[T](value: Maybe[T]): Unit = value match {

  case Value(v) => println(s"We have a value here: $v")

  case NoValue => println("I'm sorry, no value")

}

logValue(v) // prints We have a value here: 42

logValue(v\_1) // prints I'm sorry, no value

   }

}

OUTPUT:-

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Hello, world!

We have a value here: 42

I'm sorry, no value

**Example Even Odd**

object HelloWorld {

def main(args: Array[String]) {

for (n <- 1 to 10) n % 2 match {

case 0 => println("even")

case 1 => println("odd")

}

}

}

**While, Do while and for loop**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      var x = 1

      var y = 1

      while (x < 10){

  x \*= 2

  println(x)

 // y += 1

}

var x = 1

do {

  x \*= 2

} while (x < 1000)

      val num = List(1,2,3,4,5)

      for (x <- num)

  println(x)

for (i <- 1 to 10)

  println(i)

      var J : Int = 10

      for (i <- 1 until J){

  println(i)

for( a <- 1 to 3; b <- 1 to 3){

println( "Value of a: " + a );

println( "Value of b: " + b );

**//Infinite loop**

      var a = 10;

while( true ){

println( "Value of a: " + a );

   }

   }

}

When using a for loop, the <- symbol can be read as “in”.

**Scala functions:-**

def functionName ([list of parameters]) : [return type] = {

function body

return [expr]

}

whenever you leave off the equals sign before the body of a function, its result type will definitely be Unit.

This is true no matter what the body contains, because the Scala compiler can convert any type to Unit.

For Example:-

def add(b: Byte) { sum += b }

In this example, the String is converted to Unit because Unit is the declared result type of function f. The Scala compiler treats a function defined in the procedure style, i.e., with curly braces but no equals sign, essentially the same as a function that explicitly declares its result type to be Unit

def f(): Unit = "this String gets lost"

In this function it will return a Non-unit return type:-

def h() = { "this String gets returned!" }

scala> h

res0: java.lang.String = this String gets returned!

Higher Order Functions

Scala allows the definition of higher-order functions. These are functions that take other functions as parameters, or whose result is a function. Here is a function apply which takes another function f and a value v and applies function f to v:

class Decorator(left: String, right: String) {

def layout[A](x: A) = left + x.toString() + right

}

object FunTest extends App {

def apply(f: Int => String, v: Int) = f(v)

val decorator = new Decorator("[", "]")

println(apply(decorator.layout, 7))

}

**Example1:-**

object add{

def addInt( a:Int, b:Int ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

**Example2:**

object HelloWorld {

    def main(args: Array[String]) = println("Hello Scala!")

 val ds = List(1,2,3,4,5,6)

  val val1= ds.map(\_ + 1)

    println(val1)

}

**Anonymous function :**

Note:- The type A => B is the type of a function that takes an argument of type A and returns a result of type B.

So, Int => Int is the type of functions that map integers to integers.

var inc = (x:Int) => x+1

var mul = (x: Int, y: Int) => x\*y

Now inc and mul are anonymous functions.

Example:-

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      var num = (x: Int) => x\*x\*x

       println(num(3))

   }

}

**Map and convert to IndexedSeq(Row) from Integer 1 to 100**

val data = (1 to 100).map { x: Int =>

x match {

case t if t <= 50 => Row("A", t.toDouble)

case t => Row("B", t.toDouble)

}

}

data.foreach(x => println(x))

**IMPLICIT KEYWORD**

Implicitly is avaliable in Scala 2.8 and is defined in [Predef](http://www.scala-lang.org/api/current/scala/Predef$.html) as:

def implicitly[T](implicit e: T): T = e

scala> implicit val a = "test" // define an implicit value of type String

a: java.lang.String = test

scala> val b = implicitly[String] // search for an implicit value of type String and assign it to b

b: String = test

scala> val c = implicitly[Int] // search for an implicit value of type Int and assign it to c

<console>:6: error: could not find implicit value for parameter e: Int

val c = implicitly[Int]

**Passing a List as an argument to a function**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      val numbers = List(1,2,3,4,5,6,7,8,9,10,1000).map(\_.toDouble)

      println("New List:  "  +  removeOutliers(numbers))

   }

    def removeOutliers(rdd: List[Double]): List[Double] = {

        val nums = rdd.filter(\_<= 100).map(x => x\*2)

        return nums

}

}

**Practice 1**

object HelloWorld {

   def main(args: Array[String]) {

      //println("Hello, world!")

     // val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

     // println(numbers.partition(\_ % 2 == 0))

      val s = List("Hello World India republic day World Hello Python tree")

   //   println(s.flatMap(x => "." + x))

      val k = s.flatMap(x => x.split(" "))

          .map(word => word + "Ind")

      println(k)

      val counts = k.partition(\_.length() % 2 == 0)

      println(counts)

   }

}

**How to refer any item from a list (lift) method:-**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      val list = List("Rahul Dravid","Sourav","Sachin","Sehwag","Harbhajan"  )

        println(list.lift(3))

   }

}

sh-4.3$ scala HelloWorld.scala

Hello, world!

Some(Sehwag)

**Very important concepts on functions:**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      var num = (x: Int) => x\*x\*x

      println(num(3))

       println(sumInts(3,4))

   }

/\*def sumInts(a:Int, b:Int) =  sum(id,a,b) \*/

def id(x :Int): Int = x

def sumInts(a:Int, b:Int) = sum (x => x,a,b)

def sum (f:Int => Int, a:Int, b:Int): Int =

if (a>b) 0

else f(a) + sum(f, a+1, b)

}

Output

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Hello, world!

7

50

Explanation :

Var num is a

Similar program as above with slight change in function return  id = X\*X

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      var num = (x: Int) => x\*x\*x

      println(num(3))

       println(sumInts(3,5))

   }

/\*def sumInts(a:Int, b:Int) =  sum(id,a,b) \*/

def id(x :Int): Int = x \* x

def sumInts(a:Int, b:Int) = sum (id,a,b)

def sum (f:Int => Int, a:Int, b:Int): Int =

if (a>b) 0

else f(a) + sum(f, a+1, b)

}

Output:-

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Hello, world!

27

50

Program 1: Functions more examples

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

      val x = 0

      println(x)

      def f(y: Int) = y+1

      val result = {

          val x = f(3)

          println(x\*x)

          println(x)

      }

   }

}

sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Hello, world!

0

16

4

**Hash Map**

// Empty hash table whose keys are strings and values are integers:

var A:Map[Char,Int] = Map()

// A map with keys and values.

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")

**Tuple**

val t = new Tuple3(1, "hello", Console)

val t = (1, "hello", Console)

iterate over the tuple

val t = (4,3,2,1)

t.productIterator.foreach{ i =>println("Value = " + i )}

**Pattern matching :**

**Prgrm 1:**

import scala.util.matching.Regex

object Demo {

def main(args: Array[String]) {

println (addInt("Scala is Scalable and cool"))

}

def addInt( a:String ) : Option[String] = {

    var str : String = a

    val pattern = "scala".r

     return (pattern findFirstIn str)

}

}

**Prgrm 2:**

import scala.util.matching.Regex

object Demo {

def main(args: Array[String]) {

addInt("ablaw is able1 and coo")

}

def addInt( a:String ) : Unit = {

    var str : String = a

    val pattern = new Regex("abl[ae]\\d+")

// val str = "ablaw is able1 and cool"

     println (pattern findAllIn str)

}

}

Note:-Option [T] produces Some(value) if a value corresponding to a given key has been found, or None if the given key is not defined in the Map.

Convert String to List

Input - Val xyz = “Is it possible to convert a String into a List of Long”

Output – List(Is, it, possible, to, convert, a, String, into, a, List, of, Long)

Program:-

object HelloWorld {

   def main(args: Array[String]) {

        println("Hello, world!")

        val string = "Is it possible to convert a String into a List of Long"

        val k = string.split(" ").toList

        println(k)

 }

}

Output:- sh-4.3$ scalac HelloWorld.scala

sh-4.3$ scala HelloWorld

Hello, world!

List(Is, it, possible, to, convert, a, String, into, a, List, of, Long)

Convert a list to string

       val l = k.mkString(" ")

object HelloWorld {

   def main(args: Array[String]) {

        println("Hello, world!")

        val string = "Is it possible to convert a String into a List of Long"

      //val string2 = List("Hello","world")

        val k = string.split(" ").toList

        println(k)

       val l = k.mkString(" ")

        println("in L:" + l)

        println(string)

 }

}

List(("Key1", 1),("Key2", 2),("Key3",3),("Key2",4),("Key1",4),("Key2",5),("Key3",1)  )

How Json is loaded in spark :-

val sqlCtx = new SQLContext(sc)

val input = sqlCtx.jsonFile(inputFile)

input.printSchema()

**How to create paid RDD’s from Rdd’s**

object HelloWorld {

   def main(args: Array[String]) {

      println("Hello, world!")

           val list = List("Key1","Key2","Key3","Key2","Key1","Key2","Key3")

       val list1 = list.map(x => (x.split(" ")(0), 1))

println(list1)

   }

}

**Simple filter program in scala**

object HelloWorld {

    def main(args: Array[String]) = println("Hello Scala!")

    val str = List(1,2,3,4,5)

    val res1 = str.filter(x => x!=1)

    println(res1)

}

**Factorial program in scala**

import scala.util.matching.Regex

object Demo {

def main(args: Array[String]) {

//var a:Int = 0

println (factorial(5))

}

def factorial(n: Int): Int = {

            var a = 1;

            var b = n;

    while(b>=1){

    a = a \* b;

   b = b - 1;

    }

          return a;

}

}

**A good example to invoke function**

   val file = sc.textFile(“file:///C:/CHANGES.txt”)

    val words = file.flatMap(line => tokenize(line))

    val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_)

    wordCounts.saveAsTextFile(args(1))

  }

  // Split a piece of text into individual words.

  private def tokenize(text : String) : Array[String] = {

    // Lowercase each word and remove punctuation.

    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+")

  }

}

**getorElse method**

package main.scala

import scala.util.matching.Regex

object TestScala {

def main(args: Array[String]) {

//var a:Int = 0

println(toInt("a").getOrElse(0))

val result = toInt("shiv") match {

case Some(x) => x

case None => 0 // however you want to handle this

}

println(result)

}

}

def toInt(s: String):Option[Int] = {

try {

Some(s.toInt)

} catch {

case e: NumberFormatException => None

}

}

}

**Scala Random Number generation**

**val r = scala.util.Random**

scala> **r.nextInt**

res0: Int = −1323477914

You can limit the random numbers to a maximum value:

scala> **r.nextInt(100)**

**Foreach**  
Array[Int] = Array(1, 2, 3)  
scala> array.foreach(println)

1  
2  
3

**Forloop for multiple counters**

**for (i <- 1 to 2; j <- 1 to 2) println(s"i = $i, j = $j")**

i = 1, j = 1

i = 1, j = 2

i = 2, j = 1

i = 2, j = 2

**def** abs(x**: Int**) **= if** (x >= 0) x **else** -x

**def** max(a**: Int**, b**: Int**) **= if** (a > b) a **else** b

def add(b: Byte) { sum += b } ---- Return Unit

def f(): Unit = "this String gets lost" –-- Return Unit

def h() = { "this String gets returned!"} ---- Return Non-unit

**def** getStockInfo **=** ("NFLX", 100.00) will return a tuple

Val (name1,name2) = getStockInfo

val result = getStockInfo

result.\_1 and result.\_2

**val** c **= if** (a > b) a **else** b

Access modifier Description

private[this] --The method is available only to the current instance of the class it’s declared in.

private -- The method is available to the current instance and other instances of the class it’s declared in.

protected -- The method is available only to instances of the current class and subclasses of the current class.

private[model] -- The method is available to all classes beneath the com.acme.coolapp.model package.

private[coolapp] -- The method is available to all classes beneath the com.acme.coolapp package.

private[acme] -- The method is available to all classes beneath the com.acme package.

(no modifier) -- The method is public.

**Scala inheritance:**

class FourLeggedAnimal {

def walk { println("I'm walking") }

def run { println("I'm running") }

}

class Dog extends FourLeggedAnimal {

def walkThenRun {

super.walk

super.run

}

}

Running this code in the Scala REPL yields:

scala> val suka = new Dog

suka: Dog = Dog@239bf795

scala> suka.walkThenRun

I'm walking

I'm running

**Traits,Override and control traits method**

package main.scala

trait Human {

def hello = "the Human trait"

}

trait Mother extends Human {

override def hello = "Mother"

}

trait Father extends Human {

override def hello = "Father"

}

class Child extends Human with Mother with Father {

def printSuper = super.hello

def printMother = super[Mother].hello

def printFather = super[Father].hello

def printHuman = super[Human].hello

}

object TestApp1 extends App {

val c = new Child

println(s"c.printSuper = ${c.printSuper}")

println(s"c.printMother = ${c.printMother}")

println(s"c.printFather = ${c.printFather}")

println(s"c.printHuman = ${c.printHuman}")

}

**File Handling program**

import java.io.FileReader

import java.io.FileNotFoundException

import java.io.IOException

object Demo {

def main(args: Array[String]) {

try {

val f = new FileReader("input.txt")

} catch {

case ex: FileNotFoundException =>{

println("Missing file exception")

}

case ex: IOException => {

println("IO Exception")

}

}

}

}

**To declare the methods which can throw exceptions:-**

@throws(classOf[IOException])

@throws(classOf[LineUnavailableException])

@throws(classOf[UnsupportedAudioFileException])

def playSoundFileWithJavaAudio {

// exception throwing code here ...

}

**a1.getClass is used to get the name of the class of an object.**

**Creation of object and constructor without new keyword:\_**

class Person {

var name = ""

var age = 0

}

object Person {

// a one-arg constructor

def apply(name: String): Person = {

var p = new Person

p.name = name

p

}

// a two-arg constructor

def apply(name: String, age: Int): Person = {

var p = new Person

p.name = name

p.age = age

p

}

val fred = Person("Fred")

val john = Person("John", 42)

**object** TestApp {

**def** main(args: Array[*String*]) {

**val** (name1,name2) = getStockInfo

println(name1)

}

**def** getStockInfo = ("NFLX", 100.00)

}

**Assign function literal to a variable**

val double = (i: Int) => {i \* 2 }

println(double(2))

Res: 4

Val double can be passed a parameter in any method as well.

scala> val list = List.range(1, 5)

list: List[Int] = List(1, 2, 3, 4)

scala> list.map(double)

res0: List[Int] = List(2, 4, 6, 8)

// implicit approach

val add = (x: Int, y: Int) => { x + y }

val add = (x: Int, y: Int) => x + y

// explicit approach

val add: (Int, Int) => Int = (x,y) => { x + y }

val add: (Int, Int) => Int = (x,y) => x + y

val coder: (Int => String) = (arg: Int) => { if (arg < 300) "slow" else "high" }

**VERY IMPORTANT**

scala> val sayHello = () => { println("Hello") }

sayHello: () => Unit = <function0> --- Takes no parameter, return type unit

scala> val add: (Int, Int) => Int = (x,y) => x + y

add: (Int, Int) => Int = <function2>

scala> val plusOne = (i: Int) => { println(i+1) }

plusOne: Int => Unit = <function1> --- Takes Int as a parameter

val coder: (Int => String) = (arg: Int) => { if (arg < 300) "slow" else "high" }

coder: Int => String = <function1>

scala> val sayHello = () => { 2 }

sayHello: () => Int = <function0>

As long as your function signature matches what your method expects, your algorithms can do anything you want.

def exec(callback: Int => Unit) {

// invoke the function we were given, giving it an Int parameter

callback(1)

}

val plusOne = (i: Int) => { println(i+1) }

exec(plusOne) will result 2

Any function that matches this signature can be passed into the exec method.

To understand it better from the below example

def executeAndPrint(f:(Int, Int) => Int, x: Int, y: Int) { // Define a method

val result = f(x, y)

println(result)

}

val sum = (x: Int, y: Int) => x + y //Define a function which takes two parameters and give one result of Int datatype

val multiply = (x: Int, y: Int) => x \* y ////Define a function which takes two parameters and give one result of Int datatype

executeAndPrint(sum, 2, 9) // prints 11

executeAndPrint(multiply, 3, 9) // prints 27

Summary notes:

• Think of the => symbol as a transformer. It transforms the input data on its left side

to some new output data, using the algorithm on its right side.

• Use def to define a method, val, to create a function.

• When assigning a function to a variable, a function literal is the code on the right

side of the expression.

• A function value is an object, and extends the FunctionN traits in the main scala

package, such as Function0 for a function that takes no parameters.

**Difference between list and tuples:-**

Both are immutable

List has only one type. For Example:- List(Int) or List(Any) or List(String)

List elements can be accessed by pair(0)

scala> val pair = List(1,2,3)

pair: List[Int] = List(1, 2, 3)

scala> pair(1)

res3: Int = 2

Eeach element of tuple can have diff type. For example below:-

scala> val tup = (1,"Hello",2)

tup: (Int, String, Int) = (1,Hello,2)

scala> tup.\_1

res7: Int = 1

scala> tup.\_2

res8: String = Hello

**def** modMethod(i: Int) = i % 2 == 0

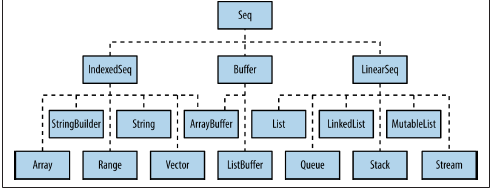
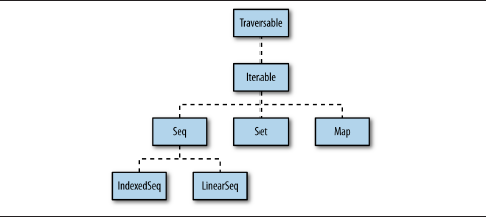
println(list.filter(modMethod))

**val** modFunction = (i: Int) => i % 2 == 0

println(list.filter(modFunction))

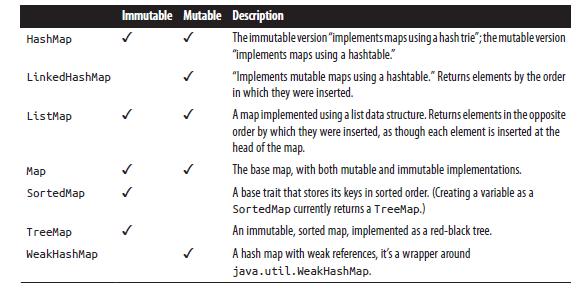
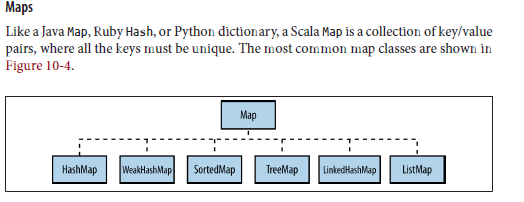
}

Scala Collections:-



Defining a empty map:-

val treasureMap = Map[Int, String]()



When you just need a simple, immutable map, you can create one without requiring an import:

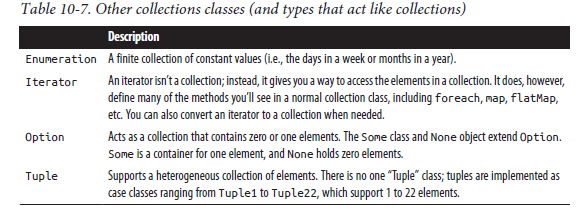
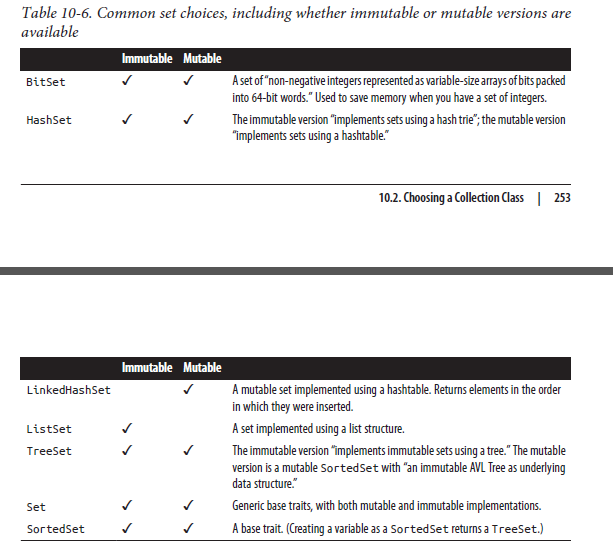
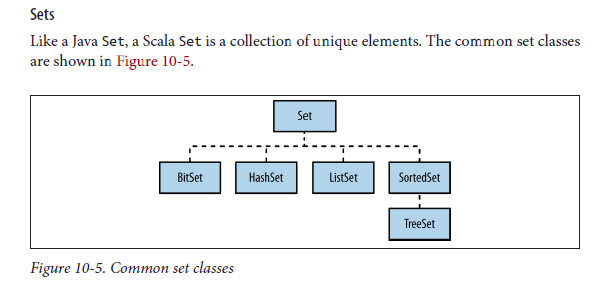
scala> val m = Map(1 -> "a", 2 -> "b")

The *mutable* map is not in scope by default, so you must import it (or specify its full

path) to use it:

scala> **val m = collection.mutable.Map(1 -> "a", 2 -> "b")**

LinkedHashMap to store elements in insertion order



scala.collection.immutable - Immutable, sequential data-structures such as Vector, List, Range, HashMap or HashSet

**scala.collection.mutable** - Mutable, sequential data-structures such as ArrayBuffer, StringBuilder, HashMap or HashSet

**scala.collection.concurrent** - Mutable, concurrent data-structures such as TrieMap

**scala.collection.parallel.immutable** - Immutable, parallel data-structures such as ParVector, ParRange, ParHashMap or ParHashSet

**scala.collection.parallel.mutable** - Mutable, parallel data-structures such as ParArray, ParHashMap, ParTrieMap or ParHashSet

**scala.concurrent** - Primitives for concurrent programming such as Futures and Promises

**scala.io** - Input and output operations

**scala.math** - Basic math functions and additional numeric types like BigInt and BigDecimal

**scala.sys** - Interaction with other processes and the operating system

scala.util.matching - Regular expressions

**Strict and lazy collections :-**

In a strict collection, memory for the elements is allocated immediately, and all of its

elements are immediately evaluated when a transformer method is invoked. In a lazy

collection, memory for the elements is not allocated immediately, and transformer

methods do not construct new elements until they are demanded.

**IMPORTANT**  
• A mutable variable (var) can be reassigned to point at new data.

• An immutable variable (val) is like a final variable in Java; it can never be

reassigned.

To be clear about collections:

• The elements in a mutable collection (like ArrayBuffer) can be changed.

• The elements in an immutable collection (like Vector) cannot be changed.

using immutable variables with mutable collections as a “best practice.”

Defining and adding elements in a Vector

Vector is an immutable collection.

If you create an instance of an IndexedSeq, Scala returns a Vector:

scala> val x = IndexedSeq(1,2,3)

x: IndexedSeq[Int] = Vector(1, 2, 3)

scala> **var a = Vector(1, 2, 3)**

scala> **a = a ++ Vector(4, 5)**

res a: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)

To update

a = a.updated(0,7)

a: scala.collection.immutable.Vector[Int] = Vector(7, 2, 3, 4, 5)

To Add

a = a ++ Vector(8)

a: scala.collection.immutable.Vector[Int] = Vector(7, 2, 3, 4, 5, 8)

scala> val a = Vector(1, 2, 3)

scala> **a = a ++ Vector(4, 5)**

**<console>:21: error: reassignment to val**

**a = a ++ Vector(4, 5)**

Just as the Vector is the recommended “go to” class for immutable, sequential collections,

the ArrayBuffer class is recommended as the general-purpose class for mutable sequential collections. (ArrayBuffer is an indexed sequential collection. Use ListBuffer if you prefer a linear sequential collection that is mutable.

import scala.collection.mutable.ArrayBuffer

You can then create an empty ArrayBuffer:

var fruits = ArrayBuffer[String]()

var ints = ArrayBuffer[Int]()

var nums = ArrayBuffer(1, 2, 3)

scala> nums += 4

res0: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)

// add elements from another collection

scala> nums ++= List(7, 8)

res2: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)

Note:- elements are added and removed from the syntax += and ++= and -= and --=

Some more methods:-

val a = ArrayBuffer(1, 2, 3)

a.append(4) // ArrayBuffer(1, 2, 3, 4)

val a = ArrayBuffer(9, 10) // ArrayBuffer(9, 10)

a.insert(0, 8) // ArrayBuffer(8, 9, 10*)*

val a = ArrayBuffer.range('a', 'h') // ArrayBuffer(a, b, c, d, e, f, g)

a.remove(0) // ArrayBuffer(b, c, d, e, f, g)

a.remove(2, 3) // ArrayBuffer(b, c, g)

val a = ArrayBuffer.range('a', 'h') // ArrayBuffer(a, b, c, d, e, f, g)

a.trimStart(2) // ArrayBuffer(c, d, e, f, g)

a.trimEnd(2) // ArrayBuffer(c, d, e)

Note:- “A ListBuffer is like an array buffer except that it uses a linked list internally instead of an array”

For each and if loop in a single statement:-

"Hello world it's Al".split(" ").foreach{ e =>

if (e.length > 4) longWords.append(s" $e")

else println("Not added: " + e)

}

scala> **val fruits = Array("apple", "banana", "orange")**

fruits: Array[String] = Array(apple, banana, orange)

scala> **for (i <- 0 until fruits.size) println(s"element $i is ${fruits(i)}")**

element 0 is apple

element 1 is banana

element 2 is orange

## StringBuilders:-

Just like an array buffer is useful for building arrays, and a list buffer is useful for building lists, a [StringBuilder](http://www.scala-lang.org/api/2.12.0/scala/collection/mutable/StringBuilder.html) is useful for building strings. String builders are so commonly used that they are already imported into the default namespace. Create them with a simple new StringBuilder, like this.

To build a new collection from an input collection, use the for/yield construct.

scala> val fruits = Array("apple", "banana", "orange")

fruits: Array[java.lang.String] = Array(apple, banana, orange)

scala> val newArray = for (e <- fruits) yield e.toUpperCase

newArray: Array[java.lang.String] = Array(APPLE, BANANA, ORANGE)

**Defining MAP and iterating:**

scala> val names = Map("fname" -> "Ed", "lname" -> "Chigliak")

names: scala.collection.immutable.Map[String,String] = Map(fname -> Ed, lname -> Chigliak)

scala> for ((k,v) <- names) println(s"key: $k, value: $v")

key: fname, value: Ed

key: lname, value: Chigliak

**Transferring one collection from another**

scala> val fruits = Vector("apple", "banana", "lime", "orange")

fruits: scala.collection.immutable.Vector[String] = Vector(apple, banana, lime, orange)

scala> val ucFruits = for (e <- fruits) yield e.toUpperCase

ucFruits: scala.collection.immutable.Vector[String] = Vector(APPLE, BANANA, LIME, ORANGE)

scala> ucFruits.foreach(println)

APPLE

BANANA

LIME

ORANGE

**For and if loop together:-**

val x = for (e <- fruits if e.length < 6) yield e.toUpperCase

scala> val cars = Vector("Mercedes", "Porsche", "Tesla")

cars: Vector[String] = Vector(Mercedes, Porsche, Tesla)

scala> for {

| c <- cars

| if c.startsWith("M")

| } yield c

res0: Vector[String] = Vector(Mercedes)

Important Methods:-

**reduceLeft:**

reduceLeft starts by comparing the first two elements in the collection with your algorithm, and returns a result. That result is compared with the third element, and that comparison yields a new result. That result is compared to the fourth element to yield a new result, and so on.

scala> val a = Array(12, 6, 15, 2, 20, 9)

scala> a.reduceLeft(\_ + \_)

res0: Int = 64

scala> a.reduceLeft(\_ \* \_)

res1: Int = 388800

scala> a.reduceLeft(\_ min \_)

res2: Int = 2

scala> a.reduceLeft(\_ max \_)

res3: Int = 20

scala> val peeps = Vector("al", "hannah", "emily", "christina", "aleka")

// longest

scala> peeps.reduceLeft((x,y) => if (x.length > y.length) x else y)

res0: String = christina

// shortest

scala> peeps.reduceLeft((x,y) => if (x.length < y.length) x else y)

res1: String = al

**foldLeft:** The foldLeft method works just like reduceLeft, but it lets you set a seed value to be

used for the first element.

scala> val a = Array(1, 2, 3)

a: Array[Int] = Array(1, 2, 3)

scala> a.reduceLeft(\_ + \_)

res0: Int = 6

scala> a.foldLeft(20)(\_ + \_)

res1: Int = 26

reduceRight

foldRight

**Extracting Unique Elements from a Sequence**

scala> val x = Vector(1, 1, 2, 3, 3, 4)

scala> val y = x.distinct

scala> val s = x.toSet

**Merging Sequential Collections**

• Use the ++= method to merge a sequence into a mutable sequence.

• Use the ++ method to merge two mutable or immutable sequences.

• Use collection methods like union, diff, and intersect.

**MUTABLE**

scala> **val a = collection.mutable.ArrayBuffer(1,2,3)**

a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

scala> **a ++= Seq(4,5,6)**

res0: a.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

**IMMUTABLE**

scala> **val a = Array(1,2,3)**

a: Array[Int] = Array(1, 2, 3)

scala> **val b = Array(4,5,6)**

b: Array[Int] = Array(4, 5, 6)

scala> **val c = a ++ b**

c: Array[Int] = Array(1, 2, 3, 4, 5, 6)

Common elements:- **val c = a.intersect(b)**

All elements from both collections :- scala> **val c = a.union(b)**

Distinct elements from both collections :- scala> **val c = a.union(b).distinct**

c: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)

**Difference :- val c = a diff b  
scala> val a = Array(1,2,3,4,5)**

**a: Array[Int] = Array(1, 2, 3, 4, 5)**

**scala> val b = Array(4,5,6,7,8)**

**b: Array[Int] = Array(4, 5, 6, 7, 8)**

**scala> val c = a diff b**

**c: Array[Int] = Array(1, 2, 3)**

**Concat:- val c = a ::: b**

toArray toBuffer toIndexedSeq toIterable toIterator

toList toMap toSeq toSet toStream

toString toTraversable

|  |  |
| --- | --- |
| Mutable | Immutable |
| ListBuffer | List |
| ArrayBuffer | Array |
| **collection.mutable.Map** | MAP |
|  | vector |
|  |  |
|  |  |
|  |  |

Because a List is immutable, if you need to create a list that is constantly changing, the

preferred approach is to use a ListBuffer while the list is being modified, then convert

it to a List when a List is needed.

:: operator

scala> **val list1 = 3 :: Nil**

list1: List[Int] = List(3)

**Merge Two LISTS:-**

Merge two lists using the ++, concat, or ::: methods.

scala> **val a = List(1,2,3)**

a: List[Int] = List(1, 2, 3)

scala> **val b = List(4,5,6)**

b: List[Int] = List(4, 5, 6)

scala> **val c = a ::: b**

c: List[Int] = List(1, 2, 3, 4, 5, 6)

**MAP:**

LinkedHashMap:- Maintains Insertion order,Mutable. The LinkedHashMap implements a mutable map using a hashtable.

collection.immutable.Map This is the default, general-purpose immutable map you get if you don’t

import anything.

collection.mutable.Map A mutable version of the basic map.

collection.mutable.LinkedHashMap All methods that traverse the elements will visit the elements in their insertion

order.

collection.immutable.ListMap

collection.mutable.ListMap Per the Scaladoc, “implements immutable maps using a list-based data

structure.” As shown in the examples, elements that are added are prepended to the head of the list.

collection.SortedMap Keys of the map are returned in sorted order. Therefore, all traversal methods

(such as foreach) return keys in that order.

Class or trait Description

collection.immutable.HashMap From the Scaladoc, “implements immutable maps using a hash trie.”

collection.mutable.ObservableMap From the Scaladoc: “This class is typically used as a mixin. It adds a

subscription mechanism to the Map class into which this abstract class is

mixed in.”

Class or trait Description

collection.mutable.MultiMap From the Scaladoc: “A trait for mutable maps with multiple values assigned

to a key.”

collection.mutable.SynchronizedMap From the Scaladoc: This trait “should be used as a mixin. It synchronizes

the map functions of the class into which it is mixed in.”

collection.immutable.TreeMap From the Scaladoc: “implements immutable maps using a tree.”

collection.mutable.WeakHashMap A wrapper around java.util.WeakHashMap, “a map entry is

removed if the key is no longer strongly referenced.”

READING A FILE:-

**import scala.io.Source**

**val** filename **=** "fileopen.scala"

**for** (line **<- Source**.fromFile(filename).getLines) {

println(line)

}

**val** lines **= Source**.fromFile("/Users/Al/.bash\_profile").getLines.toList

**val** lines **= Source**.fromFile("/Users/Al/.bash\_profile").getLines.toArray

Use this approach to get all of the lines from the file as one String:-

**val** fileContents **= Source**.fromFile(filename).getLines.mkString

**Properly closing the file**

To properly close the file, get a reference to the BufferedSource when opening the file, and manually close it when you’re finished with the file:

**val** bufferedSource **= Source**.fromFile("example.txt")

**for** (line **<-** bufferedSource.getLines) {

println(line.toUpperCase)

}

bufferedSource.close

writing TEXT files:-

**import java.io.\_**

**val** pw **= new PrintWriter**(**new File**("hello.txt" ))

pw.write("Hello, world")

pw.close

*// FileWriter*

**val** file **= new File**(canonicalFilename)

**val** bw **= new BufferedWriter**(**new FileWriter**(file))

bw.write(text)

bw.close()

Reading and Writing Binary Files :-

**import java.io.\_**

**object CopyBytes extends App** {

**var** in **= None: Option**[**FileInputStream**]

**var** out **= None: Option**[**FileOutputStream**]

**try** {

in **= Some**(**new FileInputStream**("/tmp/Test.class"))

out **= Some**(**new FileOutputStream**("/tmp/Test.class.copy"))

**var** c **=** 0

**while** ({c **=** in.get.read; c != −1}) {

out.get.write(c)

}

} **catch** {

**case** e**: IOException** => e.printStackTrace

} **finally** {

println("entered finally ...")

**if** (in.isDefined) in.get.close

**if** (out.isDefined) out.get.close

}

}

**Deleting a file in spark:**

**def** rm(file: *String*): Unit = {

// rm(new File(file))

**val** fileTemp = **new** File(file)

**if** (fileTemp.exists) {

fileTemp.delete()

import org.apache.commons.io.FileUtils for different file related operations

**How the function parameter is defined and argument**

private def runBasicParquetExample(spark: SparkSession): Unit = {

To read:-

InputStream ins = new FileInputStream(new File(headerPath));

BufferedReader rd = new BufferedReader(new InputStreamReader(ins));

To Write:-

FileOutputStream os = new FileOutputStream(new File(args[4]+".hql"))

**CREATING DATA FRAMES:-**

1. **Defining using case class:-**

def main(args: Array[String]) {

/\*\*

\* Create RDD and Apply Transformations

\*/

val fruits = sc.textFile("src/main/resources/fruits.txt")

.map(\_.split(","))

.map(frt => Fruits(frt(0).trim.toInt, frt(1), frt(2).trim.toInt))

.toDF()

/\*\*

\* Store the DataFrame Data in a Table

\*/

fruits.registerTempTable("fruits")

/\*\*

\* Select Query on DataFrame

\*/

val records = sqlContext.sql("SELECT \* FROM fruits")

/\*\*

\* To see the result data of allrecords DataFrame

\*/

records.show()

}

}

case class Fruits(id: Int, name: String, quantity: Int)

1. **Schema Defining programmatically:-**

object ProgrammaticallySchema {

val sc = SparkCommon.sparkContext

val schemaOptions = Map("header" -> "true", "inferSchema" -> "true")

//sc is an existing SparkContext.

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

def main(args: Array[String]) {

// Create an RDD

val fruit = sc.textFile("src/main/resources/fruits.txt")

// The schema is encoded in a string

val schemaString = "id name"

// Generate the schema based on the string of schema

val schema =

StructType(

schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))

schema.foreach(println)

// Convert records of the RDD (fruit) to Rows.

val rowRDD = fruit.map(\_.split(",")).map(p => Row(p(0), p(1).trim))

rowRDD.foreach(println)

// Apply the schema to the RDD.

val fruitDataFrame = sqlContext.createDataFrame(rowRDD, schema)

fruitDataFrame.foreach(println)

// Register the DataFrames as a table.

fruitDataFrame.registerTempTable("fruit")

/\*\*

\* SQL statements can be run by using the sql methods provided by sqlContext.

\*/

val results = sqlContext.sql("SELECT \* FROM fruit")

results.show()

}

}

**3.**

import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType }

import org.apache.spark.sql.Row

val schema = StructType(

StructField("k", StringType, true) ::

StructField("v", IntegerType, false) :: Nil)

// Spark < 2.0

// sqlContext.createDataFrame(sc.emptyRDD[Row], schema)

spark.createDataFrame(sc.emptyRDD[Row], schema)

1. Create Dataframe in One line:-

var input = sqlContext.createDataFrame(Seq(  
 (10L, "Joe Doe", 34),  
 (11L, "Jane Doe", 31),  
 (12L, "Alice Jones", 25)  
 )).toDF("id", "name", "age")

1. **val** rdd1 : RDD[Row]= sc.sparkContext.makeRDD(Seq(Row("Sumeet","Ajay","Rahul","Sachin","Mark")))

**var** datafr = sc.sqlContext.createDataFrame(rdd1, schema)

1. **val** claimsData1 = Seq(("PID1", "diag1", "diag2", 100, 200), ("PID1", "diag2", "diag3", 300, 600), ("PID1", "diag1", "diag5", 340, 680), ("PID2", "diag3", "diag4", 245, 490), ("PID2", "diag2", "diag1", 124, 248))

**val** claimRDD1 = sc.sparkContext.parallelize(claimsData1)

**val** claimRDDRow1 = claimRDD1.map(p => Row(p.\_1, p.\_2, p.\_3, p.\_4, p.\_5))

**val** claimRDD2DF1 = sc.sqlContext.createDataFrame(claimRDDRow1, Claim1)

1. val schema = StructType(Array(StructField("firstName", StringType, true), StructField("lastName", StringType, true), StructField("age", IntegerType, true)))

val rowRDD = person.map(\_.split(",")).map(p => org.apache.spark.sql.Row(p(0), p(1), p(2).toInt))

val personDF = sqlContext.createDataFrame(rowRDD, schema)

**Handling New line breaks from CSV**

.option("parserLib", "UNIVOCITY")

**Filter Using like condition:-**

filter(DDD\_DOllars("Product Name") like "%JEVTANA%").show()

**val** query = """SELECT "DDD\_DOllars.PRODUCT NAME" ,SRA2,"filter\_product\_list.Product Name" as b

FROM filter\_product\_list CROSS JOIN DDD\_DOllars

ON "DDD\_DOllars.PRODUCT NAME" LIKE CONCAT('%', "filter\_product\_list.Product Name" ,'%')"""

**Save DataFrames in a Text file**

df.select("year", "model").write()  
 .format("com.databricks.spark.csv")  
 .option("header", "true")  
 .option("codec", "org.apache.hadoop.io.compress.GzipCodec")  
 .save("newcars.csv");

**UDF in scala**

import org.apache.spark.sql.functions.udf

val coder: (Int => String) = (arg: Int) => {

if (arg < 300) "slow" else "high"

}

val sqlfunc = udf(coder)

carDataFrame.withColumn("First", sqlfunc(col("speed"))).show()

|  |  |
| --- | --- |
| Speed | First |
| 230 | Slow |
| 300 | High |
| 340 | high |

def indication\_map(str :String) : String = {

val a = str match {

case "FL" => "FL"

case "Waldenstrom" => "WM"

case "Chronic Lymphoid Leukemia (CLL)" => "CLL"

case "DLBCL" => "DLBCL"

case "Mantle Cell Lymphoma (MCL)" => "MCL"

case "Marginal Zone Lymphoma (MZL)" => "MZL"

}

return a

}

def Line\_map(str :String) : String = {

val a = str match {

case "2ND LINE +" => "2L+"

case "3RD LINE" => "3L"

case "3RD LINE +" => "3L+"

case "2ND LINE" => "2L"

case "1ST LINE" => "1L"

}

return a

}

val indication\_udf = udf(indication\_map \_)

val Line\_udf = udf(Line\_map \_)

var raw\_regimen\_part\_two = Util.loadTable(rsURL, schema+".RAW\_ONC\_IMBRUVICA\_MONTHLY\_REGIMEN\_PART\_TWO", tempS3Dir, rsDbName, username, password, sqlContext)

raw\_regimen\_part\_two = raw\_regimen\_part\_two.withColumn("indication",indication\_udf(trim(col("indication"))))

.withColumn("Line",Line\_udf(trim(col("line"))))

A <: B --Upper bound A must be a subtype of B.

A >: B -- Lower bound A must be a supertype of B. Not commonly used.

A <: Upper >: Lower -- Lower and upper bounds used together. The type A has both an upper and lower bound.

Array[T] = Invariant Used when elements in the container are mutable.

Example: Can only pass Array[String] to a method expecting Array[String].

Seq[+A] = Covariant Used when elements in the container are immutable. This makes the container more flexible.

Example: Can pass a Seq[String] to a method expected Seq[Any].

**Note V IMP -** Methods in Scala can be parameterized with both values and types. Like on the class level, value parameters are enclosed in a pair of parentheses, while type parameters are declared within a pair of brackets.

def dup[T](x: T, n: Int): List[T] = {

if (n == 0)

Nil

else

x :: dup(x, n - 1)

}

println(dup[Int](3, 4))

println(dup("three", 3))

**Scala Generics**class Stack[T] {

var elems: List[T] = Nil

def push(x: T) { elems = x :: elems }

def top: T = elems.head

def pop() { elems = elems.tail }

}

object GenericsTest extends App {

val stack = new Stack[Int]

stack.push(1)

stack.push('a')

println(stack.top)

stack.pop()

println(stack.top)

}

This means that if we have a stack of characters of type Stack[Char] then it cannot be used as an integer stack of type Stack[Int].  
To conclude, Stack[T] is only a subtype of Stack[S] if and only if S = T

**Sample learning operations**

import java.sql.Timestamp

import java.text.SimpleDateFormat

import java.util.Date

def getTimestamp(x:Any) :java.sql.Timestamp = {

val format = new SimpleDateFormat("MM/dd/yyyy' 'HH:mm:ss")

if (x.toString() == "")

return null

else {

val d = format.parse(x.toString());

val t = new Timestamp(d.getTime());

return t

}

}

def convert(row :org.apache.spark.sql.Row) :org.apache.spark.sql.Row = {

import org.apache.spark.sql.Row

val d1 = getTimestamp(row(3))

return Row(row(0),row(1),row(2),d1)

}

val strRdd = sc.textFile("hdfs://path/to/cvs-file")

val rowRdd: RDD[Row] = strRdd.map(\_.split('\t')).mapPartitions { iter =>

new Iterator[Row] {

val row = new GenericMutableRow(4)

var current: Array[String] = \_

def hasNext = iter.hasNext

def next() = {

current = iter.next()

row(0) = current(0)

row(1) = current(1)

row(2) = current(2)

val ts = getTimestamp(current(3))

if(ts != null) {

row.update(3, ts)

} else {

row.setNullAt(3)

}

row

}

}

}

**Use case 1:-**

{"employee":"Michale","Address":"NY"}

{"employee":"Michale","Address":"NJ"}

{"employee":"Sam","Address":"NY"}

{"employee":"Max","Address":"NJ"}

Spark Application

val df = sqlContext.read.json("sample.json")

// Printing the original Df

df.show()

//Defining the Schema for the aggregated DataFrame

val dataSchema = new StructType(

Array(

StructField("employee", StringType, nullable = true),

StructField("Address", ArrayType(StringType, containsNull = true), nullable = true)

)

)

// Converting the df to rdd and performing the groupBy operation

val aggregatedRdd: RDD[Row] = df.rdd.groupBy(r =>

r.getAs[String]("employee")

).map(row =>

// Mapping the Grouped Values to a new Row Object

Row(row.\_1, row.\_2.map(\_.getAs[String]("Address")).toArray)

)

// Creating a DataFrame from the aggregatedRdd with the defined Schema (dataSchema)

val aggregatedDf = sqlContext.createDataFrame(aggregatedRdd, dataSchema)

// Printing the aggregated Df

aggregatedDf.show()

Output :

+-------+--------+---+

|Address|employee|num|

+-------+--------+---+

| NY| Michale| 1|

| NJ| Michale| 2|

| NY| Sam| 3|

| NJ| Max| 4|

+-------+--------+---+

+--------+--------+

|employee| Address|

+--------+--------+

| Sam| [NY]|

| Michale|[NY, NJ]|

| Max| [NJ]|

+--------+--------+

**Use case 2:**

**Reducebykey and find the hisghest value**

import org.apache.spark.{SparkContext, SparkConf}

object Main extends App {

val data = List(

"Bob,123,USA,Pizza,Soda,,Blue",

"Bob,456,UK,Chocolate,Cheese,Soda,Green",

"Bob,12,USA,Chocolate,Pizza,Soda,Yellow",

"Mary,68,USA,Chips,Pasta,Chocolate,Blue")

val sparkConf = new SparkConf().setMaster("local").setAppName("example")

val sc = new SparkContext(sparkConf)

val lineRDD = sc.parallelize(data)

val pairedRDD = lineRDD.map { line =>

val fields = line.split(",")

(fields(0), List(fields(3), fields(4), fields(5)).filter(\_ != ""))

}.filter(\_.\_1 == "Bob")

/\*pairedRDD.collect().foreach(println)

(Bob,List(Pizza, Soda))

(Bob,List(Chocolate, Cheese, Soda))

(Bob,List(Chocolate, Pizza, Soda))

\*/

val flatPairsRDD = pairedRDD.flatMap {

case (name, foodList) => foodList.map(food => ((name, food), 1))

}

/\*flatPairsRDD.collect().foreach(println)

((Bob,Pizza),1)

((Bob,Soda),1)

((Bob,Chocolate),1)

((Bob,Cheese),1)

((Bob,Soda),1)

((Bob,Chocolate),1)

((Bob,Pizza),1)

((Bob,Soda),1)

\*/

val nameFoodSumRDD = flatPairsRDD.reduceByKey((a, b) => a + b)

/\*nameFoodSumRDD.collect().foreach(println)

((Bob,Cheese),1)

((Bob,Soda),3)

((Bob,Pizza),2)

((Bob,Chocolate),2)

\*/

val resultsRDD = nameFoodSumRDD.map{

case ((name, food), count) => (name, (food,count))

}.groupByKey.map{

case (name, foodCountList) => (name, foodCountList.toList.sortBy(\_.\_2).reverse.head)

}

resultsRDD.collect().foreach(println)

/\*

(Bob,(Soda,3))

\*/

sc.stop()

}

**Iteration of a dataframe**

**var** input = sqlContext.createDataFrame(Seq((10L, "Joe Doe", 34),(11L, "Jane Doe", 31),

(12L, "Alice Jones", 25))).toDF("id", "name", "age")

input.collect().map{

**case** Row(user\_id: Long, category\_id: *String*, rating: Integer) => (println(user\_id))}

**Important Dataset Example:-**

Datasets — Strongly-Typed DataFrames with Encoders

It is only with Datasets to have syntax and analysis checks at compile time (that was not possible using DataFrame, regular SQL queries or even RDDs).

1. def getDFRow(row: Row):DFRow = {

return DFRow(row.getLong(row.fieldIndex("item0")),

row.getString(row.fieldIndex("item1")),

row.getString(row.fieldIndex("item2")))

}

df.map(DFRow(\_))

.groupByKey(row => row.getLong(0))

.mapGroups((key, iterable) => println(key))

1. //val personRDD = sc.makeRDD(Seq(Person("A",10),Person("B",20)))
2. **import** sqlContext.implicits.\_

**val** personDF = sqlContext.read

.format("com.databricks.spark.csv").option("header", "true")

.load("C:\\Users\\sumeet.agrawal\\Downloads\\sample\_datamap.csv")

**val** ds:Dataset[**Person**] = personDF.as[**Person**]

**val** df = Seq((1,"asdf"),(2,"34234")).toDF("key", "value")

**val** ds1 = df.as[**KeyValue**]

}

**case** **class** **Person**(variable : *String* , email : *String*, Comments : *String*)  
**case** **class** **KeyValue**(key: Int, value: *String*)

1. **Create an Empty dataset**

val strings = spark.emptyDataset[String]

1. **import** sparkSession.implicits.\_

**val** one = sparkSession.createDataset(Seq(1))

1. scala> val one = Seq(1).toDS
2. one: org.apache.spark.sql.Dataset[Int] = [value: int]

**IMP: Example of scala types and asInstanceOf(Used for typecasting)**

def failFastCast[A: Manifest, T[A] <: Traversable[A]](as: T[A], any: Any) = {

val res = any.asInstanceOf[T[A]]

if (res.isEmpty) res

else {

manifest[A].newArray(1).update(0, res.head) // force exception on wrong type

res

}

}

scala> val x = List(1, 2, 3): Any

x: Any = List(1, 2, 3)

scala> failFastCast(List[Int](), x)

res22: List[Int] = List(1, 2, 3)

Example of Dynamically creating dataframe schmea:-

object App {

case class Person(name: String, age: Int)

def main(args: Array[String]) {

val sparkConf = new SparkConf().setAppName("Test").setMaster("local[4]")

val sc = new SparkContext(sparkConf)

val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

val input = sc.parallelize(Seq(Person("a", 1), Person("b", 2)))

val dataFrame = input.df

dataFrame.show()

// create the extended rows RDD

val rowRDD = dataFrame.rdd.map{

row =>

val blob = row(1).asInstanceOf[Int]

val newColumns: Seq[Any] = Seq(blob, blob \* 2, blob \* 3)

Row.fromSeq(row.toSeq.init ++ newColumns)

}

val schema = dataFrame.schema

// we know that the new columns are all integers

val newColumns = StructType{

Seq(new StructField("1", IntegerType), new StructField("2", IntegerType), new StructField("3", IntegerType))

}

val newSchema = StructType(schema.init ++ newColumns)

val newDataFrame = sqlContext.createDataFrame(rowRDD, newSchema)

newDataFrame.show()

}

}

Note:- Init method get the first column(schema and data)

Output:-

+----+---+

|name|age|

+----+---+

| a| 1|

| b| 2|

+----+---+

+----+---+---+---+

|name| 1| 2| 3|

+----+---+---+---+

| a| 1| 2| 3|

| b| 2| 4| 6|

+----+---+---+---+

Union Two Dataframes:-

val firstDF = spark.range(3).toDF("myCol")

val newRow = Seq(20)

val appended = firstDF.union(newRow.toDF())

display(appended)

Create DataFrames from RDD….

val people = sc.textFile("examples/src/main/resources/people.txt").map(\_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()

XML handling

<http://alvinalexander.com/scala/xml-parsing-xpath-extract-xml-tag-attributes>

<https://dzone.com/articles/basic-xml-processing-scala>

Reading a XML file in scala

Import scala.xml.XML

val test = XML.loadFile("C:\\Users\\sumeet.agrawal\\Desktop\\test.xml")

To access the tag

val a = test \\ "catalog"

val a = test \\ "catalog" \\ "color\_swatch" \\ "@image"

It will show all the labels of image under color\_swatch tag

The xml are of type - scala.xml.Elem

1. Extracting Text:-

scala> <a>Sounds <tag/> good</a>.text

res8: String = Sounds good

**Loggers and best practices**

import org.apache.log4j.Logger

var logger = Logger.getLogger(this.getClass())

if (arg.length < 2) {

logger.error("=> wrong parameters number")

System.err.println("Usage: MainExample <path-to-files> <output-path>")

System.exit(1)

}

logger.info("=> jobName \"" + jobName + "\"")  
logger.info("=> pathToFiles \"" + pathToFiles + "\"")  
val today = Calendar.getInstance().getTime  
println(s"Script executed on $today")

**SPARK SQL functions:-**

1. GroupedData class provides a number of methods for the most common functions, including count, max, min, mean and sum.

val df = sc.parallelize(Seq(

(1.0, 0.3, 1.0), (1.0, 0.5, 0.0),

(-1.0, 0.6, 0.5), (-1.0, 5.6, 0.2))

).toDF("col1", "col2", "col3")

df.groupBy($"col1").min().show

// +----+---------+---------+---------+

// |col1|min(col1)|min(col2)|min(col3)|

// +----+---------+---------+---------+

// | 1.0| 1.0| 0.3| 0.0|

// |-1.0| -1.0| 0.6| 0.2|

// +----+---------+---------+---------+

Optionally you can pass a list of columns which should be aggregated

df.groupBy("col1").sum("col2", "col3")

You can also pass dictionary / map with columns a the keys and functions as the values:

val exprs = df.columns.map((\_ -> "mean")).toMap

df.groupBy($"col1").agg(exprs).show()

// +----+---------+------------------+---------+

// |col1|avg(col1)| avg(col2)|avg(col3)|

// +----+---------+------------------+---------+

// | 1.0| 1.0| 0.4| 0.5|

// |-1.0| -1.0|3.0999999999999996| 0.35|

// +----+---------+------------------+---------+

1. Finally you can use varargs:

import org.apache.spark.sql.functions.sum

val exprs = df.columns.map(sum(\_))

df.groupBy($"col1").agg(exprs.head, exprs.tail: \_\*)