Spark Programming: Using Scala

Scala

Scala has many other nice features:

- Traits.
- Implicit conversions.
- Pattern Matching.
- XML literals,
- Parser combinators, ...

Scala

- Virtual stack machine that executes Java bytecode (.class files).
- Bytecode is hardware/system independent.
- Provides layer of protection to host machine.
- Built-in garbage collection.
- Just-in-time compilation (bytecode $\rightarrow$ machine code).
  Often as fast as C.
- Many languages:
  - JVM language: Java, Scala, Clojure, Groovy.
  - Other JVM compilers: Python, Ruby, C, Common Lisp, $\cdots$

Scala Shell

```
$ scala
Welcome to Scala version 2.11.2.
Type in expressions to have them evaluated.
Type :help for more information.

scala>

scala> 40 + 2
res0: Int = 42

scala> res0 * 2
res1: Int = 84
```

Running Scala Script

A script is sequence of statements in a file, interpreted sequentially.

hello.scala:

```
println("Hello, World!")
```

```
$ scala hello.scala
Hello, World!
```

A Standalone Scala Application

A standalone Scala application needs to have a singleton object with a method called main.

- main method takes an input of type Array[String] and does not return any value.
- It is the entry point of a Scala application.
- The singleton object containing the main method can be named anything.

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
      println("Hello World!")
    }
}
```

Compiling and Running Scala

`Test.scala` → scalac → `Test.class` → scala

Hello.scala:

```
object Hello {
    def main(args: Array[String]) = println("Hello, World!");
}
```

```
$ scalac Hello.scala
$ ls
Hello$.class    Hello.class    Hello.scala
$ scala Hello
Hello, World!
```

- Can use `fsc` (compile server) for faster compilation when frequently compiling.
- `scala` is actually a bash script calling `java` (the JVM).

Basic Scala Variables

Variable Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello

scala> var msg2 = " World"
msg2: String = " World"

scala> msg = "Hi"
msg: String = Hi

scala> msg + msg2
res0: String = Hello World

scala> msg = 3
<console>:8: error: type mismatch;
  found   : Int(3)
  required: String
       msg = 3
             ^
```

variables vs values

vars can be reassigned.

```
scala> var number  = 2
number: Int = 2

scala> var number  = number + 3
number: Int = 5
```

vals cannot be reassigned once defined.

```
scala> val number  = 42
number: Int = 42

scala> number = 23
<console>:8: error: reassignment to val
       number = 23
              ^
```

When in doubt, try to use vals for readability and a more functional programming style.

Integer types

The following two statements are equivalent.

```
val y: Int = 10;
val y = 10
```

Floating point types

`Float`	32-bit single precision float
`Double`	64-bit single precision float

```
scala> val y = 42E-4; val x = 32.0
y: Double = 0.0042
x: Double = 32.0

scala> x * y
res8: Double = 0.134
```

Strings and Symbols

String	a sequence of Chars.
Symbol	an 'interned' String.

```
scala> val hello = "Hello World"
hello: String = Hello World

scala> val color = 'hearts; val value= 'queen
color: Symbol = 'hearts
value: Symbol = 'queen
```

- String is simply an alias for java.lang.String.
- Symbols can often be used in place of enums or global constants.

Booleans and comparisons

Boolean	true or false

```
scala> val x = 5
x: Int = 5

scala> x < 3+2
res3: Boolean = false

scala> val y = "Hello"
y: String = Hello

scala> y == "Hello"
res4: Boolean = true
```

- == tests for value equality, not reference equality.

Conditionals with if

if (*condition*) *expression* [else if (*condition*) *expression*] [else *expression*]

- condition is any expression that returns a `Boolean`.

```
val a = 7
if (a % 2 == 0) {
    println("a is even")
} else if (a % 3 == 0) {
    println("a is a multiple of 3")
} else println("none of the above")
```

Everything is an Expression

- There are no statements in Scala.
- Every block of code returns a value. This value can be `Unit`.
- Compound expressions (with { . . . } return result of last expression).
- Type of expression automatically inferred (or can make it explicit).

```
scala> val z : Int = {val x = 4; val y = 2; x / y}
z: Int = 2
scala> val a = {val b = 4; }
a: Unit = ()
```

Conditionals are also Expressions

```
scala> val a = -7;
a: Int = -7

scala> val abs_a = if (a > 0) a else -a
abs_a: Int = 7
```

- What happens if return type is unknown (e.g. missing else)?

```
scala> val z : Boolean = if (42 > 23) true
<console>:7: error: type mismatch;
 found   : Unit
 required: Boolean
       val z : Boolean = if (42 > 23) true
                         ^

scala> val z =  if (42 > 23) true
z: AnyVal = true
```

Loops with while and dowhile

```
scala> var x = 1
scala> while (x <= 5) {println(x); x+= 1}
1
2
3
4
5
scala> x
res1: Int = 6

scala> do {x+=1; println(x);} while (x<=5)
7
```

- While loops usually indicate imperative programing style (manipulate the content of some variable in each step).
- Result type of while is `Unit`.

for loops

```
scala> for (y<-List(1,2,3)) {println(y)}
1
2
3
```

- used in this way the result of a `for` expression is `Unit`

Range objects

```
scala> 1 to 10    // or 1.to(10)
res1: scala.collection.immutable.Range.Inclusive =
    Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 10 to (0,-2)
res2: scala.collection.immutable.Range.Inclusive =
    Range(10, 8, 6, 4, 2, 0)

scala> for (i <- 1 to 3) println(i*2);
2
4
6
```

for as an expression

`for (`*seq*`) yield` *expression*

- *seq* contains at least one generator of the form x <- sequence
- *seq* can contain definitions and filters.

```
scala> for (x <- List(1,2,3)) yield x*2
res8: List[Int] = List(2, 4, 6)

scala> for { x <- 1 to 7 // generator
             y = x % 2;     // definition
             if (y == 0)    // filter
           } yield {
              println(x)
              x
           }
2
4
6
res1: scala.collection.immutable.IndexedSeq[Int] =
    Vector(2, 4, 6)
```

Defining Functions

$$\text{def } name(param_1 : type_1, \cdots param_n : type_n) : return\_type = body$$

```
scala> def max(x: Int, y: Int) : Int = {
            if (x > y) x
            else y
        }
max: (x: Int, y: Int)Int
```

Return value of a function is the result of the body expression ({} are optional in this case).

Defining Functions

```
def add(firstInput: Int, secondInput: Int): Int = {
  val sum = firstInput + secondInput
  return sum
}
```

Scala allows a concise version of the same function, as shown next.

```
def add(firstInput: Int, secondInput: Int) = firstInput + secondInput
```

The second version does the exact same thing as the first version.
The type of the returned data is omitted since the compiler can infer it from the code.

Calling functions

```
scala> max(2,3)
res1: Int = 3
```

If a function does not take parameters, do not use parantheses

```
scala> def greet() = println("Hello")
greet: ()Unit

scala> greet
Hello
```

Higher-Order Methods

A method that takes a function as an input parameter is called a *higher-order method*.

- a high-order function is a function that takes another function as input.
- helps reduce code duplication.
- helps you write concise code.

The following example shows a simple higher-order function.

```
def encode(n: Int, f: (Int) => Long): Long = {
  val x = n * 10
  f(x)
}
```

Function Literals

A function literal is an unnamed or anonymous function in source code and defined with input parameters in parenthesis, followed by a right arrow and the body of the function.

- can be used in an application just like a string literal.
- can be passed as an input to a higher-order method or function.
- can also be assigned to a variable.
- is enclosed in optional curly braces.

An example is:

```
(x: Int) => {
              x + 100
            }

(x: Int) => x + 100
```

Can be used higher-order functions:

```
val code = encode(10, (x: Int) => x + 100)
```

Closures

A closure is a function literal that uses a non-local non-parameter variable captured from its environment.
Sometimes people use the terms function literal and closure interchangeably, but technically, they are not the same.

The following code shows an example of a closure:

```
def encodeWithSeed(num: Int, seed: Int): Long = {

  def encode(x: Int, func: (Int) => Int): Long = {
    val y = x + 1000
    func(y)
   }

   val result = encode(num, (n: Int) => (n * seed))
   result
}
```

Classes

A class is an object-oriented programming concept. It provides a higher-level programming abstraction

- A class in Scala is similar to that in other object-oriented languages.
- An object is an instance of a class.
- A class is defined in source code, whereas an object exists at runtime
- A class is defined using the keyword class

```
class Car(mk: String, ml: String, cr: String) {
  val make = mk
  val model = ml
  var color = cr

  def repaint(newColor: String) = {
    color = newColor
  }
}
```

An instance of a class is created using the keyword new.

```
val mustang = new Car("Ford", "Mustang", "Red")
val corvette = new Car("GM", "Corvette", "Black")
```

Singletons

In object-oriented programming, it is to define a class that can be instantiated only once.

- A class that can be instantiated only once is called a singleton.
- Scala provides the keyword object for defining a singleton class.

```
object DatabaseConnection {
  def open(name: String): Int = {
    ...
  }

  def read (streamId: Int): Array[Byte] = {
    ...
  }

  def close (): Unit = {
    ...
  }

}
```

All types are classes

- All values are instances of some class.
- This is even true for basic numeric types (unlike Java).
- Can call methods on instances of these classes.

```
scala> 42.toString
res1: String = 42
```

Case class

A **case class** is a class with a case modifier.

```
case class Message(from: String, to: String, content: String)
```

Scala provides a few syntactic conveniences to a case class:

- Can create an instance of a case class without the keyword **new**.

```
val request = Message("harry", "sam", "fight")
```

- all input parameters specified in the definition of a case class implicitly get a **val** prefix. Scala treats the case class Message as if it was defined.

```
class Message(val from: String, val to: String, val content: String)
```

Pattern Matching

Pattern matching is a Scala concept that looks similar to a **switch** statement in other languages.

- it is a more powerful tool than a switch statement.
- Instead of the keyword switch, Scala uses the keyword **match**
- Unlike switch, a break statement is not required after the code for each case.
- Can be used as a replacement for a multi-level if-else statement

```
def colorToNumber(color: String): Int => {
  val num = color match {
              case "Red" => 1
              case "Blue" => 2
              case "Green" => 3
              case "Yellow" => 4
              case _ => 0
            }
  num
}
```

Pattern Matching

Code on the right-hand side of each right arrow is an expression returning a value. Therefore, a pattern-matching statement itself is an expression returning a value.

```
def f(x: Int, y: Int, operator: String): Double = {
  operator match {
    case "+" => x + y
    case "-" => x - y
    case "*" => x * y
    case "/" => x / y.toDouble
  }
}

val sum = f(10,20, "+")
val product = f(10, 20, "*")
```

All operators are methods

```
scala> x = 3
x: Int = 3

scala> x.+(2)
res1: Int = 5

scala> x.==(5)
res2: Boolean = true

scala> "fortunate".contains("tuna")
res3: Boolean = true

scala> "fortunate" contains "tuna"
res4: Boolean = true
```

Traits

A trait represents an interface supported by a hierarchy of related classes.

- Scala traits are similar to Java interfaces.
- Unlike Java, a Scala trait can include implementation of a method.
- Trait can include fields.
- A class can reuse the fields and methods implemented in a trait.

The following code shows an example:

```
trait Shape {
   def area(): Int
}

class Square(length: Int) extends Shape {
   def area = length * length
}

class Rectangle(length: Int, width: Int) extends Shape {
   def area = length * width
}

val square = new Square(10)
val area = square.area
```

Tuples

A tuple is a container for storing two or more elements of different types.
It is immutable; it cannot be modified after it has been created.

A tuple is useful in situations where you want to group non-related elements.
If the elements are of the same type, you can use a collection, such as an array
An element in a tuple has a one-based index.

```
val twoElements = ("10", true)
val threeElements =  ("10", "harry", true)

val first = threeElements._1
val second = threeElements._2
val third = threeElements._3
```

Option Type

An Option is a data type that indicates the presence or absence of some data.

- It represents optional values.
- It can be an instance of either a case class called Some or singleton object called None.
- An instance of Some can store data of any type.
- The None object represents absence of data.

```
def colorCode(color: String): Option[Int] = {
  color match {
    case "red" => Some(1)
    case "blue" => Some(2)
    case "green" => Some(3)
    case _ => None
  }
}

val code = colorCode("orange")
code match {
  case Some(c) => println("code for orange is: " + c)
  case None => println("code not defined for orange")
}
```

Array

An Array is an indexed sequence of elements. All the elements in an array are of the same type

- It is a mutable data structure; you can update an element in an array.
- cannot add an element after it has been created. It has a fixed length.
- Elements have a zero-based index.
- you specify its index in parenthesis

```
val arr = Array(10, 20, 30, 40)
arr(0) = 50
val first = arr(0)
```

Lists

A List is a linear sequence of elements of the same type.

- It is a recursive data structure, unlike an array, itis a flat data structure.
- it is an immutable data structure; it cannot be modified after it has been created.

```
val xs = List(10,20,30,40)
val ys = (1 to 100).toList
val zs = someArray.toList
```

Basic operations are:

- Fetching the first element. List provides a method named **head**.
- Fetching all the elements except the first element. List has a method named **tail**.
- Checking whether a list is empty. List has a method named **isEmpty**.

Vector

The Vector class is a hybrid of the List and Array classes.

- combines the performance characteristics of both Array and List.
- provides constant-time indexed access and linear access.
- allows both fast random access and fast functional updates.

```
val v1 = Vector(0, 10, 20, 30, 40)
val v2 = v1 :+ 50
val v3 = v2 :+ 60
val v4 = v3(4)
val v5 = v3(5)
```

Map

Map is a collection of key-value pairs. In other languages, it known as a dictionary, associative array, or hash map

- it is an efficient data structure for looking up a value by its key.
- It should not be confused with the map in Hadoop MapReduce.

```
val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
val indiaCapital = capitals("India")
```

Higher-Order Methods on map

The real power of Scala collections comes from their higher-order methods. A higher-order method takes a function as its input parameter.

The map method of a Scala collection

- applies its input function to all the elements and returns another collection.
- returned collection has the exact same number of elements
- but returned collection need not be of the same type as the original.

```
val xs = List(1, 2, 3, 4)
val ys = xs.map((x: Int) => x * 10.0)
```

xs is of type *List[Int]* but ys is of type *List[Double]*

Higher-Order Methods on map

If a function takes a single argument, opening and closing parentheses can be replaced with opening and closing curly braces, respectively.
The two statements shown next are equivalent.

```
val ys = xs.map((x: Int) => x * 10.0)
val ys = xs.map{(x: Int) => x * 10.0}
```

Scala allows calling any method using operator notation. For readability, it can also be written as follows. Scala can infer the type of the parameter passed.

```
val ys = xs map {(x: Int) => x * 10.0}
val ys = xs map {x => x * 10.0}
```

Higher-Order Methods on map

If an input to a function literal is used only once in its body, the right arrow and left-hand side of the right arrow can be dropped. You can write just the body of the function literal. The two statements shown next are equivalent.

```
val ys = xs map {x => x * 10.0}
val ys = xs map {_ * 10.0}
```

The underscore character represents an input passed to the map method. This can be read as multiplying each element in the collection xs by 10.

```
val ys = xs.map((x: Int) => x * 10.0)
val ys = xs map {_ * 10.0}
```

flatMap

The **flatMap** method of a Scala collection is similar to map.

- It takes a function as input, applies it to each element in a collection, and returns another collection.
- function passed to flatMap generates a collection for each element in the original collection.
- The flatMap method returns a flattened collection.

```
val line = "Scala is fun"
val SingleSpace = " "
val words = line.split(SingleSpace)
val arrayOfChars = words flatMap {_.toList}
```

- The toList method creates a list of all the elements in the collection.
- It is a useful method for converting a string, an array, or any other collection type to a list.

filter

The filter method applies a predicate to each element in a collection and returns another collection consisting of only those elements for which the predicate returned true.

- A predicate is function that returns a Boolean value.
- It returns either true or false.

```
val xs = (1 to 100).toList
val even = xs filter {_ %2 == 0}
```

foreach

The foreach method of a Scala collection calls its input function on each element of the collection, but does not return anything.

- It is similar to the map method.
- The only difference between the two methods is that map returns a collection and foreach does not return anything.
- It is a rare method that is used for its side effects.

```
val words = "Scala is fun".split(" ")
words.foreach(println)
```

reduce

The reduce method returns a single value for a given collection.

- The input function to the reduce method takes two inputs at a time and returns one value.
- Essentially, the input function is a binary operator that must be both associative and commutative.

```
val xs = List(2, 4, 6, 8, 10)
val sum  = xs reduce {(x,y) => x + y}
val product  = xs reduce {(x,y) => x * y}
val max = xs reduce {(x,y) => if (x > y) x else y}
val min = xs reduce {(x,y) => if (x < y) x else y}
```

```
Val words = "Scala is fun" split(" ")
val longestWord = words reduce {(w1, w2) => if(w1.length > w2.length) w1 else w2}
```