Scala for the Python Geeks

Abhishek Tiwari

Citation: *A. Tiwari*, "Scala for the Python Geeks", Abhishek Tiwari, 2011.
doi:10.59350/hhdc8-axf19

Published on: September 16, 2011

Scala is a JVM languages designed by Martin Odersky in early 2001 and it brings features of object-oriented programming and functional programming together. Scala is competing again several new JVM languages including but not limited to Groovy, Clojure, JRuby and Jython.

Scala resembles Python (and Ruby) in many ways and as a Pythonist I found it real easy to switch on Scala. The purpose of this tutorial series is to give Python geeks an idea about the similarities and differences between Scala and Python world.

##Getting Started Before we start this tutorial, below are some suggested reading for both Scala as well Python. I will be pulling lot of ideas and examples from these sources.

- Programming in Scala.Written by none other than Martin Odersky - the creator of Scala
- Scala for the Impatient. You can download a free PDF version here. A set of exercises at end of each chapter.
- Scala in Depth
- Beginning Scala. This book is a useful complement to Programming in Scala.
- Programming Python. First class reference text on Python
- Python Pocket Reference. A Swiss Army knife for Python developers.

Installation

You can download latest stable release and installers for Scala here. One of the key feature of Scala is Actors. In Scala Actors provide a concurrency model which is lot a easier than Java's native concurrency model. To add this creators of Scala has developed Akka a framework which simplifies writing concurrent, scalable and highly available applications through Actors. To get the most out of Scala's scalability features, use of Akka is highly desirable. Although Akka can be installed separately from Scala, it is better to use a simple, pre-integrated stack together with Akka provided by Typesafe.

After installation make sure that the scala/bin directory is on the PATH (User environment are set up correctly with `SCALA_HOME` and *bin* `PATH` variables).

Scala Interactive Interpreter (REPL)

Scala Interpreter (often called a REPL for Read-Evaluate-Print Loop) is Python equivalent of Interactive Mode. To get started type `scala` in your terminal.

```
$ scala
Picked up _JAVA_OPTIONS: -Xmx1024m
Welcome to Scala version 2.9.0.1 (Java HotSpot(TM) 64-Bit Server VM, Java
   1.6.0_26).
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala>
```

Initially you will find that Scala REPL takes bit longer load time than expected due to enormous start-up cost for the REPL. According to this Stackoverflow thread using REPL gradually causes JVM byte-code to be converted to native code, after which it's very fast.

Now let's check the `:help` on REPL ~~~ scala> :help All commands can be abbreviated, e.g. :he instead of :help. Those marked with a * have more detailed help, e.g. :help imports.

:cp add a jar or directory to the classpath :help [command] print this summary or command-specific help :history [num] show the history (optional num is commands to show) :h? search the history :imports [name name …] show import history, identifying sources of names :implicits [-v] show the implicits in scope :javap <path|class> disassemble a file or class name :keybindings show how ctrl-[A-Z] and other keys are bound :load load and interpret a Scala file :paste enter paste mode: all input up to ctrl-D compiled together :power enable power user mode :quit exit the interpreter :replay reset execution and replay all previous commands :sh run a shell command (result is implicitly => List[String]) :silent disable/enable automatic printing of results :type display the type of an expression without evaluating it ~~~

You can use Scala REPL as a calculator pretty much like Python interactive mode, ~~~ scala> 2+2 res0: Int = 4 scala> 2*2 res1: Int = 4 ~~~

```
>>> 2+2
4
>>> 2*2
4
```

The key difference is `res` or automatically generated `result` variable names such as `res0`, `res1` when a variable name is not provided. REPL interpreter also displays the type of the result in this case `Int` via Scala's built-in type inference mechanism. In Scala, often it is not necessary to specify the type of a variable, in fact the compiler can infer the type from the initialization expression of the variable.

For the record the Scala is statically typed language with type inference and it is completely orthogonal to dynamic languages like Python or Ruby. Although type inference may confuse some of us from Python or Ruby world which is actually done by the compiler (see more details on static vs dynamic).

Scala REPL Tab Completion

Scala REPL provides tab completion or suggestion which can be used for package and class completion as well as member completion (suggesting static and instance methods for a Scala object). For member completion use `object.` and tab. ~~~ scala> import scala.co collection compat concurrent

scala> var t = 2+2 t: Int = 4

scala> t. % & * + -
/ > >= » »>
^ asInstanceOf isInstanceOf toByte toChar
toDouble toFloat toInt toLong toShort
toString unary_+ unary_- unary_~ |

scala> t.to toByte toChar toDouble toFloat toInt toLong toShort
toString
~~~

Python does not provide an out of the box tab completion solution like Scala but you can use either `rlcompleter` module or use iPython

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> a = "astring"
>>> a.
a.__add__(                        a.decode(
a.__class__(                      a.encode(
a.__contains__(                   a.endswith(
a.__delattr__(                    a.expandtabs(
a.__doc__                         a.find(
a.__eq__(                         a.format(
```

Scala REPL Power Mode

Scala REPL power mode can be invoked by `:power` command.

```
scala> :power
** Power User mode enabled - BEEP BOOP WHIR **
** scala.tools.nsc._ has been imported      **
** global._ and definitions._ also imported **
** New vals! Try repl, intp, global, power  **
** New cmds! :help to discover them         **
** New defs! Type power.<tab> to reveal     **
```

First of all, you can see above that power mode adds new variables (val type), command options and defintions. For instance if you type `:help` before and after power mode activation you can see following additional cammand
~~~

```
:dump                        displays a view of the interpreter's internal
   state
:phase <phase>               set the implicit phase for power commands
:wrap <method>             * name of method to wrap around each repl line
```

`:wrap` is more like a Python decorator. You can define a custom method to wrap around each REPL line. For instance, let say you want to profile each line on REPL by calculating it's run time. Define a custom method which takes the code body and execute it and return the run-time (credits).

```
def timed[T](body: => T): T = {
   val start = System.nanoTime
   try body
   finally println((System.nanoTime - start) + " nanos elapsed.")
}
```

On REPL,

```
scala> def timed[T](body: => T): T = {
     |
     | val start = System.nanoTime
     |
     | try body
     |
     | finally println((System.nanoTime - start) + " nanos elapsed.")
     |
     | }
timed: [T](body: => T)T
scala> :wrap timed
Set wrapper to 'timed'

scala> (1 to 10000000).sum
755108174 nanos elapsed.
res2: Int = -2004260032

scala> (1 to 20000).sum
1410095 nanos elapsed.
res3: Int = 200010000

scala>  1733811000 / 3916300
602312 nanos elapsed.
res4: Int = 442
```

Scala Script Mode vs Compile Mode

Rather than typing on REPL you can also write your Scala code on a file, either in script mode (Scala Script) or compile mode (Scala Program). In script mode Scala files do not have an explicit main method. In both modes Scala files have an extension `.scala`. You can also run a Scala file as a shell or bat script if appropriate header is added, for instance in Unix `script.sh`

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello world "+ args(0) +"!")
```

and for Windows `script.bat` ~~~ ::#! **echo?** off call scala %0 %* goto :eof ::!# println("Hello world"+ args(0) +"!") ~~~

Note `$0` or `%0` is set to the name of the file, `$@` or `%*` is set to the positional parameters.

Loading & Running Scala Script in REPL

Loading a Scala script in REPL is quite straight foreword. Create a Scala script file `test.scala` and add some code to it ~~~ //test.scala println("Hell world"*3) ~~~ On REPL, `:load script.scala arg1 arg2` ~~~ scala> :load test.scala Loading test.scala… Hell world Hell world Hell world ~~~

Again it is quite similar to `execfile("script.py arg1 arg2")` in Python interactive shell. ~~~ »> execfile("test.py") Hello world Hello world Hello world ~~~

Alternatively, you can open the Python interpreter by typing `python -i script.py` on your terminal. It will execute `script.py` and then drop you into interactive mode with the environment left behind by `script.py`. Moreover you can also add `#!/usr/bin/python -i` directly into start of your Python scripts.

Running Scala Script on Terminal

You can run you Scala script as `scala script.scala arg1 arg2` ~~~ $scala test.scala Hell world Hell world Hell world ~~~ Note command-line arguments can be accessed in your script with the `argv` variable which is an Array[String]. ~~~ //test.scala println("Hello world"+argv(0)) ~~~

```
$scala test.scala Abhishek
Hello world Abhishek
```

In Python you can access the command-line arguments using `sys.argv` which also includes name of Python script file as first argument. ~~~ import sys print "Hello world" + sys.argv[1] print "File name:" + sys.argv[0] ~~~

```
$ python test.py Abhishek
Hello world Abhishek
File name:test.py
```

Compile and Execute Scala Program

Use `scalac` command to compiles one (or more) Scala program file(s) to generates Java bytecode which can be executed on any standard JVM. To compile using `scalac` source files must contain one or more class, trait, or object definitions.

```
$scalac File1.scala File2.scala
```

Note that complilation using `scalac` is very slow due to overhead related to enormous start-up cost every time you run compiler. To speed up this, standard Scala distribution ships with a compiler daemon called `fsc` (for fast Scala compiler) which you can run like,

```
$fsc File1.scala File2.scala
```

When you comiple with `fsc` first time it will take longer due to delay in starting the daemon but after that the compliation will be quite fast as daemon will already be running in background.

To compile and execute a Scala program, you must supply the name of a standalone singleton object with a main method that takes one parameter, an `Array[String]`. Singleton object with a main method acts as the entry point.

```
//HelloWorld.scala
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello world "+args(0))
    }
}
```

Obvisouly you can simply run it as a script,

```
$scala HelloWorld.scala Abhishek
```

or better compile and execute it,

```
$scalac HelloWorld.scala
$scala HelloWorld Abhishek
Hello world Abhishek
```

Compiling `HelloWorld.scala` will generate `HelloWorld.class` file which can be executed using `scala` command. Note that bytecode file `HelloWorld.class` corresponds to object defintion of same name. As best practise you should always use same source file name as object name.

Corresponding shell script version `HelloWorld.sh` for Unix which you can run from terminal as `./HelloWorld.sh Abhishek` if script has executable permission. ~~~ #!/bin/sh exec scala "$0" "$@" !# object HelloWorld { def main(args: Array[String]): Unit = { println("Hello world"+args(0)) } } HelloWorld.main(args) ~~~