**Big Data Programming Languages**

* Each one of Python, Java and Scala can be used for Big data applications. Each of these has its own strengths and weaknesses.
* The winners are those that not only can scale-up, but also can be integrated with more Big Data technologies

**Apache Spark**

* Fast and scalable cluster computing engine that built on Java Virtual Machine (JVM)
* Supports Apache Hadoop MapReduce (MR) programming
* Integrates with Hadoop Distributed File System (HDFS)
* Interfaced with Java, Python, and Scala
* Scala is Spark’s native programming language
* Python requires translation by its interpreter and a socket connection to the JVM which can potentially slow down the processing
* Java does not suffer from such overhead although it is verbose and requires more code writing effort
* Scala is the native Spark programming languages with full Spark API support. Scala runs on JVM and integrates with Java.
* Scala is less verbose than Java for fast development
* Scala is a pure object-oriented language in the sense that every value is an object
* Scala is also a functional language in the sense that every function is a value therefore every function is an object
* Scala is statically typed so types are inferred

**Alphanumeric Identifiers**

* Composed of letters, underscores, and digits, beginning with a letter or underscore
* Examples: abc123, myVar, myVal, myMethod, MyClass, MyObject, Pi, E

**Operator Identifiers**

* Composed of operator characters (most punctuation marks)
* May not include:  letters or digits, or ( ) [ ] { } ' " \_ . , ; , `
* Examples:  +  =>  <?>  :::

**Mixed Identifiers**

* Composed of an alphanumeric identifier, an underscore, and an operator identifier.
* Examples:  unary\_+ , myVar\_=

**Literal Identifiers**

* Composed of any arbitrary string enclosed in backticks (the ` character).
* Examples:  `Hello, World!`, `class`

**Using Interactive Scala with REPL**

mhan0002@bigvm:~/.../week1$ scala

scala> println("Hello Everyone!")

Hello Everyone!

**Using a Shell Script**

1. #!/usr/bin/env scala
2. object HelloWorld extends App {
3. println("Hello, Scala!")
4. }
5. HelloWorld.main(args)

* Edit file : nano HelloWorld.sh and save
* To give the file execution permission : chmod 755 HelloWorld.sh -
* To run script : ./HelloWorld.sh

**Scala Application**

* Edit file : nano HelloWorld.scala and save

object HelloApp extends App {

println("Hello, Scala!")

}

* File name (HelloWorld.scala) does not have to be the same as application name (HelloWorld)
* The App trait wraps code in a main method
* Compile scala file : scalac HelloWorld.scala
* To run application : scala HelloApp

**Scala Program**

Edit file : nano HelloWithClass.scala

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()

}

}

* File name (HelloWithClass.scala) does not have to be the same as application name (HelloObjectPlusClass)
* Compile scala file : scalac HelloWithClass.scala
* To run program : scala HelloObjectPlusClass

mhan0002@bigvm:~/.../week1$ scala HelloObjectPlusClass

In object

In main

In class

**Scala Data Types**

* Scala has no primitives, only objects
* Any
  + **AnyVal**
    - Boolean, Char, Byte, Short, Int, Long, Float, Double
    - Unit (has only a single value, (); returned by functions)
  + **AnyRef** (corresponds to Object in Java)
    - ScalaObject
      * All scala.\* reference types, including Array and List
  + **Null** (bottom of all AnyRef objects)
* Nothing (bottom of Any)

**Data Type Description**

* Byte - 8 bit signed value (-128 to 127)
* Short - 16 bit signed value (-32768 to 32767)
* Int - 32 bit signed value (-2147483648 to 2147483647)
* Long - 64 bit signed value (-9223372036854775808 to 9223372036854775807)
* Float - 32 bit IEEE 754 single-precision float
* Double - 64 bit IEEE 754 double-precision float
* Char - 16 bit unsigned Unicode character (U+0000 to U+FFFF)
* String - A sequence of Chars
* Boolean - literal true or the literal false
* Unit - corresponds to no value, has only one value () returned by functions
* Null - null or empty reference
* Nothing - The subtype of every other type; includes no values
* Any - The supertype of any type; any object is of type Any
* AnyRef - The supertype of any reference type
* All types start with capital letters and appear after the : symbol
  + var x = 3.14
  + var y: Double = 3.14

**Var V Val**

var : value may change or be variable, val : value is constant

Example : var myVar : String = "Foo"

* myVar is declared using the keyword **var**. It is a variable that **can change** value and this is called a **mutable** variable

Example : val myVal : String = "Foo"

* myVal is declared using the keyword **val**. This means that it is a variable that **can not be changed** and this is called an **immutable** variable

var str1: String = "Hi!"

val str2: String = "Bye!" \\ constant declaration

str1 = "Hello!"

str2 = "Good Bye!" \\ this will throw an error, values can only be assigned to var variables

**Declarations**

If given an initial value in the declaration, the variable's type is inferred and need not be explicitly stated (but it may be). If explicitly stated, the type comes after the variable and a colon, for example, var q: Boolean = true.

The types of variables must be declared:

* When the variable is the formal parameter of a method
* When the variable is declared but not initialized within a class

Within a method, variables must be given an initial value. Variables within a class (and not within a method) may optionally be given an initial value.

When initial values are not given, new is required: val ary1 = new Array[Int](5) and default values of zero, false, or null are used.   
When initial values are given, new is not allowed: val ary2 = Array(3, 1, 4, 1, 6)

The type of a function is written with the double-headed arrow =>

## **Numbers**

* Char, Byte, Short, Int, Long, Float, Double are numeric types
* Operations on numbers include + (addition), - (subtraction or negation), \* (multiplication), / (division), and % (modulus)
* Operations on integers also include << (left shift), >> (right shift with sign extension), >>> (right shift with zero fill), & (bitwise and), | (bitwise or), and ^ (bitwise exclusive or)

**Strings**

* A String may be enclosed in double quotes, "...", or in triple double quotes, """...""". The latter is a raw string (that is, \ does not "escape" characters) and may contain newlines

**Data Collections** – arrays, lists, sets, tuples

**Arrays**

* A collection of values under one name
* Order of the array elements is preserved by Scala
* Scala arrays can contain duplicates
* Arrays are mutable, so we can easily change their elements
* Arrays indices start from zero, and all elements must have the same type
* The basic operations on arrays are getting a value from it, storing a value in it, and getting its length. Note the use of () instead of []
* array(n) returns the n-th value (0-based) in the array
* array(n) = value puts the value in the n-th location (0-based) of the array
* array.length returns the number of elements in the array
* Array functions include range() , concat(), fill(), empty() and repeat()

// Create and array of strings with length 10

var anArray : Array[String] = new Array[String](10)

// Does the same thing:

var anotherArray = new Array[String](10)

// Create an array of Int by initialization:

var someNumbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

// And now, a two dimensional array of size 10X5:

import Array.\_

val nRow = 10

val nCol = 5

var myMatrix = ofDim[Int](nRow,nCol)

// Create an array of even and odd numbers between 1 and 1000

var even = Array.range(2,1001,2)

var odd = Array.range(1,1000,2)

// Concatenate both arrays

var allArray = Array.concat(even,odd)

// Fill array with values

var hello5 = Array.fill(5)(“hello”)

// Empty array

var empty = Array.empty

**Lists**

* Lists are immutable so we cannot change their elements, the opposite to arrays
* List indices start from zero, and all elements must have the same type
* Lists preserve the order of their elements
* The type of the empty list is List[Nothing]
* list(n) returns the n-th value (0-based) in the array
* head(list) accesses the first element of a list
* tail(list) accesses all other elements
* List functions include +, ::, :::, contains(), drop(), filter(), foreach(), last, min, max, and reverse

// Create an empty String List

val aList : List[String] = List[String]()

// Does the same thing:

val anotherList = List[String]()

// Create a list of Int by initialization:

val someNumbers = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

// Recursively create a list of numbers, :: puts the value as the new head of the list.

val someNumbers = 1 :: (2 :: (3 :: (4 :: (5 :: (1 :: (2 :: (3 :: (4 :: (5 :: Nil)))))))))

//head and tail of list

val head = someNumbers.head // 1

val tail = someNumbers.tail // List[2,3,4,5,1,2,3,4,5]

//check last element on list

someNumbers.last // = 5

//concatenate two lists using :::

head ::: tail // [1,2,3,4,5,1,2,3,4,5]

//find minimum and maximum values on list

val minlist = someNumbers.min

val maxlist = someNumbers.max

//reverse list

val revlist = someNumbers.reverse

//drop 3 numbers from list

val drop3 = someNumbers.drop(3) // List[4,5,1,2,3,4,5]

//check if list contains a value

val check6 = someNumbers.contains(6) // false

// define sqr function

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

//apply sqr to each list element

val sqrlist = someNumbers.foreach(sqr) // returns no values!

val sqrlist = someNumbers.map(sqr) // List[1,4,9,16,25,1,4,9,16,25]

//filter all elements >= 3

val list\_ge\_3 = someNumbers.filter(\_>=3) // List[3,4,5,3,4,5]

//zip two lists together

val ziplist = someNumbers.zip(someNumbers) // List((1,1),(2,2),(3,3) … (5,5))

**Sets**

* Sets do not preserve the order of their elements after any operations
* Sets are immutable, like lists
* Sets cannot have duplicate elements
* The methods head(), tail(), and isEmpty can be applied on sets as well as lists
* + adds an element to the set
* - excludes an element from set
* ++concatenates two sets
* & returns the intersection of two sets (common elements)
* &~returns the differences of two sets (uncommon elements)

val set1 = Set(5,10,15,20,25,30)

val set2 = Set(0,10,20,30,40)

val set3 = set1 ++ set2 // Set(0,5,10,15,25,20,40,30,15)

val set4 = set1 & set2 // Set(10,20,30)

val set5 = set1 &~ set2 // Set(5,15,25)

val set6 = set2 &~ set1 // Set(0,40)

val set7 = set1 – 25 // Set(5,10,20,30,15)

val set8 = set1 + 50 // Set(5,10,25,20, 50,30,15)

**Tuples**

* A simple collection of elements with different types
* It differs from other kinds of Collections such as Arrays and Lists in that it can hold objects with different types
* Tuples are immutable, like lists and sets
* If you want to make a tuple containing 3 elements you use Tuple3 data type
* To access the ith element of a tuple, we can use \_i method. The index is 1 based

// Create a tuple

val tuple1 = new Tuple2("port", 80)

// Does the same thing

val tuple1 = ("port", 80)

println(tuple1.\_1) // “port”

println(tuple1.\_2) // 80

**Map**

* A collection of key-value pairs (very similar to Python dictionary)
* A value can be retrieved based on its key, not its index in the collection

// Create a color map

val colorMap = Map("red" -> "FF0000", "green" -> "00FF00", "blue" -> "0000FF")

println(colorMap.get(“red”)) // FF00000

**Functional Combinators**

**map -** evaluates a function over each element of a list and returns a list (the results) with the same number of elements

// Define a function that returns square of a number

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

val numList = List.range(1, 10)

val sqrList = numList.map(sqr) // List(1,4,9,16,25,36,49,64,81)

val doublelist = numList.map(\_\*2) //List(2,4,6,8,10,12,14,16,18)

**filter** - removes any elements where the passed function returns a **false**

// Define a function

def isPositive(x: Int): Boolean = x > 0

// Create a list of Int numbers

val numList = List.range(-5, 5)

// Apply the function on all list members

numList.filter(isPositive) // List(1,2,3,4,5)

**partition** - splits a list based on where it falls with respect to a **predicate** function

numList.partition(isPositive) // (List(1, 2, 3, 4, 5),List(-5, -4, -3, -2, -1, 0))

**find** - returns the first element of a collection (e.g. List or Map) that matches a predicate function

numList.find(isPositive) // List(1)

**drop(i) -** drops the first i elements of a list

// A list of 10 consecutive integers starting from 1 and ending at 10.

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

// Dropping the first 5 elements of the "numbers" list:

numbers.drop(5) // List(6, 7, 8, 9, 10)

**dropWhile(condition**) – drops the elements that match the condition. Applied to each element, from the very first element till the condition is false

// A list of 10 consecutive integers starting from 1 and ending at 10.

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

// Dropping the "odd" elements of the "numbers" list:

numbers.dropWhile(\_ % 2 != 0) // List(2, 3, 4, 5, 6, 7, 8, 9, 10)

**reduceLeft**, **reduceRight** - apply a given function to all elements of a list, but in pairs. **reduceLeft** starts from the front of the list forwards, **reduceRight** starts from the end of the list backwards

// Define a function

def sum (x:Int, y:Int): Int = x + y

// Create a list of Int numbers

val numList = List.range(1, 6)

// Apply the reduce function

numList.reduceLeft(sum) // 1+2+3+4+5 = 15

numList.reduceRight(sum) // 5+4+3+2+1 = 15

// Use (\_-\_) as the function to subtract

numList.reduceLeft(\_-\_) // 1-2-3-4-5 = -13

numList.reduceRight(\_-\_) // 5-4-3-2-1 = -5

## **foldLeft** and **foldRight** - these functions perform very similar to **reduceLeft** and **reduceRight**, but they accept another input and use it as the initial value of the accumulator

numList.foldLeft(100)(\_+\_) // 100+1+2+3+4+5 = 115

## **flatten** - collapses one level of nested structure. Therefore, it converts a N-order nested structure to (N-1)-order structure

// apply "flatten" on a List of Lists

val flatlist = List(List(1, 2), List(3, 4)).flatten

// the result is just one outer List as the nested (internal) Lists have been flattened.

println(flatlist) // List(1, 2, 3, 4)

**foreach**

val numList = List.range(1,6)

numList.foreach(x => println(x\*x)) // List(1,4,9,16,25)

val anotherList = numlist.map(x=>x\*2) // double it!

**Extended Data types**

class point(val x:Int, val y:Int)

{

def move(dx: Int, dy: Int): point = {

return new point(x + dx, y + dy)

}

override def toString(): String = "(" + x.toString() + "," + y.toString() + ")"

}

* **point** class has two properties to store x and y values, both are **val** so immutable class. Mutable data types like **var** are discouraged, since they are harder to be used in parallel computing for fast executions
* method **move** is used to move the current point to a new position
* All classes inherit from the base class **Object**. To modify the base class **toString** method, the keyword **override** must be used
* The **return** keyword is optional if the last line of the function holds the return value
* Parenthesis for functions without an argument can be omitted, toString() can be changed to toString
* A modified version for point class is listed below

class point(val x:Int, val y:Int){

def move(dx: Int, dy: Int) = new point(x+dx,y+dy)

override def toString= "(" + x.toString + "," + y.toString + ")"

}