

Quick Introduction - Access Scala on CloudxLab

Start scala by typing **scala** in the web console to start scala

Welcome to a short tutorial on Scala.

Scala is a programming language. In other words, with scala you can program computers.

The best way to learn a programming language is by writing code. So, please do not just watch this tutorial. Instead, work with it.

Let's get started. Login to cloudxlab.

Scala provides a nice interactive console which would give us result instantaneously. It is also called REPL - **read-evaluate-print-loop**. To start scala console or REPL, type scala after logging into cloudxlab web console.

Click on I am done after starting scala. Please do not close the scala console or the web console.

Scala - Quick Introduction - Variables and Methods

In the previous step, we started Scala successfully. Let's continue from there. You can use this Scala console a very simple calculator as well.

So, if you type 2+4 it would compute the sum. You can see that it has given the answer as 6 and also defined a variable called res0 which contains the result. The data type of res0 is Int which means integer.

To see the value of a variable you can simply type the name of the variable and it would print the value contained in the variable.

You can see that it has displayed the value of res0.

We can also define our own variables by using var. So, if I say var x = 10, it would create x having value as 10.

Let's check the value of x by simply typing it.

Also, we can use x in other calculation. For example: var y = x * 4 So, this first multiplies x by 4 and then it assigns the result to y. You can see that the value of y is 40.

The way we have used addition and multiplication operators so far, we also have name operators called methods. These methods take inputs via arguments and optionally returns the result.

For example, print displays the value on the screen. Let's see.

if you type: `print("Hello, World")`

It prints the "Hello, World" on the screen.

Here print method takes a string or text as input by the way of the first argument. Also, notice that we define a string by enclosing something in double quotes.

Also, note that we define a single character by enclosing it in single quotes. You can see that we defined a variable "a" having character "h" as the value.

This print is an inbuilt method. There are some other libraries of methods which are provided by scala. For example, math library.

Let import all functions from maths library it by typing: `import math._` Now we can use functions such as `sqrt()` for computing square root of a value.

You can see the square root of 25 is 5.

Now, let's try to compute simple interest.

Let's define our principal amount is 10 [`var principal = 10`] And `rate_of_interest` as 10 percent annually `var rate = 10` And duration as 4 year `Var duration = 4` So, our interest would be `principal * rate * duration` divided by 100. `var interest = principal * rate * duration / 100`.

So, you can see the interest is 4.

Now, we change the principal amount to 100. And re-execute the formula we will see the interest is 40. Also, note that you can up arrow key to go to previous expressions.

If principal amount is 500 and rate is 12 % yearly, how much would be the interest for a duration of 5 years.

We can also define our own methods so that we don't need to write the same code again and again.

Let's define a method called `simpleInterest` in the following way: `def simpleinterest` bracket followed by arguments

Our first argument is `principal` which is integer and second argument is the rate which double or decimal and third argument `duration` in years which is an integer.

And the return type is of type double.

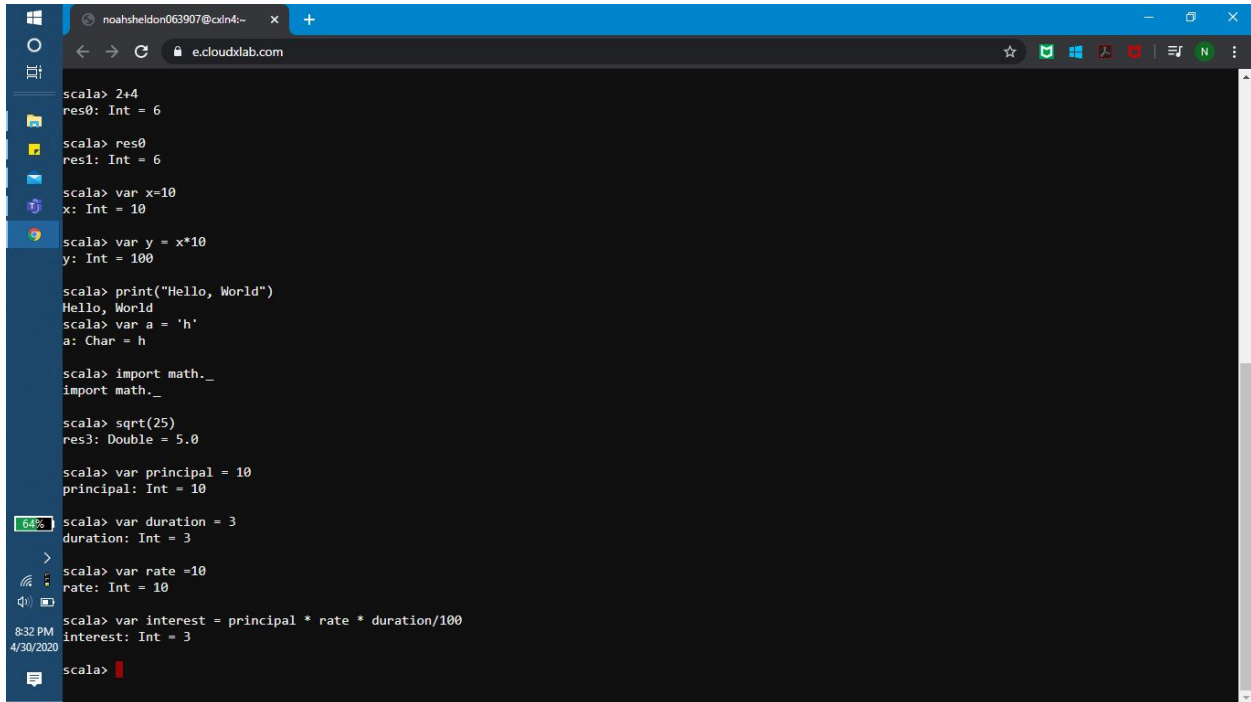
We start the body of the method after the open curly bracket. Inside this function, we are going to return the `simpleinterest` which `principal * rate * duration` divided by 100.

Then we close the curly brackets to mark the end of the method definition.

Let's try to call the method that we have just defined by providing `principal` as 700, rate of interest is 7, duration as 4 years

You can see that the interest is 196.

So, we have created our own interest computing machine which we can use as many times as we wish.

A screenshot of a web browser window showing a Scala REPL session. The browser's address bar displays 'e.cloudxlab.com'. The REPL interface has a dark background with white text. The session log on the left shows the following commands and results: 'scala> 2+4' resulting in 'res0: Int = 6'; 'scala> res0' resulting in 'res1: Int = 6'; 'scala> var x=10' resulting in 'x: Int = 10'; 'scala> var y = x*10' resulting in 'y: Int = 100'; 'scala> print("Hello, World")' resulting in 'Hello, World'; 'scala> var a = 'h'' resulting in 'a: Char = h'; 'scala> import math._' resulting in 'import math._'; 'scala> sqrt(25)' resulting in 'res3: Double = 5.0'; 'scala> var principal = 10' resulting in 'principal: Int = 10'; 'scala> var duration = 3' resulting in 'duration: Int = 3'; 'scala> var rate =10' resulting in 'rate: Int = 10'; 'scala> var interest = principal * rate * duration/100' resulting in 'interest: Int = 3'; and finally 'scala>' with a red cursor. The system tray at the bottom left shows the time as 8:32 PM on 4/30/2020. The browser's status bar at the bottom indicates a 64% zoom level.

```
scala> 2+4
res0: Int = 6

scala> res0
res1: Int = 6

scala> var x=10
x: Int = 10

scala> var y = x*10
y: Int = 100

scala> print("Hello, World")
Hello, World
scala> var a = 'h'
a: Char = h

scala> import math._
import math._

scala> sqrt(25)
res3: Double = 5.0

scala> var principal = 10
principal: Int = 10

scala> var duration = 3
duration: Int = 3

scala> var rate =10
rate: Int = 10

scala> var interest = principal * rate * duration/100
interest: Int = 3

scala>
```

Scala - Features

Let's go deep into Scala.

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way.

Scala smoothly integrates the features of object-oriented and functional languages

Scala is statically typed. When we deploy jobs which will run for hours in production, we do not want to discover mid way that the code has unexpected runtime errors. With Scala, you can be sure that your code will not give you unexpected errors while running in production.

Since Scala is statically typed we get performance and speed over dynamic languages.

Unlike Java, in Scala, we do not have to write quite as much code to perform simple tasks and its syntax is very similar to other data-centric languages.

Scala is a stable language used in enterprises, financial sectors, retail, gaming for many years

In 1990, Martin Odersky, creator of Scala, made Java better by introducing generics in `javac` - the Java Compiler

In 2001, he decided to create even better Java and in 2003, there was the first experimental release of Scala. To prove the correctness of Scala, `scala2.0` was compiled in Scala itself in 2005.

In 2011, corporate stewardship was brought to ensure that the language was enterprise ready at all times.

Scala compiler compiles scala code into Java bytecode. The resulting bytecode is executed by a JVM - the Java virtual machine.

Since Scala and Java have a common runtime, Java libraries may be used directly in Scala code and vice versa.

Scala - Installation on your own machine

Note: You can skip this page if you are okay with using CloudxLab environment. Instructions below are for installing Scala on your PC.

Since Scala code is compiled to Java Bytecodes, Scala installation requires Java Software Development Kit (SDK) installed on the machine.

On CloudxLab, Scala is already installed and readily available in the right-hand side frame. The right-hand side frame is essentially Jupyter which is an online environment for programming, analytics and machine learning.

If you want to use scala elsewhere you might have to go for the installation as outlined below.

Java Installation

- Download Java SDK from [Oracle Download Page](#)
- Install the java using the downloader
- Check if Java is installed using command `java -version` in terminal

Set Your Java Environment

Here, `java-current` is the directory where java is installed

```
export JAVA_HOME=/usr/local/java-current
```

Add java to the path

```
export PATH=$PATH:$JAVA_HOME/bin/
```

Install Scala

Download Scala from <http://www.scala-lang.org/downloads>. Current version downloaded file is - scala-2.12.7.tgz and it is downloaded to ~/Downloads Unpack Scala binary

```
cd ~/Downloads
tar -xvf scala-2.12.7.tgz
sudo mv scala-2.12.7 /usr/local/
sudo chmod -R 755 /usr/local/scala-2.12.7/bin
export SCALA_HOME=/usr/local/scala-2.12.7/
export PATH=$PATH:$SCALA_HOME/bin
```

Add the last 2 lines from above in .profile or .bash_profile to set the paths permanently.

Now run, `scala` command and you should get scala prompt.

```
$ scala
Welcome to Scala 2.12.7 (OpenJDK 64-Bit Server VM, Java 10.0.2).
Type in expressions for evaluation. Or try :help.

scala> print("Hello");
Hello
scala>
```

Compiling & Running Code

Login to the CloudxLab web console. Create a directory 'scala' and go inside it. Create a file hello_world.scala using the command "nano hello_world.scala". You can also use vim editor instead of nano. Type the code as displayed on the screen. Press "control o" to save the file and then press enter. Press "control x" to exit the nano editor.

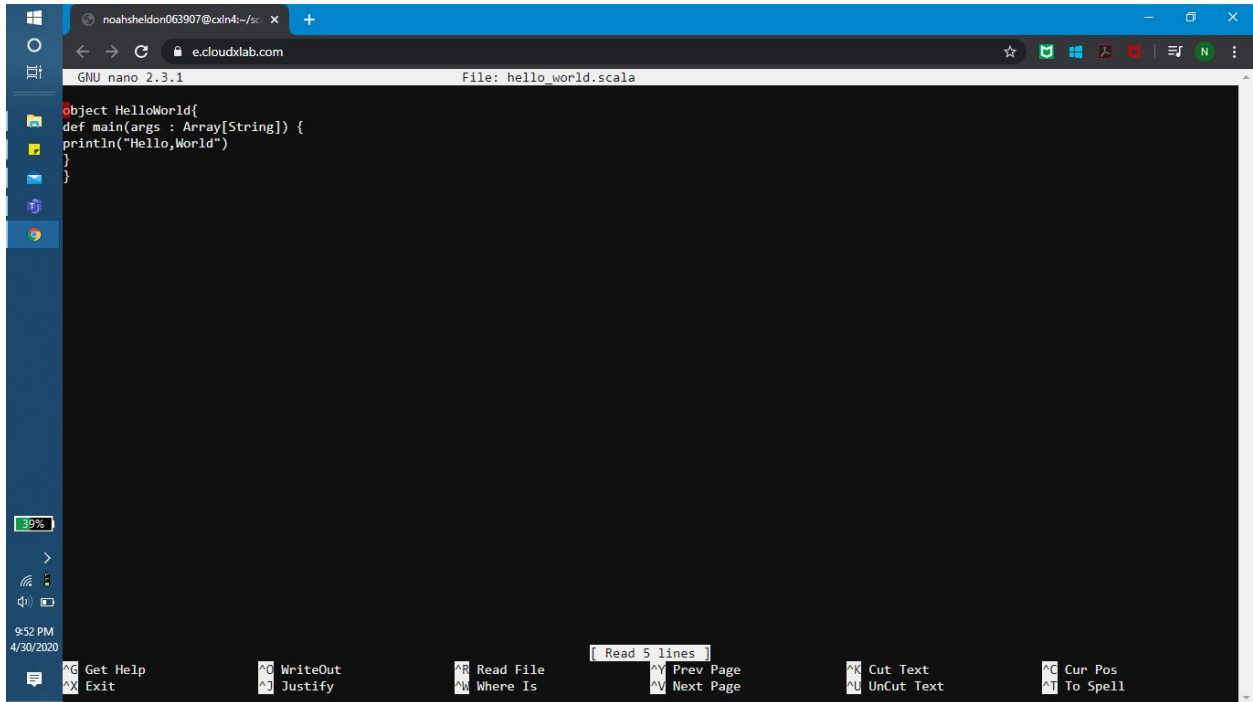
To compile the code, type "scalac hello_world.scala" and press enter. You can see two class files HelloWorld\$.class, HelloWorld.class. These .class files are generated bytecode which gets executed by JVM.

To run the code, type "scala HelloWorld". You can see that "Hello, world!" is printed on the screen.

Let's understand the structure of the code. It consists of a method called "main". The "main" method takes the command line argument as a parameter, which is an array of strings. and prints "Hello world" on the screen. HelloWorld is a singleton object, that is a class with a single instance. We'll discuss "object" later in the course.

If we've just one .scala file, we can also run it using the scala interpreter. Type "scala hello_world.scala" and press enter. We can see that "Hello, world!" gets printed on the screen.

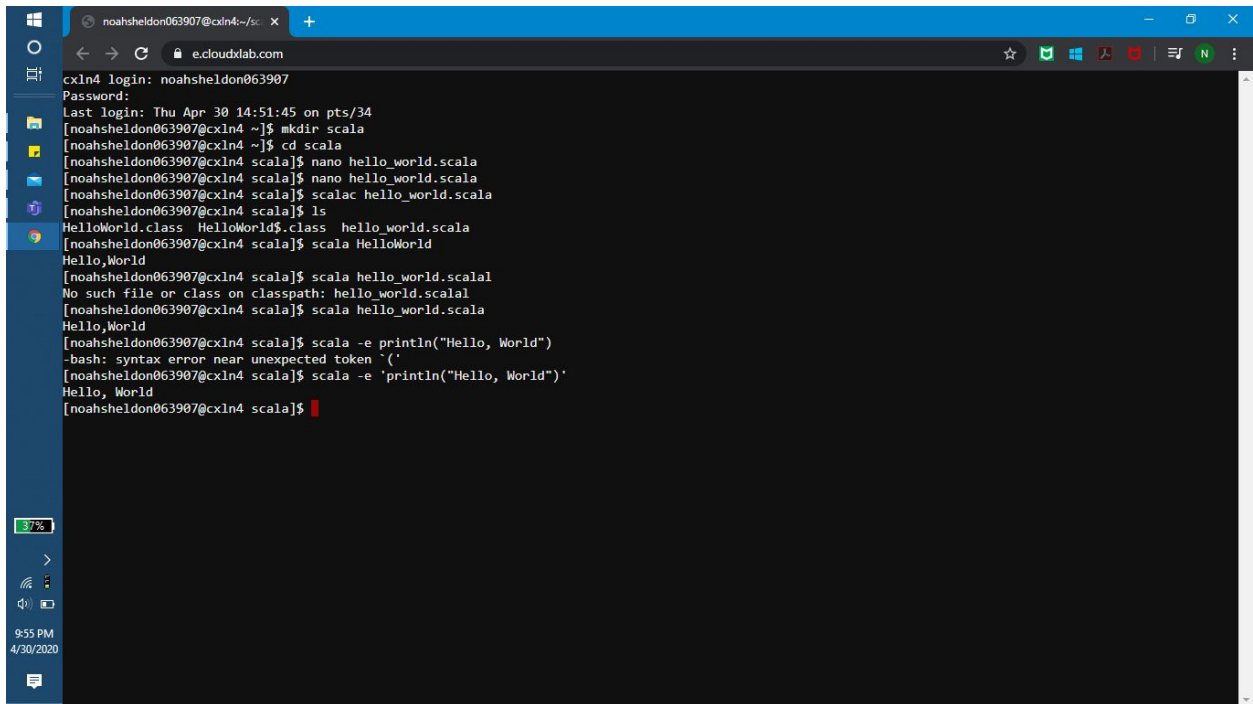
We can evaluate Scala code by the way of arguments using -e option. Type scala -e 'println("Hello, World!")' and "Hello, world!" gets printed on the screen



The screenshot shows a terminal window with a nano text editor open. The editor is editing a file named `hello_world.scala`. The code in the file is:

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

The terminal window shows the user `noahsheldon063907` at a prompt. The nano editor's status bar at the bottom indicates the file is `File: hello_world.scala` and shows various keyboard shortcuts like `Get Help`, `Exit`, `WriteOut`, `Justify`, `Read File`, `Where Is`, `Read 5 lines`, `Prev Page`, `Next Page`, `Cut Text`, `UnCut Text`, `Cur Pos`, and `To Spell`.



The screenshot shows a terminal window with the following commands and output:

```
cxln4 login: noahsheldon063907
Password:
Last login: Thu Apr 30 14:51:45 on pts/34
[noahsheldon063907@cxln4 ~]$ mkdir scala
[noahsheldon063907@cxln4 ~]$ cd scala
[noahsheldon063907@cxln4 scala]$ nano hello_world.scala
[noahsheldon063907@cxln4 scala]$ nano hello_world.scala
[noahsheldon063907@cxln4 scala]$ scalac hello_world.scala
[noahsheldon063907@cxln4 scala]$ ls
HelloWorld.class HelloWorld$.class hello_world.scala
[noahsheldon063907@cxln4 scala]$ scala HelloWorld
Hello,World
[noahsheldon063907@cxln4 scala]$ scala hello_world.scala
No such file or class on classpath: hello_world.scala
[noahsheldon063907@cxln4 scala]$ scala hello_world.scala
Hello,World
[noahsheldon063907@cxln4 scala]$ scala -e println("Hello, World")
-bash: syntax error near unexpected token '('
[noahsheldon063907@cxln4 scala]$ scala -e 'println("Hello, World")'
Hello, World
[noahsheldon063907@cxln4 scala]$
```

Scala - Program Structure

A Scala program can be defined as a collection of objects that communicate by invoking each other's methods.

Components of a Scala program:

Class - A class can be defined as a template/blueprint/structure that describes the behaviors/states that are related to the class.

Object - An object is an instance of a class and has states and behaviors. Example - A car has color, make, model as states and move, brake, as behaviors.

Methods - A method is a behavior of a class and hence of its instantiated objects. Methods have logic and code to manipulate the data and execute the action.

Fields - A field is state of a class and hence of its instantiated objects, which define properties of the objects. An object's state is created by the values assigned to these fields.

Closure - A closure is a function, which returns values depending on the input to it. Input is given via values assigned to variables declared outside this function.

Traits - A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

Variables - Variables are placeholders for any attributes which need to be calculated, manipulated or assigned to. This is similar to any other programming language.

We will study about these building blocks in upcoming sessions.

Scala - Data Types

Scala has all the same data types as Java, with the same memory footprint and precision.

You will notice that integers and strings are the most commonly used data types to represent the real world entities.

Let's have a look at Scala data types.

Byte - 8 bit signed value. Range from -128 to 127

Short - 16 bit signed value. Range -32768 to 32767

Int - 32 bit signed value. Range -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. Range from U+0000 to U+FFFF

String - A sequence of Chars

Boolean - Either the literal true or the literal false

Unit - Corresponds to no value

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

Scala - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Arithmetic Operators Relational Operators Logical Operators Bitwise Operators Assignment Operators

Examples with A = 35 and B = 15

Arithmetic Operators

+	Add	(A + B) = 50
-	Subtract	(A - B) = 20
*	Multiply	(A * B) = 525
/	Divide	(A / B) = 2
%	Modulus	(A % B) = 5

Relational Operators

==	Equality check	(A == B) is not true
!=	Inequality check	(A != B) is true
>	Greater than check	(A > B) is true
<	Less than check	(A < B) is not true
>=	Greater than or equal	(A >= B) is true
<=	Less than or equal	(A <= B) is not true

Logical Operators(assume A = true, B = false)

&&	Logical AND	(A && B) is false
	Logical OR	(A B) is true
!	Logical NOT	!(A && B) is true

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation.

a	b	a & b	a b	a ^ b
0	0	0	0	0

0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format, they will be as follows ?

```
A   = 0011 1100
B   = 0000 1101
-----
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A  = 1100 0011
```

Assignment Operators

=	Simple assignment	C = A + B	
+=	Add AND assignment	C += A	(same as C = C + A)
-=	Subtract AND assignment	C -= A	(same as C = C - A)
*=	Multiply AND assignment	C *= A	(same as C = C * A)
/=	Divide AND assignment	C /= A	(same as C = C / A)
%=	Modulus AND assignment	C %= A	(same as C = C % A)
<<=	Left shift AND assignment	C <<= 2	(same as C = C << 2)
>>=	Right shift AND assignment	C >>= 2	(same as C = C >> 2)
&=	Bitwise AND assignment	C &= 2	(same as C = C & 2)
^=	bitwise exclusive OR and assignment	C ^= 2	(same as C = C ^ 2)
=	bitwise inclusive OR and assignment	C = 2	(same as C = C 2)

Scala - Variables and Type Inference

Variables are nothing but reserved memory locations to store values. This means that when we create a variable, the compiler allocates a memory based on its data type.

There are two types of variables in Scala - Mutable and Immutable. Mutable variables are defined using var keyword and their value can be changed. Immutable variables are defined using val keyword and their value can not be changed once assigned. You can think of Immutable variables as final variables in Java

Let's do a hands-on on variables. Login to CloudxLab and type scala to launch the scala shell.

Let's define an immutable variable x of integer type with value 8. Type val x: Int = 8. Now try changing the value of x to 9. We have got an error. As discussed we can not change the value of an immutable variable.

Let's define a mutable variable "y" of integer type with value 7. Type var y: Int = 7. Now try changing the value of "y" to 9. We can see that its value is now 9.

Let's understand why do we need immutable variables?

In large systems, we do not want to be in a situation where a variable's value changes unexpectedly as this may lead to unpredictable results.

In the case of Threads, APIs, functions, and classes, we may not want some of the variables to change.

In these scenarios, we define the variables as immutable variables.

In scala, we can define the variables without specifying their datatype. The scala compiler can understand the type of the variable based on the value assigned to it. This is called variable type inference.

Let's do a hands-on on type inference

Let us define a variable x and assign it a value "hi". Type `var x = "hi"` in the scala shell. As we can see, scala has determined the type of variable x as "String".

Let us explicitly define the type of variable x as a string. Type `var x:String = "hi"`. We can see that variable x is a string.

We should always try to explicitly define a type of variable to ensure that the code is compiled faster and to avoid any unpredictable results.

Scala - Assessment - Variable Definitions

As part of this exercise, we will learn how to define variables.

INSTRUCTIONS

- Define an immutable (using `val`) integer variable `my_imx` with value 10.
- Define a mutable (using `var`) integer variable `my_mutx` with value 20.
- Let us define a variable of type String with the name `myname` and assign it a value "hi".
- Press "Shift + Enter" to execute the code in the Jupyter notebook.
- Click on "Submit Answer".

Scala - Assessment - Declare a variable for city

Let's try to define another variable.

Declare a variable named `state` whose value is "New York". Note that this is a multi-character string so this variable will be a string, not a character.

Do this in Jupyter Notebook on the right-hand side tab and submit the answer.

INSTRUCTIONS

- Switch to Jupyter tab.
- Write the code in Jupyter.
- Press Shift+Enter.
- Click on "Submit Answer".

Scala - Assessment - Calculate Simple Interest

Let's try to solve a simple problem in Scala - Calculation of the simple interest.

We want to calculate what would be the simple interest for 5 years if the principal amount is 500 and the interest rate is 12% annually.

We will need to define 3 variables and perform the calculation to get the interest.

INSTRUCTIONS

- Open Jupyter notebook on the right-hand side tab.
- Declare integer variables as described below.
- `principal` - assign a value of 500
- `interest_rate` - assign a value of 12
- `years` - assign a value of 5
- Now, declare a new variable `interest` which will be assigned the value of calculation of interest like below:
`interest` = [calculation of simple interest using above defined variables]
- Press Shift+Enter in each of the cells where you wrote the code.
- Click on "Submit Answer".

Scala - Strings

A string is a sequence of one or more characters.

The `String` is an immutable object in Scala which means that its object can not be changed, only reference to the object can be changed.

In below example, when variable `str` is assigned a new value "hello all", String object doesn't change, rather a new String object is created in memory and reference to the new object is set to `str`.

Similarly, when string `str` is concatenated with another string " there", String object doesn't change and its value remains same which is "hello all".

But when string `str` is assigned the output of concatenation, a new String object is created in memory and reference to the new object is set to `str`.

```
object Hello {  
  def main(args: Array[String]) {  
    var str:String = "hello"  
    println(str)  
    str = "hello all"  
    println(str)  
    str.concat(" there")  
    println(str)  
    str = str.concat(" there")  
    println(str)  
  }  
}
```

Scala - String Formatting and Interpolation

Formatting:

Strings can be formatted using `printf()` method to get output in required formats.

```
object Demo {  
  def main(args: Array[String]) {  
    var floatVar = 10.123  
    var intVar = 5000  
    var stringVar = "Hello There"  
  
    var fs = printf("The value of the float variable is " + "%f", while the value of  
the integer " + "variable is %d, and the string" + "is %s", floatVar, intVar,  
stringVar);  
  
    println(fs)  
  }  
}
```

Interpolation:

This mechanism evaluates the string value in runtime.

Strings can be interpolated or expanded using various ways.

- **The `s` interpolator**

It allows the use of variable directly in processing a string, when you prepend 's' to it.

```
val name = "John"
println(s"Hello $name") //output: Hello John
```

String interpolation can also process arbitrary expressions.

```
println(s"2 + 3 = ${2 + 3}") //output: 2 + 3 = 5
```

- **The `f` interpolator**

It allows creating a formatted String, similar to printf in C language. While using 'f' interpolator, all variable references should be followed by the printf style format specifiers such as %d, %i, %f, etc.

```
val height = 1.7d
val name = "John"
println(f"$name%s is $height%2.2f meters tall") //John is 1.70 meters tall
```

- **The `raw` interpolator**

It is similar to `s` interpolator except that it performs no escaping of literals within a string.

Output with `s` option:

```
println(s"Result = \n a \n b")
```

```
Result =
 a
 b
```

Output with `raw` option:

```
println(raw"Result = \n a \n b")
```

```
Result = \n a \n b
```

Scala - String Methods

There are many methods to work on strings and we will discuss here a few important ones.

int length() - Get string length:

```
var str1:String = "one"
println("Length is " + str1.length());
```

String concat(String) - Concat one string to another:

```
var str1:String = "one"
var str2:String = str1.concat(" two")
println("Combined string is " + str2);
```

char charAt(int index) - Returns the character at the given index first index being 0:

```
var str:String = "New York"
println("Char at 5th index is " + str.charAt(4));
```

String replace(char oldChar, char newChar) - Returns a new string after replacing all occurrences of string oldChar with string newChar:

```
var str:String = "New York"
println("New string is " + str.replace("York", "Shire"));
```

String replaceAll(char oldChar, char newChar) - Returns a new string after replacing all occurrences of string oldChar with string newChar. It is same as `replace` function but additionally, it can also use regular expressions (regex).

```
var str:String = "New York 1ab2c3"
println("New string is " + str.replaceAll("[0-9]", "x"));
```

To replace multiple chars, use `|` separator between patterns.

```
var str:String = "New York"
// replace all w and k with z
println("New string is " + str.replaceAll("w|k", "z"));
```

String[] split(String regex) - Splits the string around matches of the given regular expression.

```
var str:String = "New,York,Hello, There"
var strarray = new Array[String](3)
strarray = str.split(",")
println("New string is " + strarray(0));
println(strarray.deep.mkString("\n"))
```

String **substring**(int beginIndex, int endIndex) - Returns the substring of a string ending at endIndex-1.

```
var str:String = "NewxYorkyHello, There"  
var str1 = str.substring(4,9)  
println("New string is " + str1)
```

Scala - Assessment - Write function to remove vowels

Let's now write a program which will remove all vowels from a given string.

Here, we will write a scala program whose main class name is **Purge** having a function named **purgeVowels** which would take a string as input and return another string with all vowels removed from the input string.

Usage Example:

Purge.purgeVowels("New York") should return "Nw Yrk".

INSTRUCTIONS

- Switch to Jupyter tab.
- Write the code in Jupyter.

The program skeleton will look like this. Complete the code of the function in this to remove the vowels from the input string.

```
object Purge {  
  def purgeVowels( str:String ) : String = {  
    return .....  
  }  
}
```

- Press Shift+Enter.

- Test the function with various inputs to make sure that it is returning the outputs as per requirements.
- Click on "Submit Answer".

```
object Purge {
  def purgeVowels( str:String ) : String = {
    var str1: String=str.replaceAll("a|e|i|o|u|A|E|I|O|U","")

    return str1

  }
}
Purge.purgeVowels("New York")
```

Scala - Quick Introduction - Conditions and Loops

In programming, the first step of logic is a conditional statement such as if-else.

Let's redefine our function and put a conditional check in our method of simple interest. If someone enters negative principal that is less than 0, we print "Wrong principal" and return 0 as result.

Everything between curly brackets will be executed if the condition "principal less than 0" is true. If the condition is false, the statement outside the if block will be executed and our interest is calculated and returned as usual.

Let's check by calling our method with principal amount as negative 100. You can see that it has printed "Wrong principal" and the interest is 0.

Sometimes you may need to operate on the list of things. For that Scala provide a List data type.

Let define our list having number 4 9 8. var a equals List(4, 9, 8)

We can operate on a list in a variety of ways. One most common way is to go through every element using a for loop.

For x in a where a is a list and x is the current element, we are printing x i.e. each value of a. Notice the arrow which is basically angle bracket followed by -.

So, you can see that it has called println method for each value of the list.

This is called a definite because it definitely ends.

There is another way of executing a logic multiple times - A while loop.

A while loop has two parts - one condition and another body. The body keeps getting executed as long as the condition is true. If the condition remains true forever, it would keep executing the body forever.

Let's print all numbers less than 15. var x = 15; So, we define a variable x with value 15.

`"while(x > 0){` We define a while loop which will execute if the condition `x > 0` is true.

`"println(x) "x = x - 1 ""}`

Inside while loop we have two statements - one for printing value of x and another decreases the value of x by 1.

This loop will stop as soon as the value of x is zero.

As you can see we have printed all the number less than 15 but greater than 0.

Scala - Conditional Statements Examples

Conditional statements are controlled if `if` and `else` keywords.

`if` Statement

An `if` statement consists of a Boolean expression followed by one or more statements.

Syntax The syntax of an 'if' statement is as follows.

```
if(Boolean_expression) {  
    // Statements will execute if the Boolean expression is true  
}
```

Example:

```
object Hello {  
    def main(args: Array[String]) {  
        var x = 5;  
  
        if( x < 10 ){  
            println("If condition is true");  
        }  
    }  
}
```

`If-else` Statement

An `if` statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax The syntax of an `if...else` is ?

```

if(Boolean_expression){
    //Executes when the Boolean expression is true
} else{
    //Executes when the Boolean expression is false
}

```

Example:

```

object Hello {
    def main(args: Array[String]) {
        var x = 5;

        if( x < 10 ){
            println("If condition is true");
        } else {
            println("If condition is false");
        }
    }
}

```

Scala - Loop Statements Examples

Loop statements are used to run a program code in a loop which can break or continue subject to certain conditions.

Scala has 3 types of loop statements:

- **while loop** Repeats a statement or group of statements as long as a given condition is true. It tests the condition before executing the loop body.
- **do-while loop** Same as a while statement, except that it tests the condition at the end of the loop body.
- **for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

One way to run a **for loop** is using the left-arrow operator. The left-arrow ? operator is called a generator, so named because it's generating individual values from a range.

Example 1:

```

object Hello {
    def main(args: Array[String]) {
        // Local variable declaration:
        var a = 5;

        // while loop execution
        while( a < 10 ){
            a = a + 1;

```

```

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

```

Example 2:

```

object Hello {
    def main(args: Array[String]) {
        // Local variable declaration:
        var a = 5;

        // do loop execution
        do {
            a = a + 1;
            println( "Value of a: " + a );
        } while( a < 10 )
    }
}

```

Example 3:

```

object Hello {
    def main(args: Array[String]) {
        var a = 0;

        // for loop execution with a range
        println("Loop 1 - simple loop")
        for( a <- 1 to 5){
            println( "Value of a: " + a );
        }

        println("Loop 2 - loop with custom incremental value")
        for( a <- 1 to 10 by 2){
            println( "Value of a: " + a );
        }

        println("Loop 3 - loop in reverse, decreasing order")
        for( a <- 5 to 1 by -1){
            println( "Value of a: " + a );
        }

        println("Loop 4 - loop using until, one iteration less than simple loop")
        for( a <- 5 until 1 by -1){
            println( "Value of a: " + a );
        }

        println("Loop 5 - loop over collections")
        val myList = List(1,2,3,4,5);
    }
}

```

```

    for( a <- myList ){
        println( "Value of a: " + a );
    }

    println("Loop 6 - loop over multiples ranges, loop within loop")
    for( a <- 1 to 5; b <- 1 to 3){
        println( "Value of a: " + a );
        println( "Value of b: " + b );
    }
}
}

```

Scala - Assessment - Display Greeting Message

Let's take a look at a program which will return different greeting messages according to the time of the day.

We will write a Scala program with the main class name of `Greet` having a function named `greet` which would return different greeting messages as described below, according to the time input to the function.

Input time - 4 to till before 12 : Message Output - "Good Morning"

Input time - 12 to till before 16 : Message Output - "Good Afternoon"

Input time - 16 to till before 21 : Message Output - "Good Evening"

Input time - 21 to till before 4 : Message Output - "Good Night"

Example Usage:

If we execute `Greet.greet(5)`, then it should return "Good Morning" If we execute `Greet.greet(12)`, then it should return "Good Afternoon" and so on...

INSTRUCTIONS

- Write the code in Jupyter and press Shift + Enter.
- Test the function with various inputs to make sure that it is returning the outputs as per requirements.

- Click on "Submit Answer".

```
object Greet {
```

```
  def greet(time : Int ) : String = {
```

```
    if(time >= 4 & time < 12){
```

```
      return "Good Morning"
```

```
    }
```

```
    else if(time >= 12 & time < 16){
```

```
      return "Good Afternoon"
```

```
    }
```

```
    else if(time >= 16 & time < 21){
```

```
      return "Good Evening"
```

```
    }
```

```
    else{
```

```
      return "Good Night"
```

```
}  
  
}  
  
}
```

Scala - Classes and Objects

What is a class? A class is a way of creating your own data type. A class is a way of representing a type of value inside the system. For example, we can create a data type that represents a customer using a class. A customer would have states like first name, last name, age, address and contact number. A customer class might represent behaviors for how these states can be transformed for example how to change a customer's address or name. These behaviors are called methods. A class is not concrete until it has been instantiated using a "new" keyword.

Let's define a class. Login to the CloudxLab web console and type scala. As you can see on the screen, we've defined a person class and it has three parameters - first name of string type, last name of string type and age of integer type.

We've assigned these parameters to immutable class variables. Let's create an instance of the class. Type `val obj = new Person("Robin", "Gill", 42)`; ..we've created an instance of Person class with firstname as Robin, lastname as Gill and age as 42.

We can access class variables from the instance. Type `obj.firstname` to get the firstname of the person and `obj.age` to get the age of the person.

Let's define a class method to get the full name of the person.

As you can see on the screen, we have defined a method `getFullName`. This method concatenates `firstname` and `lastname` and returns a full name which is of string type.

Let's create an instance of the class and call the `getFullName` method. We can see that the full name of the person "Robin Gill" is printed on the screen.

Objects in Scala are Singleton. A singleton is a class which can only have one instance at any point in time.

Methods and values that aren't associated with individual instances of a class belong in singleton objects.

A singleton class can be directly accessed via its name. We can create a singleton class using the keyword `object`.

Let's create an object `Hello` which has a method `message` which returns a string `"hello"`. We can access the method `message` without instantiating the instance of `"Hello"`. This is because `Hello` is a singleton object and it is already instantiated for us when we try to call the `message` method. Type `Hello.message` to access the `message` method

Objects are useful in defining constants and utility methods as they are not related to any specific instance of the class. Utility methods take parameters and return values based on a calculation and transformation

Scala - Class examples

- A class can be defined as a template/blueprint/structure that describes the behaviors/states that are related to the class.
- Different instances of a class are called objects.
For example, a car be a class and different cars created/instantiated based on the car class will be objects of the car class.
- The objects of a class are created using `new` keyword.

Example:

```
class Car(makeOfCar: String, modelOfCar: String, yearOfCar: Int) {
  var make: String = makeOfCar;
  var model: String = modelOfCar;
  var year: Int = yearOfCar;
  var color = "";
  def changeColor(colorOfCar: String) {
    color = colorOfCar;
    println ("New color of car:" + color);
  }
}

object Demo {
  def main(args: Array[String]) {
    var car1 = new Car("Honda", "City", 1996);
    var car2 = new Car("Honda", "Accord", 1999);
    var car3 = new Car("Honda", "Amaze", 2015);
    println("Car details:" + car1.make + " " + car1.model + " " + car1.year + " " +
car1.color)
    println("Car details:" + car2.make + " " + car2.model + " " + car2.year + " " +
car2.color)
    println("Car details:" + car3.make + " " + car3.model + " " + car3.year + " " +
car3.color)
  }
}
```

```
}
```

Scala - Classes with multiple constructors

Classes have multiple constructors i.e. you can instantiate them in different ways.

In below example, a car be created with make/model/year or just with make/model.

Take note of the class method `dispCar` and function `dispCarDetails` to display the information of a car.

```
import java.io._

class Car() {
  var make: String = ""
  var model: String = ""
  var year: Int = 0
  var color = ""

  def this(makeOfCar: String, modelOfCar: String, yearOfCar: Int) {
    this()
    println("Creating new car object per first constructor")
    this.make = makeOfCar
    this.model = modelOfCar
    this.year = yearOfCar
    this.color = "Black"
  }

  def this(makeOfCar: String, modelOfCar: String) {
    this()
    println("Creating new car object per second constructor")
    this.make = makeOfCar
    this.model = modelOfCar
    this.year = 1990
    this.color = "Blue"
  }

  def dispCar() {
    println("Car details:" + this.make + " " + this.model + " " + this.year + " "
+ this.color)
  }
}

object Demo {
  def main(args: Array[String]) {
    var car1 = new Car("Honda", "City", 1996)
    var car2 = new Car("Honda", "Accord")
    var car3 = new Car("Honda", "Amaze", 2015)
    println("")
  }
}
```



```

        println("Displaying car info in main program:")
        println("Car details:" + car1.make + " " + car1.model + " " + car1.year + " "
+ car1.color)
        println("Car details:" + car2.make + " " + car2.model + " " + car2.year + " "
+ car2.color)
        println("Car details:" + car3.make + " " + car3.model + " " + car3.year + " "
+ car3.color)
        println("")
        println("Displaying car info via function:")
        dispCarDetails(car1)
        dispCarDetails(car2)
        dispCarDetails(car3)
        println("")
        println("Display car info via class func:")
        car1.dispCar()
        car2.dispCar()
        car3.dispCar()
    }

    def dispCarDetails(car:Car) = {
        println("Car details via func:" + car.make + " " + car.model + " " + car.year
+ " " + car.color)
    }
}

```

Scala - Assessment - Declare a class

Let's try to write a simple class.

This would be class named `Student` having below members.

- `name` of type String
- `sex` of type Char (M / F)
- `grade` of type Integer (ex - 1, 2, 3, 4 etc)
- `age` of type of Integer (ex - 7, 10, 13 etc)

Using the resulting class, one should be able to instantiate new objects like below:

```
var stu1 = new Student("John Doe", 'M', 15, 23 );
```

The class members of the object should be accessible as below:

```
var name_stu = stu1.name  
var sex_stu = stu1.sex  
var grade_stu = stu1.grade  
var age_stu = stu1.age
```

INSTRUCTIONS

- Switch to the Jupyter tab.
- Write the code of the class in Jupyter.
- Press Shift+Enter to execute the code.
- Try to create objects of the class and make sure that it works as per the above requirements.
- Click on "Submit Answer".

Scala - Assessment - Declare a class with multiple constructors

A class can have multiple constructors which means its objects can be created with different types of input fields.

For such a class, default constructor doesn't take any field and required constructors are defined inside the body of the class.

Let's declare a class `Student` having below member fields.

- `name` of type String
- `sex` of type Char (M / F)
- `grade` of type Integer (ex - 1, 2, 3, 4 etc)
- `age` of type of Integer (ex - 7, 10, 13 etc)

The Class should have multiple constructors so that using the resulting class, one should be able to instantiate new objects in various ways as shown below:

An object of `Student` can be created just by `name` with default values for other fields (sex = F, grade = 5 and age = 12).

```
var stu = new Student("Dey");
```

An object of `Student` can be created just by `name` and `sex` with default values for other fields (grade = 5 and age = 12).

```
var stu = new Student("Mack", 'F');
```

An object of `Student` can be created by `name`, `sex`, `grade` and `age`.

```
var stu = new Student("Doe", 'F', 8, 16);
```

The class members of the object should be accessible as below:

```
var name_stu = stu.name  
var sex_stu = stu.sex  
var grade_stu = stu.grade  
var age_stu = stu.age
```

INSTRUCTIONS

- Write the code of the class in Jupyter and press Shift+Enter.
- Try to create objects of the class in various ways as described and make sure that it works as required.
- Test that you are able to access the value of member fields using the dot operator.
- Click on "Submit Answer".

Scala - Function Representations

We have already discussed functions. We can write a function in different styles in Scala. The first style is the usual way of defining a function. Please note that the return type is specified as `Int`.

In the second style, please note that the return type is omitted, also there is no "return" keyword. The Scala compiler will infer the return type of the function in this case.

If the function body has just one statement, then the curly braces are optional. In the third style, please note that there are no curly braces.

Scala - Function Examples

Functions are program components which optionally take some input and perform some actions and optionally give some output. They are similar to methods of a class but functions exist without any classes too.

Syntax

```
def functionName ([list of parameters]) : [return type] = {  
    function body  
    return [expr]  
}
```

Example:

```
object Hello {  
    def addNumbers( x:Int, y:Int ) : Int = {  
        var sum:Int = 0  
        sum = x + y  
        return sum  
    }  
}
```

In below example, `calcSum` is a function which takes 2 numbers as input and gives the sum of those 2 numbers as `Integer` output. This function is called inside the main Scala program.

There is another function `dispDateTime` to display the current date and time. It doesn't return any input, rather just performs the action of printing the date and time. The return type of non-returning functions is `Unit`, same as `void` in other languages, and is an optional keyword.

```
//this is import of non-default packages to access specific utilities  
import java.text.SimpleDateFormat  
import java.util.Calendar  
//this is main class of the scala program  
object HelloWorld {  
    //this is the main method of the program  
    //this is the method which called when the program is run  
  
    def main(args: Array[String]) {
```

```

println("Hello, world!") // prints Hello World

//display current date time using a function
dispDateTime();

var result: Int = 0;
//get sum of 2 numbers using a function
result = calcSum(5, 7);
println("Result is:" + result)
}

def dispDateTime(): Unit= {
    val now = Calendar.getInstance().getTime()
    //println is a function which take st
    println("Current time is " + "as below")
    println(now+"\n")
}

def calcSum(a: Int, b: Int): Int = {
    var sum: Int = 0
    sum = a + b
    return sum
}
}

```

Scala - Assessment - Function to calculate sum of consecutive numbers

A function is a reusable component of a program which optionally takes some input parameters or arguments and performs some actions based on those parameters.

We will write a simple Scala program with a singleton object named `Calc` which will have a function named `calcSum`. This function takes 2 numbers as arguments and calculates the sum of all numbers between these 2 numbers including these 2 numbers.

For example: Executing `Calc.calcSum(5,8)` will calculate sum of numbers 5,6,7,8 which is $5 + 6 + 7 + 8 = 26$.

INSTRUCTIONS

- Switch to the Jupyter tab.

- Write the code of the class. The program's main singleton class name will be `Calc` and it will have a function named `calcSum`.

The program code would be like:

```
object Calc {
  def calcSum( x:Int, y:Int ) : Int = {
    ....
  }
}
```

- Press Shift + Enter to execute the code.
- Check that you are able to run `Calc.calcSum(5,8)` in Jupyter and that you are getting the expected result with some more test cases.
- Click on "Submit Answer".

Scala - Closures

A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

A closure also works as a variable.

Example 1:

```
val multiplier = (i:Int) => i * 10
```

Example 2:

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

Example 3:

```
object Hello {
  def main(args: Array[String]) {
    println( "multiplier(1) value = " + processme(1) )
    println( "multiplier(2) value = " + processme(2) )
  }
  var factor = 4
  val processme = (i:Int) => i * (factor+1)
}
```

Scala - Collections Overview

Scala collections provide different ways to store data.

The Scala collections hierarchy represents a structure of collections that can contain data in different ways.

At the top is the "traversable", something that can be traversed. Next in the hierarchy is the iterable. It contains data that can be iterated or looped over. Beyond iterable, we have a Sequence, a set, and a map. The sequence consists of linear Sequence and indexed Sequence.

In Linear Sequence, we iterate over one by one to find the elements that we want. A linked list is an example of linear Sequence. An indexed Sequence gives the ability to directly access the value inside of a sequence.

Scala - Sequences and Sets

A sequence is an ordered collection of data. Elements in the sequence may or may not be indexed. Examples of sequence are array, list, and vector

An array contains elements of the same type. Arrays are fixed in size and contain an ordered sequence of data. Array values are contiguous in memory, which means that the values are stored in consecutive memory addresses. Array elements are indexed by position. In the code shown on the screen, variable language is an array which contains 3 elements of string type "Ruby", "SQL" and "Python". We access array elements by their indexes. Arrays have a zero-based index. To access the first element type `languages(0)`. Arrays in Scala are mutable. We can change the values at specific indexes. To change "SQL" to "C++" type `languages(1) = "C++"`

To iterate over array elements we can use the 'for' loop. Let's iterate over languages array. Copy the code displayed on the screen. As you can see, each element of the languages array gets printed on the screen. The left arrow operator is called generator. We're iterating over the languages array one by one, assigning the element's value to x and printing x.

List in Scala represents a linked list having elements such that each element has a value and pointer to the next element. These lists have poor performance as data could be located anywhere in the memory. Compared to an array, a list is very flexible, as you do not have to worry about exceeding the size of the list. Theoretically, lists are unbounded in size, but practically their size is limited by the amount of memory allocated to the JVM.

Let us do a hands-on and create a list of integers 1, 2 and 3. Type `var number_list = List(1, 2, 3)`. To add a new element to the list, type `number_list:+ 4` As you can see, element 4 is added to the list. Let us try to change the value at index 1. Type

`number_list(1) = 7` We've got an error. We can not change the value since lists are immutable.

A set in Scala is a bag of data with no duplicates. Also, the ordering is not guaranteed in sets.

Let us create a set with integers 76, 5, 9, 1 and 2. Let us add 9 to it. Since duplicates are not allowed in sets, 9 is not added to the set. Let's add 20. We can see that 20 is added to the set. Note the order of 20. As discussed, set does not guarantee ordering. `set(5)` will result in boolean true as 5 is present in the set. `set(14)` will result in boolean false as 14 is not present in the set.

Scala - Collections - Tuples and Maps

Unlike an array and list, tuples can hold elements of different data types. Let's create a tuple with elements 14, 45.69 and "Australia"). We can create it either with `var t = (14, 45.69, "Australia")` or with `var t = Tuple3(14, 45.69, "Australia")`. In the second syntax, we are explicitly specifying that tuple will contain 3 elements.

tuples can be accessed using a 1-based accessor for each value. To access the first element, type `t._1`. To access the third element type `t._3`

tuples can be deconstructed into names bound to each value in the tuple. Let us understand it. Type `var (my_int, my_double, my_string) = t` As you can see that, `my_int` is assigned 14, `my_double` is assigned 45.69 and `my_string` is assigned Australia

Tuples are immutable. Let us try to set the first element to 18. Type `t._1 = 18`. We have an error.

Scala maps are a collection of key/value pairs. Maps allow indexing values by a specific key for fast access. You can think of a Scala map as a Java HashMap and Python dictionary.

Let's do a hands-on on Scala maps. Copy the code displayed on the screen. Colors map contains key/value pair of colors and their hex code. To see the hex code of color yellow, type `colors("yellow")`. As you can see, the hex code of yellow is #FFFF00. We can add new key/value pairs using += operator. Let's add color green and its hex value. Type `colors += "green" -> "#008000"` and press enter. Type `colors` to see the list of updated key/value pairs. We can remove key/value pair using -= operator. Let's remove key "red". Type `colors -= "red"` You can see that key "red" is no more in the map.

Let's iterate through colors map and print key/value pairs. Type the code displayed on the screen. Here we are iterating through colors map and printing the corresponding key and value.

Scala provides two types of maps - immutable and mutable. Please note by default, Scala uses the immutable Maps. It means that you can not change the value of a key. If you want to use the mutable maps, import `scala.collection.mutable.Map` class explicitly

Scala - Assessment - Declare a List of strings

A list is another collection available in Scala.

A list is similar to an array but lists are immutable which means that the elements of a list cannot be changed by assignment and the lists represent a linked list whereas the arrays are flat.

for example, the list of days in week:

```
// Make a list via the companion object factory  
val days = List("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",  
"Saturday")
```

Here is an example of a list of numbers:

```
val mainList = List(3, 2, 1)
```

INSTRUCTIONS

Declare the list in Jupyter named `strList`, having these elements in the same order:

- apple
- banana
- mango
- orange
- pears

Here, we must define the variable as a list of Strings as is the requirement above.

Note that the list is displayed with its contents. Once you have run the scala code using SHIFT+ENTER, please click on "Submit Answer".

Scala - Assessment - Declare a Set of integers

A set is a collection in Scala to store unordered unique items. For example, a set having strings "my" and "name" is defined as:

```
var myset:Set[String] = Set("my", "name")
```

Now, let's declare a set of integers, named `intSet`, having below elements.

- 2
- 4
- 17
- 19

To accomplish this, we will need to declare the set variable of type `Set[Int]` as this set is going to have integers only.

INSTRUCTIONS

- Switch to Jupyter tab.
- Declare the set variable in Jupyter.
- Press Shift+Enter.
- Submit your answer.

Scala - Assessment - Declare a Map of Char, Int

A map in Scala is like a dictionary or a collection of key-value pairs. You can search the map using key or value.

A simple map having data of states and capitals may look like this:>br>

```
val capitals = Map("AZ" -> "Phoenix", "CA" -> "Sacramento")
```

Now, let's declare a map depicting the alphabets and their position. So, this map will be of type (Char, Int) and let's name it `charMap`. The data of the map will be as below.

- A,1
- B,2
- C,3

Think about how such a map will be defined.

INSTRUCTIONS

- Switch to Jupyter tab.
- Declare the map variable as per the requirement in Jupyter.
- Press Shift+Enter.
- Submit your answer.

Scala - Assessment - Declare a Tuple

A tuple in Scala is an immutable collection to hold objects of different types.

A tuple is simply declared by putting the items inside () and its items are accessed via suffixes like `._1`, `._2` and so on. For example;

```
var tuple_new = ("1", "World")  
  
//or  
  
var tuple_new = Tuple2("1", "World")
```

```
//or  
  
var tuple_new = new Tuple2("1", "World")  
  
//get the elements  
  
var first_element = tuple_new._1  
var second_element = tuple_new._2
```

Now, let's declare a tuple named `myTuple`, having below items.

- 5
- "Hello"
- 100

INSTRUCTIONS

- Switch to Jupyter tab.
- Declare the tuple variable with the required name and items in Jupyter.
- Press Shift+Enter.
- Submit your answer.

Scala - Assessment - Declare an Iterator

Iterator in Scala is a way to access the elements of a collection one by one. The two basic operations on an iterator are `next` (to access the next item and put the pointer to next to next item) and `hasNext` (to check whether an iterator has more items or not).

A simple example of declaring and looping over iterator is given below:

```
val it = Iterator("You", "are", "next")  
while (it.hasNext){  
    println(it.next())  
}
```

An iterator is a collection of items which can be traversed using `hasNext()` and `next()` to get the next item. An iterator can be traversed only once after which items are lost.

INSTRUCTIONS

Declare an Iterator named `myIter`, having below items.

"I"

"am"

"going"

- Switch to Jupyter tab.
- Declare the iterator variable in Jupyter.
- Press Shift+Enter.
- Submit your answer.

Scala - File IO

Writing to a file

Scala uses `java.io.File` class to write the files to the operating system.

A `File` object is created followed by an object of `PrintWriter` class. The `write` function of `PrintWriter` class does the writing to the file.

Below example create a new `File` object and writes the lines to it.

```
import java.io._  
  
object IOtest {  
  def main(args: Array[String]) {  
    val writer = new PrintWriter(new File("new.txt" ))  
  
    writer.write("Hello There Scala")  
    writer.close()
```

```
}  
}
```

Reading from a file

Scala uses `scala.io.Source` class to read the files from the operating system.

A `File` object is created followed by an object of `PrintWriter` class. The `write` function of `PrintWriter` class does the writing to the file.

Below example create a new `File` object and writes the lines to it.

```
import scala.io.Source  
  
object IOTest {  
  def main(args: Array[String]) {  
    println("Content of the file is:" )  
    Source.fromFile("new.txt").foreach {  
      print  
    }  
  }  
}
```

Another to read the file content:

```
import scala.io.Source;  
  
for (line <- Source.fromFile("new.txt").getLines) {  
  println(line)  
}
```

Scala - Assessment - Write to a file

In Scala, reading from and writing to files on file-system is very easy. After importing `java.io._`, you can use an instance of `PrintWriter` to write to a file.

For example, to write string "Hello" to a file named `test.txt`, you can use below code:

```
val writer = new PrintWriter(new File("test.txt" ))
```

```
writer.write("Hello")  
writer.close()
```

Now, let's write a Scala program which will write below 2 phrases in 2 lines in a file named "New.txt".

```
Hello Scala  
Bye Scala
```

To write the second line into a new line, write a new line character to the file after the first line.

New line character is indicated by "\n". so, you will write "Hello Scala", then "\n" followed by "Bye Scala" to the file.

INSTRUCTIONS

- Switch to Jupyter tab.
- Don't forget to import `java.io._`
- Write the code to do the following:
 - Open a `PrintWriter` to a file with a name "New.txt"
 - Write "Hello Scala" into this `printwriter`
 - Write "Bye Scala" into this `printwriter`
 - Close the `printwriter`
- Press Shift+Enter.
- Submit your answer.

Scala - Higher Order Functions

In Scala, a higher-order function is a function which takes another function as an argument. A higher-order function describes "how" the work is to be done in a collection.

Let's learn the higher order function `map`. The `map` applies the function to each value in the collection and returns a new collection. Say we have a list of integers 3, 7, 13 and 16 and we want to add one to each value in the list. Using a higher order `map` function, we can map

over the list and add one to each value. As displayed on the screen, we have a new list with values 4, 8, 14 and 17.

Let's do a hands-on. Define a list of integers 1, 2 and 3. Type,

```
var list = List(1,2,3)
```

Let's add 1 to each element using the map function. To define a map function, type `list.map(x => x + 1)`. Press enter. As you can see, we have a new list with integers 2, 3 and 4

Here map function adds one to every value in the list. Each value in the list has the name "x". There is another syntax to define the map, where instead of giving a name to every value being used in the function, we use a placeholder underscore to represent the value.

Let's learn the higher order function flatMap.

`Flatmap` takes a function as an argument and this function must return a collection. The collection could empty.

The flatMap is called on a collection. The `flatMap` applies the function passed to it as an argument on each value in the collection just like `map`.

The returned collections from each call of function are then merged together to form a new collection.

Let's understand it with an example

Let's define a list of languages. Type,

```
var list = List("Python", "Go")
```

Here, we have a list of strings. A string is nothing but a sequence of characters. Define a `flatMap`.

Type `list.flatMap(lang => lang + "#").flatMap` appends "#" to every string in the list and then flattens down the string to characters. As you can see on the screen, # is appended to python and go and then "python#" and "go#" flatten down into the sequence of characters. We can also define the same function using the following `flatMap(_ + "#")` where the underscore is a placeholder for each value in the list.

So, the size of output collection could be larger than the size of input collection.

While in case of map the input and output collection were of the same size.

Let's learn the higher order function Filter. The filter applies a function to each value in the collection and returns a new collection with values that satisfy a condition specified in the function. Let's understand it with an example.

We have a list of languages Scala, R, Python, Go and SQL. To filter out languages which contain capital S, type

```
list.filter(lang => lang.contains("S"))
```

As you can see we have a new list now with only Scala and SQL as values. Each value in the list has name "lang"

Each of the previous higher-order functions returned a new collection after applying the transformation. But at times we do not want the functions to return a new collection. It is a waste of memory resources on JVM if we do not want a return value. Higher order function 'foreach' allows us to apply a function to each value of collection without returning a new collection. Let's say we have a list of values 1 and 2 and we just want to print each value in the list. We can use foreach function for this as we are not interested in the return value. To use foreach higher order function, type

```
list.foreach(println)
```

As you can see value 1 and 2 are printed on the screen.

Let's learn higher order function 'reduce'. Reduce is a very important concept in the MapReduce world. Let's say we have a list and we want to reduce it to add values together. Let's understand it with an example. We've a list of integers 3, 7, 13 and 16.

In our example, reduce adds the first two elements 3 and 7 which results in 10. Then it adds 10 to the next element which is 13 resulting in 23. Then it adds 23 to the next element in the list which is 16. As you can see, that final result of the reduce function is 39

Let's do a hands-on. We have a list of integers 3, 7, 13, 16. Let's define a reduce function to add the elements, type

```
list.reduce((x, y) => x + y)
```

As we can see, the result is 39. Here the reduce function takes two values named x and y and adds them.

We can also define the reduce function using `list.reduce(_ + _)`. Here the two underscores are placeholders for the two values.

Scala - Interaction with Java

As discussed earlier, we can use Java libraries in Scala. Let's check it. Copy the code displayed on the screen. This code uses `java.util` and `java.text` libraries and returns the current date in the United States. Let's run the code. As you can see, the current date in the United States is printed on the screen.

Scala - Build Tool - SBT

So far, we have just compiled and run one `.scala` file. If we are working on a big project containing hundreds of source files, it becomes really tedious to compile these files manually. We'll then need a build tool to manage the compilation of all these files. SBT is a build tool for Scala and Java projects, similar to Java's Maven or ant. SBT is already installed on CloudxLab so you can compile your Scala project directly on it.

Let's understand how to use SBT to build a Scala project. We've provided a sample code on the CloudxLab GitHub repository. Clone the CloudxLab GitHub repository in your home folder in CloudxLab.

This will create a `cloudxlab` folder in your home directory. Now, Go to that directory `cloudxlab`. To update a previously cloned repository you can use run command `'git pull origin master'`

Since I have already cloned the repository, I will just update it. Go to `scala/sbt` directory. Let's look at the `build.sbt` file.

`build.sbt` file lists all the source files your project consists of, along with other information about your project. SBT will read the file to understand what to do to compile the entire project.

Besides managing the project, SBT automatically manages dependencies. This means that if we need to use some libraries written by others in our project, SBT can automatically download the right versions of those libraries and include them in the project.

As you can see, the name of our project is `hello`, version of the project is `1.0` and the Scala version required for the project is `2.11.8`. We do not have any dependencies here. We'll show you how to include dependencies in the Spark streaming topic.

Also, please note that all the Scala files must be in the `src/main/scala` directory. You can see that we have `hello_world.scala` file inside the `src/main/scala` directory.

Let's run the project. Type `"sbt run"` from the root of your project. It will run the main class of the project. `Hello, world!` is now printed on the screen.

The screenshot shows a web browser window with a Jupyter Notebook interface on CloudXLab. The browser tabs include 'Scala - Build Tool - SBT | Autom...' and 'My Lab'. The address bar shows 'cloudxlab.com/assessment/displayslide/374/scala-build-tool-sbt?course_id=738&playlist_id=335'. The page header includes 'CLOUD x LAB', 'Scala > by Electronics and ICT Academy IITR', and a user profile with '12,950 XP'. The notebook content on the left explains SBT's dependency management and provides instructions for running the project. A 'Mark as Completed' button and navigation links ('Previous', 'Index', 'Next') are visible. Below the notebook, there are recommendations for other labs like 'Hive - Managed Tables - Hands-on', 'Chapter Overview - Writing Spark...', and 'Python Project - Churn Emails - Count...'. The terminal window on the right shows the following commands and output:

```
[noahsheldon063907@cxln4 ~]$ cd ~/cloudxlab && git pull origin master
bash: cd: /home/noahsheldon063907/cloudxlab: No such file or directory
[noahsheldon063907@cxln4 ~]$ git clone https://github.com/singhabhinav/cloudxlab.git
Cloning into 'cloudxlab'...
remote: Enumerating objects: 734, done.
remote: Total 734 (delta 0), reused 0 (delta 0), pack-reused 734
Receiving objects: 100% (734/734), 51.32 MiB | 31.03 MiB/s, done.
Resolving deltas: 100% (313/313), done.
Checking out files: 100% (288/288), done.
[noahsheldon063907@cxln4 ~]$ cd ~/cloudxlab && git pull origin master
From https://github.com/singhabhinav/cloudxlab
* branch      master       -> FETCH_HEAD
Already up-to-date.
[noahsheldon063907@cxln4 cloudxlab]$ cd scala/sbt
[noahsheldon063907@cxln4 sbt]$ ls
build.sbt  src
[noahsheldon063907@cxln4 sbt]$ nano build.sbt
```

Scala - Case Classes and Pattern Matching

Case classes are regular classes that are immutable by default. Immutability helps us in writing classes without worrying about or keeping track of where and when things are mutated. This, in turn, helps in writing code which does not act weird.

Case classes can be pattern matched. Pattern matching simplifies the branching logic and helps in writing more readable code.

For case classes, the compiler automatically generates the hashCode and equals method, hence less code.

It is recommended to use case classes as they help in writing more expressive and maintainable code

Let's understand case classes and pattern matching with a demo. We've provided sample code on CloudxLab GitHub repository. Clone the repository if you haven't else update it

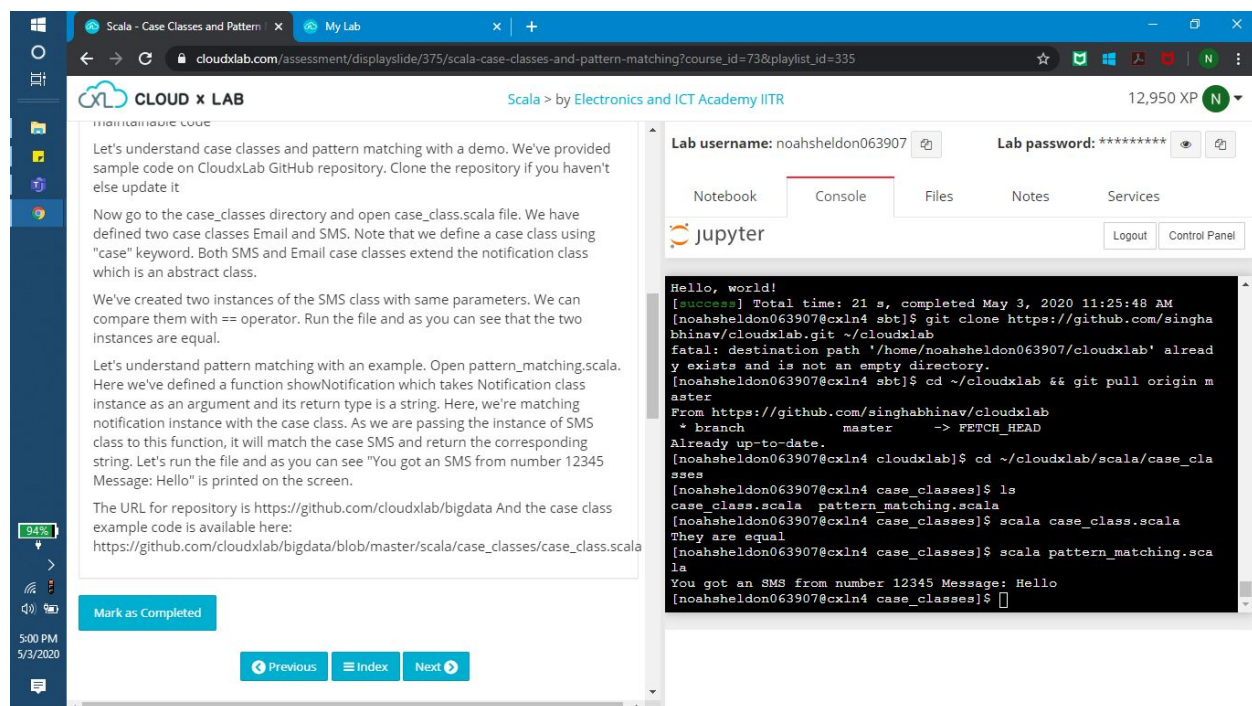
Now go to the case_classes directory and open case_class.scala file. We have defined two case classes Managed Email and SMS. Note that we define a case class using "case" keyword. Both SMS and Email case classes extend the notification class which is an abstract class.

We've created two instances of the SMS class with same parameters. We can compare them with == operator. Run the file and as you can see that the two instances are equal.

Let's understand pattern matching with an example. Open `pattern_matching.scala`. Here we've defined a function `showNotification` which takes `Notification` class instance as an argument and its return type is a string. Here, we're matching notification instance with the case class. As we are passing the instance of `SMS` class to this function, it will match the case `SMS` and return the corresponding string. Let's run the file and as you can see "You got an SMS from number 12345 Message: Hello" is printed on the screen.

The URL for repository is <https://github.com/cloudxlab/bigdata> And the case class example code is available here:

https://github.com/cloudxlab/bigdata/blob/master/scala/case_classes/case_class.scala



The screenshot shows the CloudXLab JupyterLab interface. The left pane contains the Scala code for pattern matching, which defines a `Notification` abstract class, `Email` and `SMS` case classes, and a `showNotification` function. The right pane shows the JupyterLab console output, which includes the JupyterLab startup message, the execution of `git clone`, `git pull`, and the execution of the Scala code, resulting in the output: "You got an SMS from number 12345 Message: Hello".

Scala - Variable Examples

Scala variables may or may not be any specific data type.

Scala allows variables to be defined in 2 ways - **Immutable** and **mutable**.

Immutable:

The value must be assigned at the time of declaring the variable whose value can not be changed later.

This type of variable is defined using `val`.

Example - `val x = 5`

Executing below to try to change the value of the variable will give an error.

```
x = 6
```

Mutable:

The value must be assigned at the time of declaring the variable but value can be changed later.

This type of variable is defined using `var`.

Example - `var y = 9`

Executing below to try to change the value of the variable will not give any error.

```
y = 10
```

More examples:

```
//without any data type
```

```
var x = 5
```

```
//with any data type of integer
```

```
var y:Int = 5
```

```
//with any data type of string
```

```
var z:String = "5"
```

```
//with any data type of char
```

```
var m:Char = 'a'
```

String literal has to be quoted in double quotes and single character has to be quoted in single quotes.

Multiple assignments

Scala supports multiple assignments. If a code block or method returns a Tuple (Tuple ? Holds collection of Objects of different types), the Tuple can be assigned to a `val` variable.

Example:

```
val (myVar1: Int, myVar2: String) = Pair(40, "Foo")
```