CSC 735 – Data Analytics

Introduction to Scala

Functions

Functions

- Write a function to return maximum of two inputs

```
def maximum(x: Double, y: Double)
```

Functions

- To define a function, specify its name, parameters, and body like this

```
def maximum(x: Double, y: Double) = if (x > y) x else y
```

Functions

- To define a function, specify its name, parameters, and body like this

```
def maximum(x: Double, y: Double) = if (x > y) x else y
```

- If the function is not recursive, it is optional to specify its return type

```
def maximum(x: Double, y: Double): Double = if (x > y)
x else y
```

Functions (cont.)

- If body is more then one expression, enclose in { }

Functions (cont.)

- If body is more then one expression, enclose in { }

```
/*A function to raise a number m to the power n.
   Assume that n is a nonnegative integer*/
def power(m:Int, n:Int) = {
  var result = 1
  for ( i <- 1 to n)
    result *= m
  result
}
```

Functions (cont.)

- If body is more then one expression, enclose in { }

Calling the function:
power(5, 4)

```
/*A function to raise a number m to the power n.
   Assume that n is a nonnegative integer*/
def power(m:Int, n:Int) = {
  var result = 1
  for ( i <- 1 to n)
    result *= m
  result
}
```

Functions (cont.)

- If body is more then one expression, enclose in { }

```
/*A function to raise a number m to the power n.
   Assume that n is a nonnegative integer*/
def power(m:Int, n:Int) = {
  var result = 1
  for ( i <- 1 to n)
    result *= m
  result
}
```

Calling the function: power(5, 4)

- Recursive version of above function ?

Functions (cont.)

- If body is more then one expression, enclose in { }

```
/*A function to raise a number m to the power n.
   Assume that n is a nonnegative integer*/
def power(m:Int, n:Int) = {
  var result = 1
  for ( i <- 1 to n)
    result *= m
  result
}
```

Calling the function:
power(5, 4)

- Recursive version of above function

```
def raise(m: Int, n: Int): Int =  if ( n <= 0) 1 else m * raise(m, n -1)
```

What if a function returns no value?

Procedures

- A procedure is a function that does not return a value
- Usually used for its side effect such as printing
- You can define such function/procedure by
  - either specifying **?** as the return type

Procedures

- A procedure is a function that does not return a value
- Usually used for its side effect such as printing
- You can define such function/procedure by
  - either specifying Unit as the return type

```
def sayHello(name: String): Unit = {
    println("Hello " + name)
}

sayHello("David") // Hello David
```

Procedures (cont.)

  - or, omitting the return type and not using = symbol before the function body

Procedures (cont.)

- or, omitting the return type and not using = symbol before the function body

```
def sayGoodBye(name: String) {
    println("Good Bye " + name)
}

sayGoodBye("David") // Good Bye David
```

Functions as First class Citizens

- Functions in Scala, are first class citizens
  - can be assigned to a variable, stored in a data structure, passed as a parameter to another function, or be the returned value of another function

Functions as First class Citizens - Examples

```
import math._
val num = 3.14
val f = ceil _
//_ is needed to let Scala know that you intended to assign ceil as a ref to f

f(num) // 4
```

Functions as First class Citizens - Examples

```
import math._

//A function that takes another function as a parameter
def f2(functionArgument: (Double) => Double, dataArgument: Double) =
   functionArgument(dataArgument)
```

Functions as First class Citizens - Examples

```
import math._

//A function that takes another function as a parameter
def f2(functionArgument: (Double) => Double, dataArgument: Double) =
   functionArgument(dataArgument)

f2(math.sqrt, 2)    //res156: Double = 1.4142135623730951
```

Functions as First class Citizens - Examples

```
import math._

//A function that returns a function as its result
def f3() = math.sqrt _

f3()(4) //res165: Double = 2.0
```

Anonymous Functions - Examples

Anonymous Functions - Examples

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

Anonymous Functions - Examples

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

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

Anonymous Functions

- A function without a name is called an anonymous function (aka function literal)
- An anonymous function is defined with input parameters in parenthesis, followed by a right arrow and the body of the function
- The body of a functional literal is enclosed in "optional" curly braces.

Anonymous Functions - Examples

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

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

- We can store an anonymous function in a variable
  val add100 = (x: Int) => x + 100

Anonymous Functions

- Anonymous functions are useful when a function is passed as an argument to another function

```
def doSomething(func: (Double) => Double, x: Double): Double = {
  func(x)
}

doSomething(math.sqrt, 2)
//1.4142135623730951
```

Anonymous Functions

- Anonymous functions are useful when a function is passed as an argument to another function

```
def doSomething(func: (Double) => Double, x: Double): Double = {
  func(x)
}

doSomething(math.sqrt, 2)
//1.4142135623730951

doSomething( (x: Double) => 0.25 * x, 12)
//3.0
```

Higher-Order Methods

- A function that takes a function as a parameter is called a higher-order function

```
def doSomething(func: (Double) => Double, x: Double): Double = {
  func(x)
}

doSomething( (x: Double) => 0.25 * x, 12)
```

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
    doSomething( (x: Double) => 0.25 * x, 12)
```

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

```
doSomething( (x) => 0.25 * x, 12)
```

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

```
doSomething( (x) => 0.25 * x, 12)
```

- If an anonymous function takes only one parameter, we can omit the () around the parameter

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

```
doSomething( (x) => 0.25 * x, 12)
```

- If an anonymous function takes only one parameter, we can omit the () around the parameter

```
doSomething( x => 0.25 * x, 12)
```

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

```
doSomething( (x) => 0.25 * x, 12)
```

- If an anonymous function takes only one parameter, we can omit the () around the parameter

```
doSomething( x => 0.25 * x, 12)
```

- If the parameter occurs only once on the right-hand side of the =>, we can replace it with an underscore

Parameter Inference

- Scala can deduce types whenever possible
- How can we call the following function?

```
def doSomething(func: (Double) => Double, x: Double): Double = { func(x) }
```

```
doSomething( (x: Double) => 0.25 * x, 12)
```

- Scala can deduce the type of x from the function header

```
doSomething( (x) => 0.25 * x, 12)
```

- If an anonymous function takes only one parameter, we can omit the () around the parameter

```
doSomething( x => 0.25 * x, 12)
```

- If the parameter occurs only once on the right-hand side of the =>, we can replace it with an underscore

```
doSomething( 0.25 * _, 12)
```

Closures

- A closure is a function that uses a **non-local non-parameter** variable captured from its environment.

Closures

- A function whose return value depends on one or more variables outside the function scope called a **closure**

```
var c = 10

def add(a: Int, b: Int) = a + b + c

add(3, 4) //?
```

Closures

- A function whose return value depends on one or more variables outside the function scope called a **closure**

```
var c = 10

def add(a: Int, b: Int) = a + b + c

add(3, 4) //17
```

Closures

- A function whose return value depends on one or more variables outside the function scope called a **closure**

```
var c = 10

def add(a: Int, b: Int) = a + b + c

add(3, 4) //17

c = 50
add(3, 4) //?
```

Closures

- A function whose return value depends on one or more variables outside the function scope called a **closure**

```
var c = 10

def add(a: Int, b: Int) = a + b + c

add(3, 4) //17

c = 50
add(3, 4) //57
```

Closures (cont)

- The function can be called when a non-local variable is no longer in scope
- The function is aware of any changes to such variable and will use the new values

Closures – Example (closures.scala)

Closures – Example (closures.scala)

```
class MyClass {
  def exec(f:(String) => Unit, name: String) = {
     f(name)
  }
}

var greetings = "Hello“ // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }
```

Closures – Example (closures.scala)

```
class MyClass {
  def exec(f:(String) => Unit, name: String) = {
      f(name)
  }
}

var greetings = "Hello“ // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }

val object1 = new MyClass()
object1.exec(sayHello, "Mark") // ?
```

Closures – Example (closures.scala)

```
class MyClass {
   def exec(f:(String) => Unit, name: String) = {
       f(name)
   }
}

var greetings = "Hello“ // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }

val object1 = new MyClass()
object1.exec(sayHello, "Mark") // Hello, Mark
```

Closures – Example (closures.scala)

```
class MyClass {
   def exec(f:(String) => Unit, name: String) = {
       f(name)
   }
}

var greetings = "Hello" // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }

val object1 = new MyClass()
object1.exec(sayHello, "Mark") // Hello, Mark

greetings = "Howdy"
object1.exec(sayHello, "Martin") //?
```

Closures – Example (closures.scala)

```
class MyClass {
  def exec(f:(String) => Unit, name: String) = {
     f(name)
  }
}

var greetings = "Hello" // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }

val object1 = new MyClass()
object1.exec(sayHello, "Mark") // Hello, Mark

greetings = "Howdy"
object1.exec(sayHello, "Martin") //Howdy, Martin
```

Closures – Example (closures.scala)

```
// Closure.scala file
class MyClass {
   def exec(f:(String) => Unit, name: String) = {
       f(name)
   }
}
var greetings = "Hello" // greetings is defined outside of the function
def sayHello(name: String) = { println(s"$greetings, $name") }

object Closure {
def main(args: Array[String])={
val object1 = new MyClass()
object1.exec(sayHello, "Mark") // Hello, Mark

greetings = "Howdy"
object1.exec(sayHello, "Martin") //Howdy, Martin
}
}
```

Arrays

- Similar to Arrays in Java and C++
- Homogeneous
- Fixed-length
- You can efficiently access any element in an array in constant time using () not []
- **Mutable** data structure
  - you can update an element in an array
- Arrays are zero indexed

Declaring an Array

- val names: Array[String] = new Array[String](3)
  //names: Array[String] = Array(null, null, null)

Declaring an Array

-
  ```
  val names: Array[String] = new Array[String](3)
  //names: Array[String] = Array(null, null, null)
  ```

  or

  ```
  val names = new Array[String](3)
  //names: Array[String] = Array(null, null, null)
  ```

Declaring an Array

- val names: Array[String] = new Array[String](3)
  //names: Array[String] = Array(null, null, null)

  or

  val names = new Array[String](3)
  //names: Array[String] = Array(null, null, null)

- val nums = new Array[Int](10)

Declaring an Array

-
  ```
  val names: Array[String] = new Array[String](3)
  //names: Array[String] = Array(null, null, null)
  ```

  or

  ```
  val names = new Array[String](3)
  //names: Array[String] = Array(null, null, null)
  ```

-
  ```
  val nums = new Array[Int](10)
  ```

-
  ```
  val s = Array("Hello", "World")
  ```

  // Note: **no new** when you supply initial values

Declaring an Array

- val names: Array[String] = new Array[String](3)
  //names: Array[String] = Array(null, null, null)

  or

  val names = new Array[String](3)
  //names: Array[String] = Array(null, null, null)
- val nums = new Array[Int](10)
- val s = Array("Hello", "World")
  // Note: **no new** when you supply initial values
- s(1) // World

  // Use () instead of [] to access elements

Variable-Length Arrays: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or **vector** in C++

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
```

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
b += 1 // Add an element at the end with +=
// ArrayBuffer(1)
```

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
b += 1 // Add an element at the end with +=
// ArrayBuffer(1)
b += (1, 2, 3, 5)  //  Add multiple elements at the end
// ArrayBuffer(1, 1, 2, 3, 5)
```

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
b += 1 // Add an element at the end with +=
// ArrayBuffer(1)
b += (1, 2, 3, 5)  //  Add multiple elements at the end
// ArrayBuffer(1, 1, 2, 3, 5)
b ++= Array(8, 13, 21) // You can append any collection with the ++=
// ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
```

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
b += 1 // Add an element at the end with +=
// ArrayBuffer(1)
b += (1, 2, 3, 5)  //  Add multiple elements at the end
// ArrayBuffer(1, 1, 2, 3, 5)
b ++= Array(8, 13, 21) // You can append any collection with the ++=
// ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
val a = b.toArray
//a: Array[Int] = Array(1, 1, 2, 3, 5, 8, 13, 21)
val c = a.toBuffer
//c: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
```

Variable-Length: Array Buffers

- ArrayBuffer is similar to ArrayList in Java or vector in C++

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or, b = new ArrayBuffer[Int]
b += 1 // Add an element at the end with +=
// ArrayBuffer(1)
b += (1, 2, 3, 5)  //  Add multiple elements at the end
// ArrayBuffer(1, 1, 2, 3, 5)
b ++= Array(8, 13, 21) // You can append any collection with the ++=
// ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
val a = b.toArray
//a: Array[Int] = Array(1, 1, 2, 3, 5, 8, 13, 21)
val c = a.toBuffer
//c: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
b.trimEnd(5) // Removes the last five elements
// ArrayBuffer(1, 1, 2)
```

Traversing

```
val a = Array(1, 7, 2, 9)
?
```

Traversing

```
val a = Array(1, 7, 2, 9)
for (i <- 0 until a.length)  \\ for (i <- 0 to a.length-1)
            println(s"$i: ${a(i)}")
\\ 0: 1
\\ 1: 7
\\ 2: 2
\\ 3: 9
```

Built—in Functions

- Array(1, 7, 2, 9).**sum** // 19
  - Works for ArrayBuffer too

Built—in Functions

- Array(1, 7, 2, 9).**sum** // 19
  - Works for ArrayBuffer too
- ArrayBuffer("Mary", "had", "a", "little", "lamb").**max**
  // "little"

Built—in Functions

- Array(1, 7, 2, 9).**sum** // 19
  - Works for ArrayBuffer too
- ArrayBuffer("Mary", "had", "a", "little", "lamb").**max**
  // "little"
- val b = ArrayBuffer(1, 7, 2, 9)

  val bSorted = b.**sorted**

  // bSorted is ArrayBuffer(1, 2, 7, 9)

Built-in Functions (cont.)

- val bDescending = b.**sortWith**(_ > _)

  // ArrayBuffer(9, 7, 2, 1)

Built-in Functions (cont.)

-
  ```
  val bDescending = b.sortWith(_ > _)
  // ArrayBuffer(9, 7, 2, 1)
  ```
- You can sort an array, but not an array buffer, in place:

  ```
  val a = Array(1, 7, 2, 9)
  scala.util.Sorting.quickSort(a)
  // a is now Array(1, 2, 7, 9)
  ```

Built-in Functions - mkString

- The **mkString** method displays the contents using separator between elements

  ```
  val a = Array(1, 7, 2, 9)
  a.mkString(" and ")
  // "1 and 7 and 2 and 9"
  ```

- A second variant has parameters for the prefix and suffix

  ```
  a.mkString("<", ",", ">")
  // "<1,7,2,9>"
  ```