JustEnoughScalaForSpark

October 21, 2017

```
In [1]: println("Hello World!")
```

```
Hello World!
```

```
In [2]: new java.io.File("/data").list
```

```
Out[2]: Array(shakespeare)
```

1 Just Enough Scala for Spark

Dean Wampler, Ph.D. [@deanwampler](http://twitter.com/deanwampler) (email)

Welcome. This notebook teaches you the core concepts of Scala necessary to use Apache Spark's Scala API effectively. Spark does a nice job exploiting the nicest features of Scala, while avoiding most of the more difficult and obscure features.

1.1 Introduction: Why Scala?

Spark lets you use Scala, Java, Python, R, and SQL to do your work. Scala and Java appeal to *data engineers*, who do the heavy lifting of building resilient and scalable infrastructures for *Big Data*. Python, R, and SQL appeal to *data scientists*, who build models for analyzing data, including machine learning, as well as explore data interactively, where SQL is very convenient.

These aren't hard boundaries. Many people do both roles. Many data engineers like Python and may use SQL and R. Many data scientists have decided to use Scala with Spark.

Briefly, some of the advantages of using Scala include the following: * **Performance:** Since Spark is written in Scala, you get the best performance and the most complete API coverage when you use Scala. It's true that with DataFrames, code written in all five languages performs about the same. If you need to use the RDD API, then Scala provides the best performance, with Java a close second. * **Debugging:** When runtime problems occur, understanding the exception stack frames and other debug information is easiest if you know Scala. Unfortunately, the "abstraction leaks" when problems occur. * **Concise, Expressive Code:** Compared to Java, Scala code is much more concise and several features of Scala make your code even more concise. This elevates your productivity and makes it easier to imagine a design approach and then write it down without having to translate the idea to a less flexible API that reflects idiomatic language constraints. (You'll see this in action as we go.) * **Type Safety:** Compared to Python and R, Scala code benefits from *static typing* with *type inference*. *Static typing* means that the Scala parser finds more errors in your expressions at compile time, when they don't match expected types, rather than discovering the problem later at run time. However, *type inference* means you don't have to add a lot of explicit type information to you code. In most cases, Scala will infer the correct types for you.

1.1.1 Why Not Scala?

Scala isn't perfect. There are two disadvantages compared to Python and R: * **Libraries:** Python and R have a rich ecosystem of data analytics libraries. While the picture is improving for Scala, Python and R are still well ahead. * **Advanced Language Features:** Mastering advanced language features gives you a lot of power to exploit, but if you don't understand those features, they can get in your way when you're just trying to get work done. Scala has some sophisticated constructs, especially in its *type system*. Fortunately, Spark mostly hides the advanced constructs.

1.1.2 For More on Scala

I can only scratch the surface of Scala here. We'll "sketch" concepts without too much depth. You'll learn enough to make use of them, but eventually you'll want to deepen your understanding.

When you need more information, consider these resources:

- Programming Scala, Second Edition: My comprehensive introduction to Scala.
- Scala Language Website: Where to download Scala, find documentation (e.g., the Scaladocs: Scala library documentation, like Javadocs), and other information.
- Lightbend Scala Services training, consulting, and support for Scala.
- Lightbend Fast Data Platform: Our new distribution for *Fast Data* (stream processing), including Spark, Flink, Kafka, Akka Streams, Kafka Streams, HDFS, and our production management and monitoring tools, running on Mesosphere DC/OS.

For now, I recommend that you open the Scaladocs for Scala and for Spark's Scala API. Clicking these two links will open them in new browser tabs: * Scaladocs for Scala. * Scaladocs for Spark.

> **Tips for using Scaladocs:** * Use the search bar in the upper-left-hand side to find a particular *type*. (For example, try "RDD" in the Spark Scaladocs.) * To search for a particular *method*, click the character under the search box for the method name's first letter, then scroll to it.

1.1.3 Prerequisites

I'll assume some prior programming experience in any language. Some familiarity with Java is assumed, but if you don't know Java, you should be able to search for explanations for anything unfamiliar.

This isn't an introduction to Spark itself. Some prior exposure to Spark is helpful, but I'll briefly explain most Spark concepts we'll encounter, too.

Throughout, you'll find links to more information on important topics.

1.2 About Notebooks

You're using the Jupyter All Spark Notebook Docker image. As described in the GitHub README you import this notebook into Jupyter running as a Docker image.

Notebooks let you mix documentation, like this Markdown "cell", with cells that contain code, graphs of results, etc. The metaphor is a physical notebook a scientist or student might use while working in a laboratory.

The menus and toolbar at the top provide options for evaluating a cell, adding and deleting cells, etc. You'll want to learn keyboard shortcuts if you use notebooks a lot.

Tips:

Invoke the *Help* > *Keyboard Shortcuts* menu item, then capture the page as an image (it's a modal dialog, unfortunately). Learn a few shortcuts each day.

For now, just know that you can click into any cell to move the focus. When you're in a cell, `shift+enter` evaluates the cell (parses and renders the Markdown or runs the code), then moves to the next cell. Try it for a few cells. I'll wait...

Okay. It's particularly nice that you can edit a cell you've already evaluated and rerun it. This is great when you're experimenting with code.

1.2.1 The Environment

Let's configure the environment to always show us the types of expressions.

```
In [3]: %showTypes on
```

```
Types will be printed.
```

When you start this notebook, the Jupyter Spark plugin creates a SparkContext for you. This is the entry point of any Spark application (even when you use the newer `SparkSession`. It knows how to connect to your cluster (or run locally in the same JVM), how to configure properties, etc. It also runs a Web UI that lets you monitor your running jobs. The instance of `SparkContext` is called `sc`. The next cell simply confirms that it exists.

```
In [4]: sc
```

```
Out[4]: org.apache.spark.SparkContext = org.apache.spark.SparkContext@4950e073
```

Here are few useful bits of information:

```
In [5]: println("Spark version:      " + sc.version)
        println("Spark master:       " + sc.master)
        println("Running 'locally'?: " + sc.isLocal)
```

```
Spark version:      2.2.0
Spark master:       local[*]
Running 'locally'?: true
```

1.3 Let's Load Some Data (and Start Learning Scala)

We're going to write real Spark programs and use them as vehicles for learning Scala and how to use it with Spark.

But first, we need to set up some text files we'll use, which contain some of the plays of Shakespeare. The next few cells define some helper methods (functions) to do this and then perform the steps. We'll start learning Scala concepts as we go.

Note: "method" vs. "function"

Scala follows a common object-oriented convention where the term *method* is used for a function that's attached to a class or instance. Unlike Java, at least before Java 8, Scala also has *functions* that are not associated with a particular class or instance.

In our next code example, we'll define a few helper *methods* for printing information, but you won't see a class definition here. So, what class is associated with these methods? When you use Scala in a notebook, you're actually using the Scala interpreter, which wraps any expressions and definitions we write into a hidden, generated class. The interpreter has to do this in order to generate valid JVM byte code.

Unfortunately, it can be a bit confusing when to use a method vs. a function, reflecting Scala's hybrid nature as an object-oriented and a functional language. Fortunately, in many cases, we can use methods and functions interchangably, so we won't worry about the distinction too much from now on.

We're defining methods now. We'll see what a real *function* looks like soon.

Okay, here are two convenience methods for printing either an error message or a simple "information" message. We'll explain the syntax in a subsequent cell below.

```
In [6]: /*
         * "info" takes a single String argument, prints it on a line,
         * and returns it.
         */
        def info(message: String): String = {
            println(message)

            // The last expression in the block, message, is the return value.
            // "return" keyword not required.
            // Do no additional formatting for the return string.
            message
        }
```

```
In [7]: /*
         * "error" takes a single String argument, prints a formatted error message,
         * and returns the message.
         */
        def error(message: String): String = {

            // Print the string passed to "println" and add a linefeed ("ln"):
            // See the next cell for an explanation of how the string is constructed.
            val fullMessage = s"""
                |*********************************************************************
                |
                |  ERROR: $message
                |
                |*********************************************************************
            |""".stripMargin
        println(fullMessage)

        fullMessage
    }
```

Let's try them.

```
In [8]: val infoString = info("All is well.")

All is well.


In [9]: val errorString = error("Uh oh...")


**********************************************************************

  ERROR: Uh oh...

**********************************************************************



In [10]: errorString

Out[10]: String = "
         **********************************************************************

           ERROR: Uh oh...

         **********************************************************************
         "
```

Method definitions have the following elements, in order: * The `def` keyword. * The method's name (`error` and `info` here). * The argument list in parentheses. If there are no arguments, the empty parentheses can be omitted. This is common for `toString` and "getter"-like methods that simply return a field in an instance, etc. * A colon followed by the type of the value returned by the method. This can often be inferred by Scala, so it's optional, but recommended for readibility by users! * An = (equals) sign that separates the method *signature* from the *body*. * The body in braces `{ ... }`, although if the body consists of a single expression, the braces are optional. * The last expression in the body is used as the return value. The `return` keyword is optional and rarely used. * Semicolons (`;`) are inferred at the end of lines (in most cases) and rarely used.

Look at the argument list for `error`. It is `(message: String)`, where `message` is the argument name and its type is `String`. This convention for *type annotations*, `name: Type`, is also used for the return type, `error(...): String`. Type annotations are required by Scala for method arguments. They are optional in most cases for the return type. We'll see that Scala can infer the types of many expressions and variable declarations.

Scala uses the same comment conventions as Java, `// ...` for a single line, and `/* ... */` for a comment block.

Note: Expression vs. Statement

An *expression* has a value, while a *statement* does not. Hence, when we assign an expression to a variable, the value the expression returns is assigned to the variable.

Inside `error`, we used a combination *interpolated* and *triple-quoted* string with the syntax `s"""..."""`: * **Triple-quoted string:** `"""..."""`. Useful for embedding newlines, like we did inside `error`. (We'll see another benefit later.) * **String interpolation:** Invoked by putting `s` before the string, e.g., `s"..."` or `s"""..."""`. Lets us embed variable references and expressions, where the string conversion will be inserted automatically. For example:

```
In [11]: s"""Use braces for expressions: ${sc.version}.
         You can omit the braces when just using a variable: $sc
         However, watch for ambiguities like ${sc}andextrastuff"""

Out[11]: String = Use braces for expressions: 2.2.0.
         You can omit the braces when just using a variable: org.apache.spark.SparkContext@4950e
         However, watch for ambiguities like org.apache.spark.SparkContext@4950e073andextrastuff
```

Another feature we're using for triple-quoted strings is the ability to strip the leading whitespace off each line. The `stripMargin` method removes all whitespace before and including the `|`. This lets you indent those lines for proper code formatting, but not have that whitespace remain in the string. In the following example, the resulting string has blank lines at the beginning and end. Note what happens with whitespace before `line2` and `line3` when the full string is printed:

```
In [12]: s"""
            |line 1
            |  line 2
            |  |  line 3
            |""".stripMargin

Out[12]: String = "
         line 1
           line 2
           line 3
         "
```

Character "literals" are specified single quotes, '/', while strings use double quotes, "/".

```
In [13]: '/'

Out[13]: Char = /

In [14]: "/"

Out[14]: String = /
```

1.3.1 Mutable Variables vs. Immutable Values

See how how to declare an immutable value before with `val`. Let's explore this a bit more: * `val immutableValue = ...`: Once initialized, we can't assign a *different* value to `immutableValue`. * `var mutableVariable = ...`: We can assign new values to `mutableVariable` as often as we want.

It's *highly recommended* that you only use `val`s unless you have a good reason for needing mutability, which is a very common source of bugs!!

> A `val immutableValue` could point to an instance that itself *is* mutable, e.g., an Array. In this case, while we can't assign a new array to `immutableValue`, we can change elements within the array! Put another way, immutability isn't *transitive*.

1.3.2 Setup the Files

The notebook already has the data files we need, several of Shakespeare's plays. They are in the `/data/shakespeare` subdirectory in the container (`./data/shakespeare` in the git project). There is one file for each play.

We'll write some Scala code to verify they are there, primarily so we can learn some more Scala.

Many of the types used in Scala code are from Java's library (JDK). Because Scala compiles to JVM byte code, you can use any Java library you want from Scala. We've been using java.lang.String. Now we'll use java.io.File to work with files and directories. As before, we'll use comments to explain a few other new Scala constructs.

```
In [18]: // Import File. Unlike Java, the semicolon ';' is not required.
         import java.io.File
```

Here the the directory where the files should be located.

```
In [19]: val shakespeare = new File("/data/shakespeare")
```

Scala's `if` construct is actually an expression (in Java they are *statements*). The `if` expression will return `true` or `false` and assign it to `success`, which we'll use in a moment.

```
In [20]: val success = if (shakespeare.exists == false) {   // doesn't exist already?
             error(s"Data directory path doesn't exist! $shakespeare")  // ignore returned strin
             false
         } else {
             info(s"$shakespeare exists")
             true
         }
         println("success = " + success)
```

```
/data/shakespeare exists
success = true
```

Now lets verify the files we expect are there, again to learn some more Scala.

```
In [21]: val pathSeparator = File.separator
         val targetDirName = shakespeare.toString
         val plays = Seq(
             "tamingoftheshrew", "comedyoferrors", "loveslabourslost", "midsummersnightsdream",
             "merrywivesofwindsor", "muchadoaboutnothing", "asyoulikeit", "twelfthnight")

         if (success) {
             println(s"Checking that the plays are in $shakespeare:")
             val failures = for {
                 play <- plays
                 playFileName = targetDirName + pathSeparator + play
                 playFile = new File(playFileName)
                 if (playFile.exists == false)
             } yield {
                 s"$playFileName:\tNOT FOUND!"
             }

             println("Finished!")
             if (failures.size == 0) {
                 info("All plays found!")
             } else {
                 println("The following expected plays were not found:")
                 failures.foreach(play => error(play))
             }
         }
```

```
Checking that the plays are in /data/shakespeare:
Finished!
All plays found!
```

```
Out[21]: Any = All plays found!
```

I'm using a so-called `for` *comprehension*. They are *expressions*, not *statements* like Java's `for` loops. They have the form:

```
for {
  play <- plays
  ...
} yield { block_of_final_expressions }
```

We iterate through a collection, `plays`, and assign each one to the `play` variable (actually an immutable value for each pass through the loop).

After assigning to `play`, subsequent steps in the `for` comprehension use it. First, a java.io.File instance, `playFile`, is created. Then, `playFile` is used to evaluate a conditional - does the file already exist? (It should!)

If the file already exists, the conditional returns `false`, which short-circuits the loop and goes to the next `play` in the list. If the file doesn't exit, the `yield` keyword tells Scala that I want to use the expression that follows to construct a new element, an *interpolated* string, for the missing play. From those returned elements, zero or more, a new collection is constructed. The final `if` block determines if the new collection has zero elements (expected), then prints an `info` message. If there were missing files, an `error` message is printed for each one of them.

1.4 Passing Functions as Arguments

Note how we printed the returned `successes` collection of strings. The idiom `collection.foreach(println)` is handy for looping over the elements and printing them, one per line. But how exactly does this work? (We'll use `plays` instead of `failures`, because the latter should be empty!)

```
In [22]: println("Pass println as the function to use for each element:")
         plays.foreach(println)

         println("\nUsing an anonymous function that calls println: `str => println(str)`")
         println("(Note that the type of the argument `str` is inferred to be String.)")
         plays.foreach(str => println(str))

         println("\nAdding the argument type explicitly. Note that the parentheses are required.
         plays.foreach((str: String) => println(str))

         println("\nWhy do we need to name this argument? Scala lets us use _ as a placeholder."
         plays.foreach(println(_))

         println("\nFor longer functions, you can use {...} instead of (...).")
         println("Why? Because it gives you the familiar multiline block syntax with {...}")
         plays.foreach {
           (str: String) => println(str)
         }

         println("\nThe _ placeholder can be used *once* for each argument in the list.")
         println("As an assume, use `reduceLeft` to sum some integers.")
         val integers = 0 to 10   // Return a "range" from 0 to 10, inclusive
         integers.reduceLeft((i,j) => i+j)
         integers.reduceLeft(_+_)
```

```
Pass println as the function to use for each element:
tamingoftheshrew
comedyoferrors
loveslabourslost
midsummersnightsdream
merrywivesofwindsor
muchadoaboutnothing
asyoulikeit
twelfthnight

Using an anonymous function that calls println: `str => println(str)`
(Note that the type of the argument `str` is inferred to be String.)
```

```
tamingoftheshrew
comedyoferrors
loveslabourslost
midsummersnightsdream
merrywivesofwindsor
muchadoaboutnothing
asyoulikeit
twelfthnight

Adding the argument type explicitly. Note that the parentheses are required.
tamingoftheshrew
comedyoferrors
loveslabourslost
midsummersnightsdream
merrywivesofwindsor
muchadoaboutnothing
asyoulikeit
twelfthnight

Why do we need to name this argument? Scala lets us use _ as a placeholder.
tamingoftheshrew
comedyoferrors
loveslabourslost
midsummersnightsdream
merrywivesofwindsor
muchadoaboutnothing
asyoulikeit
twelfthnight

For longer functions, you can use {...} instead of (...).
Why? Because it gives you the familiar multiline block syntax with {...}
tamingoftheshrew
comedyoferrors
loveslabourslost
midsummersnightsdream
merrywivesofwindsor
muchadoaboutnothing
asyoulikeit
twelfthnight

The _ placeholder can be used *once* for each argument in the list.
As an assume, use `reduceLeft` to sum some integers.
```

```
Out[22]: Int = 55
```

2 Our First Spark Program

Whew! We've learned a lot of Scala already while doing typical data science chores (i.e., fetching data). Now let's implement a real algorithm using Spark, *Inverted Index*.

2.1 Inverted Index - When You're Tired of Counting Words...

You'll want use *Inverted Index* when you create your next "Google killer". It takes in a corpus of documents (e.g., web pages), tokenizes the words, and outputs for each word a list of the documents that contain it, along with the corresponding counts.

This is a slightly more interesting algorithm than *Word Count*, the classic "hello world" program everyone implements when they learn Spark.

The term *inverted* here means we start with the words as part of the input *values*, while the *keys* are the document identifiers, and we'll switch ("invert") to using the words as keys and the document identifiers as values.

Here's our first version, all at once. This is *one, long expression*. Note the periods `.` at the end of the subexpressions.

```
In [23]: val iiFirstPass1 = sc.wholeTextFiles(shakespeare.toString).
             flatMap { location_contents_tuple2 =>
                 val words = location_contents_tuple2._2.split("""\W+""")
                 val fileName = location_contents_tuple2._1.split(pathSeparator).last
                 words.map(word => ((word, fileName), 1))
             }.
             reduceByKey((count1, count2) => count1 + count2).
             map { word_file_count_tup3 =>
                 (word_file_count_tup3._1._1, (word_file_count_tup3._1._2, word_file_count_tup3.
             }.
             groupByKey.
             sortByKey(ascending = true).
             mapValues { iterable =>
                 val vect = iterable.toVector.sortBy { file_count_tup2 =>
                     (-file_count_tup2._2, file_count_tup2._1)
                 }
                 vect.mkString(",")
             }
```

Now let's break it down into steps, assigning each step to a variable. This extra verbosity let's us see what Scala infers for the type returned by each expression, helping us learn.

This is one of the nice features of Scala. We don't have to put in the type information ourselves, most of the time, like we would have to do for Java code. Instead, we let the compiler give us feedback about what we just created. This is especially useful when you're learning a new API, like Spark's.

```
In [24]: val fileContents = sc.wholeTextFiles(shakespeare.toString)
         fileContents   // force the notebook to print the type.

Out[24]: org.apache.spark.rdd.RDD[(String, String)] = /data/shakespeare MapPartitionsRDD[11] at
```

The second line, with `fileContents` by itself, is there so the notebook will show us its type information. (Try to remove it and re-evaluate the cell. Nothing is printed.).

The output is telling us that `fileContents` has the type RDD[(String,String)], but `RDD` is a base class and the actual instance is a `MapPartitionsRDD`, which is a "private" implementation subclass of `RDD`.

A name followed by square brackets, `[...]`, means that `RDD[...]` requires one or more type parameters in the brackets. In this case, a single type parameter, which represents the type of the records held by the `RDD`.

The single type parameter is given by `(String,String)`, which is a convenient shorthand for Tuple2[String,String]. That is, we have two-element *tuples* as records, where the first element is a `String` for a file's fully-qualified path and the second element is a `String` for the contents of that file. This is what `SparkContext.wholeTextFiles` returns for us. We'll use the path to remember where we found words, while the contents contains the words themselves (of course).

To recap, the following two types are equivalent: * `RDD[(String,String)]` - Note parentheses nested in brackets, `[(...)]`. * `RDD[Tuple2[String,String]]` - Note nested brackets `[...[...]]`, not `[(...)]`.

We'll see shortly that you can also write *instances* of Tuple2[T1,T2] with the same syntax, e.g., `("foo", 101)`, for a `(String,Int)` tuple, and similarly for *higher-arity* tuples (up to 22 elements...), e.g., `("foo", 101, 3.14159, ("bar", 202L))`. Run the next cell to see the type signature for this last tuple.

```
In [25]: ("foo", 101, 3.14159, ("bar", 202L))

Out[25]: (String, Int, Double, (String, Long)) = (foo,101,3.14159,(bar,202))
```

Do you understand it? Do you see that it's a four-element tuple and not a five-element tuple? This is because the `("bar", 202L)` is a nested tuple. It's the fourth element of the outer tuple.

Exercise: Try creating some more tuples with elements of different types. Use the next cell.

```
In [26]: (1,2)

Out[26]: (Int, Int) = (1,2)
```

How many `fileContents` records do we have? Not many. It should be the same number as the number of files we downloaded above.

```
In [27]: fileContents.count

Out[27]: Long = 8
```

> **NOTE:** We called the `RDD.count` method, whereas most Scala collections have a `size` method.

Now for our next step in the calculation. First, "tokenize" the contents into words by splitting on non-alphanumeric characters, meaning all runs of whitespace (including the newlines), punctuation, etc.

Next, the fully-qualified path is verbose and the same prefix is repeated for all the files, so let's extract just the last element of it, the unique file name.

Then form new tuples with the words and file names.

Note: This "tokenization" approach is very crude. It improperly handles contractions, like `it's` and hyphenated words like `world-changing`. When you kill Google, be sure to use a real *natural language processing* (NLP) tokenization technique.

```
In [28]: val wordFileNameOnes = fileContents.flatMap { location_contents_tuple2 =>
             // example input record: (file_path, "all the words in the file")
             // mytuple._2 => give me the 2nd element
             val words = location_contents_tuple2._2.split("""\W+""")
             // mytuple._1 => give me the 1st element
             val fileName = location_contents_tuple2._1.split(pathSeparator).last
             // create a new tuple to return. Note how we structured it!
             words.map(word => ((word, fileName), 1))
         }
         wordFileNameOnes
```

```
Out[28]: org.apache.spark.rdd.RDD[((String, String), Int)] = MapPartitionsRDD[12] at flatMap at
```

I find this hard to read and shortly I'll show you a much more elegant, alternative syntax.

Let's understand the difference between `map` and `flatMap`. If I called `fileContents.map`, it would return exactly *one* new record for each record in *fileContents*. What I actually want instead are new records for each word-fileName combination, a significantly larger number (but the data in each record will be much smaller).

Using `fileContents.flatMap` gives me what I want. Instead of returning one output record for each input record, a `flatMap` returns a *collection* of new records, zero or more, for *each* input record. These collections are then *flattened* into one big collection, another `RDD` in this case.

What should `flatMap` actually do with each record? I pass a *function* to define what to do. I'm using an unnamed or *anonymous* function. The syntax is `argument_list => body`:

```
location_contents_tuple2 =>
    val words = ...
    ...
}
```

I have a single argument, the record, which I named `location_contents_tuple2`, a verbose way to say that it's a two-element tuple with an input file's location and contents. I don't require a type parameter after `location_contents_tuple2`, because it's inferred by Scala. The => "arrow" separates the argument list from the body, which appears on the next few lines.

When a function takes more than one argument or you add explicit type *annotations* (e.g., `: (String,Int,Double)`), then you need parentheses. Here are three examples:

```
(some_tuple3: (String,Int,Double)) => ...
(arg1, arg2, arg3) => ...
(arg1: String, arg2: Int, arg3: Double) => ...
```

We're letting Scala infer the argument type in our case, `(String,String)`.

Wait, I said we're passing a function as an argument to `flatMap`. If so, why am I using braces `{...}` around this function argument instead of parentheses `(...)` like you would normally expect when passing arguments to a method like `flatMap`?

It's because Scala lets us substitute braces instead of parentheses so we have the familiar block-like syntax `{...}` we know and love for `if` and `for` expressions. I could use either braces or parentheses here. The convention in the Scala community is to use braces for a multi-line anonymous function and to use parentheses for a single expression when it fits on the same line.

Now, for each `location_contents_tuple2`, I access the *first* element using the `_1` method and the *second* element using `_2`.

The file `contents` is in the second element. I split it by calling Java's `String.split` method, which takes a *regular expression* string. Here I specify a regular expression for one or more, non-alphanumeric characters. `String.split` returns an `Array[String]` of the words.

```
val words = location_contents_tuple2._2.split("""\W+""")
```

For the first tuple element, I extract the file name at the end of the location path. This isn't necessary, but it makes the output more readable if I remove the long, common prefix from the path.

```
val fileName = location_contents_tuple2._1.split(pathSeparator).last
```

Finally, still inside the anonymous function passed to `flatMap`, I use Scala's `Array.map` (*not* `RDD.map`) to transform each `word` into a tuple of the form `((word, fileName), 1)`.

```
words.map(word => ((word, fileName), 1))
```

Why did I embed a tuple of `(word, fileName)` inside the "outer" tuple with a `1` as the second element? Why not just write a three-element tuple, `(word, fileName, 1)`? It's because I'll use the `(word, fileName)` as a *key* in the next step, where I'll find all unique word-fileName combinations (using the equivalent of a `group by` statement). So, using the nested `(word, fileName)` as my *key* is most convenient. The `1` *value* is a "seed" count, which I'll use to count the occurrences of the unique `(word, fileName)` pairs.

> **Notes:** * For historical reasons, tuple indices start at 1, not 0. Arrays and other Scala collections index from 0. * I said previously that *method* arguments have to be declared with types. That's usually *not* required for *function* arguments, as here. * Another benefit of triple-quoted strings that makes them nice for regular expressions is that you don't have to escape regular expression metacharacters, like `\W`. If I used a single-quoted string, I would have to write it as `"\\W+"`. Your choice...

Let's count the number of records we have and look at a few of the lines. We'll use the `RDD.take` method to grab the first 10 lines, then loop over them and print them.

```
In [29]: wordFileNameOnes.count
```

```
Out[29]: Long = 173336
```

```
In [30]: wordFileNameOnes.take(10).foreach(println)
```

```
((,merrywivesofwindsor),1)
((THE,merrywivesofwindsor),1)
((MERRY,merrywivesofwindsor),1)
((WIVES,merrywivesofwindsor),1)
```

```
((OF,merrywivesofwindsor),1)
((WINDSOR,merrywivesofwindsor),1)
((DRAMATIS,merrywivesofwindsor),1)
((PERSONAE,merrywivesofwindsor),1)
((SIR,merrywivesofwindsor),1)
((JOHN,merrywivesofwindsor),1)
```

We asked for results, so we forced Spark to run a job to compute results. Spark pipelines, like `iiFirstPass1` are *lazy*; nothing is computed until we ask for results.

When you're learning, it's useful to print some data to better understand what's happening. Just be aware of the extra overhead of running lots of Spark jobs.

The first record shown has "" (blank) as the word:

```
((,asyoulikeit),1)
```

Also, some words have all capital letters:

```
((DRAMATIS,asyoulikeit),1)
```

(You can see where these capitalized words occur if you look in the original files.) Later on, We'll filter out the blank-word records and use lower case for all words.

Now, let's join all the unique `(word,fileName)` pairs together.

```
In [31]: val uniques = wordFileNameOnes.reduceByKey((count1, count2) => count1 + count2)
         uniques

Out[31]: org.apache.spark.rdd.RDD[((String, String), Int)] = ShuffledRDD[13] at reduceByKey at <
```

In SQL you would use `GROUP BY` for this (including SQL queries you might write with Spark's DataFrame API). However, in the `RDD` API, this is too expensive for our needs, because we don't care about the groups themselves, the long list of repeated `(word,fileName)` pairs. We only care about how many elements are in each group, that is their *size*. That's the purpose of the `1` in the tuples and the use of `RDD.reduceByKey`. It brings together all records with the same key, the unique `(word,fileName)` pairs, and then applies the anonymous function to "reduce" the values, the `1`s. I simply sum them up to compute the group counts.

Note that the anonymous function `reduceByKey` expects must take two arguments, so I need parentheses around the argument list. Since this function fits on the same line, I used parentheses for `reduceByKey`, instead of braces.

> **Note:** All the `*ByKey` methods operate on two-element tuples and treat the first element as the key, by default.

How many are there? Let's see a few:

```
In [32]: uniques.count

Out[32]: Long = 27276
```

As you would expect from a `GROUP BY`-like statement, the number of records is smaller than before. There are about 1/6 as many records now, meaning that on average, each `(word,fileName)` combination appears 6 times.

```
In [33]: uniques.take(30).foreach(println)
```

```
((dexterity,merrywivesofwindsor),1)
((force,muchadoaboutnothing),2)
((whole,comedyoferrors),2)
((lamb,muchadoaboutnothing),2)
((blunt,tamingoftheshrew),3)
((letter,merrywivesofwindsor),19)
((crest,asyoulikeit),1)
((bestow,asyoulikeit),1)
((rear,midsummersnightsdream),1)
((crossing,tamingoftheshrew),1)
((wronged,merrywivesofwindsor),4)
((S,tamingoftheshrew),10)
((HIPPOLYTA,midsummersnightsdream),19)
((revolve,twelfthnight),1)
((er,merrywivesofwindsor),11)
((renown,asyoulikeit),1)
((cubiculo,twelfthnight),1)
((All,twelfthnight),3)
((power,loveslabourslost),8)
((Albeit,asyoulikeit),1)
((lips,tamingoftheshrew),3)
((upshot,twelfthnight),1)
((approach,midsummersnightsdream),4)
((mean,muchadoaboutnothing),5)
((embossed,asyoulikeit),1)
((varnish,loveslabourslost),2)
((Apollo,midsummersnightsdream),1)
((spangled,midsummersnightsdream),1)
((gentlemen,comedyoferrors),1)
((Rebuke,loveslabourslost),1)
```

For *inverted index*, we want our final keys to be the words themselves, so let's restructure the tuples from `((word,fileName),count)` to `(word,(fileName,count))`. Now, I'll still output two-element, key-value tuples, but the `word` will be the key and the `(fileName,count)` tuple will be the value.

```
In [34]: val words = uniques.map { word_file_count_tup3 =>
             (word_file_count_tup3._1._1, (word_file_count_tup3._1._2, word_file_count_tup3._2))
         }
```

The nested tuple methods, e.g., `_1._2`, are hard to read, making the logic somewhat obscure. We'll see a beautiful and elegant alternative shortly.

Now I'll use an actual `group by` operation, because I now need to retain the groups. Calling `RDD.groupByKey` uses the first tuple element, now just the `words`, to bring together all occurrences of the unique words. Next, I'll sort the result by word, ascending alphabetically.

```
In [35]: val wordGroups = words.groupByKey.sortByKey(ascending = true)
         wordGroups
```

```
Out[35]: org.apache.spark.rdd.RDD[(String, Iterable[(String, Int)])] = ShuffledRDD[18] at sortBy
```

Note that each group is actually a Scala Iterable, i.e., an abstraction for some sort of collection. (It's actually a Spark-defined, private collection type called a `CompactBuffer`.)

```
In [36]: wordGroups.count
```

```
Out[36]: Long = 11951
```

```
In [37]: wordGroups.take(30).foreach(println)
```

```
(,CompactBuffer((tamingoftheshrew,1), (asyoulikeit,1), (merrywivesofwindsor,1), (comedyoferrors,
(A,CompactBuffer((loveslabourslost,78), (midsummersnightsdream,39), (muchadoaboutnothing,31), (m
(ABOUT,CompactBuffer((muchadoaboutnothing,18)))
(ACT,CompactBuffer((asyoulikeit,22), (comedyoferrors,11), (tamingoftheshrew,12), (loveslabourslo
(ADAM,CompactBuffer((asyoulikeit,16)))
(ADO,CompactBuffer((muchadoaboutnothing,18)))
(ADRIANA,CompactBuffer((comedyoferrors,85)))
(ADRIANO,CompactBuffer((loveslabourslost,111)))
(AEGEON,CompactBuffer((comedyoferrors,20)))
(AEMELIA,CompactBuffer((comedyoferrors,16)))
(AEMILIA,CompactBuffer((comedyoferrors,3)))
(AEacides,CompactBuffer((tamingoftheshrew,1)))
(AEgeon,CompactBuffer((comedyoferrors,7)))
(AEgle,CompactBuffer((midsummersnightsdream,1)))
(AEmilia,CompactBuffer((comedyoferrors,4)))
(AEsculapius,CompactBuffer((merrywivesofwindsor,1)))
(AGUECHEEK,CompactBuffer((twelfthnight,2)))
(ALL,CompactBuffer((midsummersnightsdream,2), (tamingoftheshrew,2)))
(AMIENS,CompactBuffer((asyoulikeit,16)))
(ANDREW,CompactBuffer((twelfthnight,104)))
(ANGELO,CompactBuffer((comedyoferrors,36)))
(ANN,CompactBuffer((merrywivesofwindsor,1)))
(ANNE,CompactBuffer((merrywivesofwindsor,27)))
(ANTIPHOLUS,CompactBuffer((comedyoferrors,195)))
(ANTONIO,CompactBuffer((muchadoaboutnothing,32), (twelfthnight,32)))
(ARMADO,CompactBuffer((loveslabourslost,111)))
(AS,CompactBuffer((asyoulikeit,24)))
(AUDREY,CompactBuffer((asyoulikeit,18)))
(Abate,CompactBuffer((midsummersnightsdream,1), (loveslabourslost,1)))
(Abbess,CompactBuffer((comedyoferrors,2)))
```

Finally, let's clean up these `CompactBuffers`. Let's convert each to a Scala Vector (a collection with *O(1)* performance for most operations), then sort it *descending* by count, so the locations that mention the corresponding word the *most* appear *first* in the list. (Think about how you would want a search tool to work...)

Note we're using `Vector.sortBy`, not an `RDD` sorting method. It takes a function that accepts each collection element and returns something used to sort the collection. By returning `(-fileNameCountTuple2._2, fileNameCountTuple2)`, I effectively say, "sort by the counts *descending* first, then sort by the file names." Why does `-fileNameCountTuple2._2` cause counts to be sorted descending, because I'm returning the negative of the value, so larger counts will be less than smaller counts, e.g., `-3 < -2`.

Finally, I take the resulting `Vector` and make a comma-separated string with the elements, using the helper method `mkString`.

What's `RDD.mapValues`? I could use `RDD.map`, but I'm not changing the keys (the words), so rather than have to deal with the tuple with both elements, `mapValues` just passes in the value part of the tuple and reconstructs new `(key,value)` tuples with the new value that my function returns. So, `mapValues` is more convenient to use than `map` when I have two-element tuples and I'm not modifying the keys.

```
In [38]: val iiFirstPass2 = wordGroups.mapValues { iterable =>
             val vect = iterable.toVector.sortBy { file_count_tup2 =>
                 (-file_count_tup2._2, file_count_tup2._1)
             }
             vect.mkString(",")
         }
```

We're done! The number of records is the same as for `wordGroups` (do you understand why?), so let's just see see some of the records.

```
In [39]: iiFirstPass2.take(30).foreach(println)
```

```
(,(asyoulikeit,1),(comedyoferrors,1),(loveslabourslost,1),(merrywivesofwindsor,1),(midsummersnig
(A,(loveslabourslost,78),(tamingoftheshrew,59),(twelfthnight,47),(comedyoferrors,42),(midsummers
(ABOUT,(muchadoaboutnothing,18))
(ACT,(merrywivesofwindsor,23),(asyoulikeit,22),(twelfthnight,18),(muchadoaboutnothing,17),(tamin
(ADAM,(asyoulikeit,16))
(ADO,(muchadoaboutnothing,18))
(ADRIANA,(comedyoferrors,85))
(ADRIANO,(loveslabourslost,111))
(AEGEON,(comedyoferrors,20))
(AEMELIA,(comedyoferrors,16))
(AEMILIA,(comedyoferrors,3))
(AEacides,(tamingoftheshrew,1))
(AEgeon,(comedyoferrors,7))
(AEgle,(midsummersnightsdream,1))
(AEmilia,(comedyoferrors,4))
(AEsculapius,(merrywivesofwindsor,1))
(AGUECHEEK,(twelfthnight,2))
(ALL,(midsummersnightsdream,2),(tamingoftheshrew,2))
(AMIENS,(asyoulikeit,16))
```

```
(ANDREW,(twelfthnight,104))
(ANGELO,(comedyoferrors,36))
(ANN,(merrywivesofwindsor,1))
(ANNE,(merrywivesofwindsor,27))
(ANTIPHOLUS,(comedyoferrors,195))
(ANTONIO,(muchadoaboutnothing,32),(twelfthnight,32))
(ARMADO,(loveslabourslost,111))
(AS,(asyoulikeit,24))
(AUDREY,(asyoulikeit,18))
(Abate,(loveslabourslost,1),(midsummersnightsdream,1))
(Abbess,(comedyoferrors,2))
```

Okay. Looks reasonable.

Next, I'll refine the code using a very powerful feature, *pattern matching*, which both makes the code more concise and easier to understand. It's my *favorite* feature of Scala.

Before I do that, try a few refinements on your own.

Exercises:

- Add a filter statement to remove the first entry for the blank word "". You could do this one of two ways, using another "step" with RDD.filter (search the Scaladoc page, *or* using the similar Scala collections method, scala.collection.Seq.filter. Both versions take a *predicate* function, one that returns `true` if the record should be *retained* and `false` otherwise. Do you think one choice is better than the other? Why? Or, are they basically the same? Reasons might include code comprehension and performance of one over the other.
- Convert all words to lower case. Calling `toLowerCase` on a string is all you need. Where's a good place to insert this logic?

I'll implement both changes in subsequent refinements below.

> **NOTE:** If you would prefer to make a copy of the code in a new cell, use the *Insert* menu above to add cells. Or, learn another keyboard shortcut; `ESC` (escape key), followed by `A` for insert before or `B` for insert after. Then hit return to edit. Note the toolbar pop-down for setting the format of the cell. This cell you're reading is *Markdown*. Make sure to use *Code* for your source code cells.

2.2 Pattern Matching

We've studied a real program and we've learned quite a bit of Scala. Let's improve it with my favorite Scala feature, *pattern matching*.

Here's the "first pass" version again for easy reference.

```
In [40]: val iiFirstPass1b = sc.wholeTextFiles(shakespeare.toString).
             flatMap { location_contents_tuple2 =>
                 val words = location_contents_tuple2._2.split("""\W+""")
                 val fileName = location_contents_tuple2._1.split(pathSeparator).last
                 words.map(word => ((word, fileName), 1))
             }.
             reduceByKey((count1, count2) => count1 + count2).
             map { word_file_count_tup3 =>
                 (word_file_count_tup3._1._1, (word_file_count_tup3._1._2, word_file_count_tup3.
             }.
             groupByKey.
             sortByKey(ascending = true).
             mapValues { iterable =>
                 val vect = iterable.toVector.sortBy { file_count_tup2 =>
                     (-file_count_tup2._2, file_count_tup2._1)
                 }
                 vect.mkString(",")
             }
```

Now here is a new implementation that uses *pattern matching*.

I've also made two other additions, the solutions to the last exercises, which remove empty words "" and fix mixed capitalization, using the following additions: * `filter(word => word.size > 0)` to remove the empty words. (In Spark and Scala collections, `filter` has the positive sense; what should be retained?) It's indicated by the comment `// #1`. * `word.toLowerCase` to convert all words to lower case uniformly, so that words like HAMLET, Hamlet, and hamlet in the original texts are treated as the same, since we're counting word occurrences. See comment `// #2`.

```
In [41]: val ii1 = sc.wholeTextFiles(shakespeare.toString).
             flatMap {
                 case (location, contents) =>
                     val words = contents.split("""\W+""").
                         filter(word => word.size > 0)                      // #1
                     val fileName = location.split(pathSeparator).last
                     words.map(word => ((word.toLowerCase, fileName), 1))   // #2
             }.
             reduceByKey((count1, count2) => count1 + count2).
             map {
                 case ((word, fileName), count) => (word, (fileName, count))
             }.
             groupByKey.
             sortByKey(ascending = true).
             mapValues { iterable =>
                 val vect = iterable.toVector.sortBy {
                     case (fileName, count) => (-count, fileName)
                 }
                 vect.mkString(",")
             }
```

Compare with your exercise solutions above. I added the filtering inside the function passed to `flatMap`. My choice reduces the number of output records from `flatMap` by at most one record per input line, which shouldn't have a significant impact on performance. Filtering itself adds some extra overhead.

Also, the way Spark implements steps like `map`, `flatMap`, `filter`, it would incur about the same overhead if I used `RDD.filter` instead. Note that we could also do the filtering later in the pipeline, after `groupByKey`, for example. So, whichever approach you implemented above is probably fine. You could do performance profiling of the different approaches, but you may not notice a significance difference until you use very large input data sets.

Let's verify we still get reasonable results. Now I'll use Spark's DataFrame API for its convenient display options. `DataFrames` are part of Spark SQL.

First, we need to create an instance of SQLContext that we need to access these features.

```
In [42]: import org.apache.spark.sql.SQLContext
```

```
In [43]: val sqlContext = new SQLContext(sc)
```

Now, we convert the `RDD` to a `DataFrame` with `sqlContext.createDataFrame`, then use `toDF` (convert to another `DataFrame`?) with new names for each "column".

```
In [44]: val ii1DF = sqlContext.createDataFrame(ii1).toDF("word", "locations_counts")
```

The `%%dataframe` *cell magic* provides a nice table layout display.

```
In [45]: %%dataframe
         ii1DF
```

Okay, now let's explore the new implementation. I start off as before, by calling `wholeTextFiles`:

```
val ii = sc.wholeTextFiles(shakespeare.toString).
```

The function I pass to `flatMap` now looks like this:

```
flatMap {
    case (location, contents) =>
        val words = contents.split("""\W+""").
            filter(word => word.size > 0)                    // #1
        val fileName = location.split(pathSeparator).last
        words.map(word => ((word.toLowerCase, fileName), 1))   // #2
}.
```

Compare it to the previous version (ignoring the enhancements for blank words and capitalization, marked with the #1 and #2 comments):

```
flatMap { location_contents_tuple2 =>
    val words = location_contents_tuple2._2.split("""\W+""")
    val fileName = location_contents_tuple2._1.split(pathSeparator).last
    words.map(word => ((word, fileName), 1))
}.
```

Instead of `location_contents_tuple2` a variable name for the whole tuple, I wrote `case (location, contents)`. The `case` keyword says I want to *pattern match* on the object passed to the function. If it's a two-element tuple (and I know it always will be in this case), then *extract* the first element and assign it to a variable named `location` and extract the second element and assign it to a variable named `contents`.

Now, instead of accessing the location and content with the slighly obscure and verbose `location_contents_tuple2._1` and `location_contents_tuple2._2`, respectively, I use meaningful names, `location` and `contents`. The code becomes more concise and more readable.

I'll explore more pattern matching features below.

The `reduceByKey` step is unchanged:

```
reduceByKey((count1, count2) => count1 + count2).
```

To be clear, this isn't a pattern-matching expression; there is no `case` keyword. It's just a "regular" function that takes two arguments, for the two things I'm adding.

My favorite improvement is the next line:

```
map {
    case ((word, fileName), count) => (word, (fileName, count))
}.
```

Compare it to the previous, obscure version:

```
map { word_file_count_tup3 =>
    (word_file_count_tup3._1._1, (word_file_count_tup3._1._2, word_file_count_tup3._2))
}.
```

The new implementation makes it clear what I'm doing; just shifting parentheses! That's all it takes to go from the `(word, fileName)` keys with `count` values to `word` keys and `(fileName, count)` values. Note that pattern matching works just fine with nested structures, like `((word, fileName), count)`.

I hope you can appreciate how elegant and concise this expression is! Note how I thought of the next transformation I needed to do in preparation for the final group-by, to switch from `((word, fileName), count)` to `(word, (fileName, count))` and *I just wrote it down exactly as I pictured it!*

Code like this makes writing Scala Spark code a sublime experience for me. I hope it will for you, too ;)

The next two expressions are unchanged:

```
groupByKey.
sortByKey(ascending = true).
```

The final `mapValues` now uses pattern matching to sort the `Vector` in each record:

```
mapValues { iterable =>
    val vect = iterable.toVector.sortBy {
        case (fileName, count) => (-count, fileName)
    }
    vect.mkString(",")
}
```

Compared to the original version, it's again easier to read:

```
mapValues { iterable =>
    val vect = iterable.toVector.sortBy { file_count_tup2 =>
        (-file_count_tup2._2, file_count_tup2._1)
    }
    vect.mkString(",")
}
```

The function I pass to `sortBy` returns a tuple used for sorting, with `-count` to force *descending* numerical sort (biggest first) and `fileName` to secondarily sort by the file name, for equivalent counts. I could ignore file name order and just return `-count` (not a tuple). However, if you need more repeatable output in a distributed system like Spark, say for example to use in unit test validation, then the secondary sorting by file name is handy.

2.3 Our Final Version: Supporting SQL Queries

To play with some more Spark, let's write SQL queries to explore the resulting data.

To do this, let's first refine the output. Instead of creating a string for the list of `(location,count)` pairs, which is opaque to our SQL schema (i.e., just a string), let's "unzip" the collection into two arrays, one for the `locations` and one for the `counts`. That way, if we ask for the first element of each array, we'll have nicely separate fields that work better with Spark SQL queries.

"Zipping" and "unzipping" work like a mechanical zipper. If I have a collection of tuples, say `List[(String, Int)]`, I convert this single collection of "zippered" values into two collections (in a tuple) of single values, `(List[String], List[Int])`. Zipping is the inverse operation.

Here is our final implementation, `ii1` rewritten with this change.

```
In [46]: val ii = sc.wholeTextFiles(shakespeare.toString).
             flatMap {
                 case (location, contents) =>
                     val words = contents.split("""\W+""").
                         filter(word => word.size > 0)                  // #1
                     val fileName = location.split(pathSeparator).last
                     words.map(word => ((word.toLowerCase, fileName), 1))   // #2
             }.
             reduceByKey((count1, count2) => count1 + count2).
             map {
                 case ((word, fileName), count) => (word, (fileName, count))
             }.
             groupByKey.
             sortByKey(ascending = true).
             map {                        // Must use map now, because we'll format new records
               case (word, iterable) =>   // Hence, pattern match on the whole input record.

                 val vect = iterable.toVector.sortBy {
                     case (fileName, count) => (-count, fileName)
                 }

                 // Use `Vector.unzip`, which returns a single, two element tuple, where each
                 // element is a collection, one for the locations and one for the counts.
                 // I use pattern matching to extract these two collections into variables.
                 val (locations, counts) = vect.unzip

                 // Lastly, I'll compute the total count across all locations and return
                 // a new record with all four fields. The `reduceLeft` method takes a function
                 // that knows how to "reduce" the collection down to a final value, working
                 // from the left.
                 val totalCount = counts.reduceLeft((n1,n2) => n1+n2)

                 (word, totalCount, locations, counts)
             }
```

We've changed the ending `mapValues` call to a `map` call, because we'll construct entirely new records, not just new values with the same keys. Hence the full records, two-element tuples are passed in, rather than just the values, so we'll pattern match on the tuple:

```
map {                            // Must use map now, because we'll format new records.
  case (word, iterable) =>       // Hence, pattern match on the whole input record.

    val vect = iterable.toVector.sortBy {
        case (fileName, count) => (-count, fileName)
    }
```

We have a `Vector[String, Int]` of two-element tuples `(fileName, count)`. We use `Vector.unzip` to create a single, two element tuple, where each element is now a collection, one for the locations and one for the counts. The type is `(Vector[String], Vector[Int])`.

We can also use pattern matching with assignment! We immediately decompose the two-element tuple:

```
    // I use pattern matching to extract these two collections into variables.
    val (locations, counts) = vect.unzip
```

Finally, it's convenient to know how many locations and counts we have, so we'll compute another new column for the their count and format a four-element tuple as the final output.

```
    // Lastly, I'll compute the total count across all locations and return
    // a new record with all four fields. The `reduceLeft` method takes a function
    // that knows how to "reduce" the collection down to a final value, working
    // from the left.
    val totalCount = counts.reduceLeft((n1,n2) => n1+n2)

    (word, totalCount, locations, counts)
}
```

Okay! Now let's create a DataFrame with this data. The `toDF` method just returns the same `DataFrame`, but with appropriate names for the columns, instead of the synthesized names that `createDataFrame` generates (e.g., `_c1`, `_c2`, etc.)

Caching the `DataFrame` in memory prevents Spark from recomputing `ii` from the input files *every time* I write a query!

Finally, to use SQL, I need to "register" a temporary table.

```
In [47]: val iiDF = sqlContext.createDataFrame(ii).toDF("word", "total_count", "locations", "cou
         iiDF.cache
         iiDF.registerTempTable("inverted_index")
```

Let's remind ourselves of the schema:

```
In [48]: iiDF.printSchema
```

```
root
 |-- word: string (nullable = true)
 |-- total_count: integer (nullable = false)
 |-- locations: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- counts: array (nullable = true)
 |    |-- element: integer (containsNull = false)
```

The following SQL query extracts the top location by count for each word, as well as the total count across all locations for the word. The Spark SQL dialect supports Hive SQL syntax for extracting elements from arrays, maps, and structs (details). Here I access the first element (index zero) from each array.

```
In [49]: %%SQL
         SELECT word, total_count, locations[0] AS top_location, counts[0] AS top_count
         FROM inverted_index
```

```
Out[49]: +-----------+-----------+----------------+---------+
         |       word|total_count|    top_location|top_count|
         +-----------+-----------+----------------+---------+
         |          a|       3350|loveslabourslost|      507|
         |    abandon|          6|      asyoulikeit|        4|
         |      abate|          3|loveslabourslost|        1|
         |  abatement|          1|     twelfthnight|        1|
         |     abbess|          8|   comedyoferrors|        8|
         |      abbey|          9|   comedyoferrors|        9|
         |abbominable|          1|loveslabourslost|        1|
         |abbreviated|          1|loveslabourslost|        1|
         |       abed|          2|      asyoulikeit|        1|
         |   abetting|          1|   comedyoferrors|        1|
         +-----------+-----------+----------------+---------+
         only showing top 10 rows
```

Unfortunately, the output formatting for the `%%SQL` "cell magic" is not configurable. The `%%DataFrame` magic handles variable width layout and also provides more display options. First, to see its options:

```
In [50]: %%dataframe
```

```
Out[50]: %%dataframe [arguments]
         DATAFRAME_CODE

         DATAFRAME_CODE can be any numbered lines of code, as long as the
         last line is a reference to a variable which is a DataFrame.
             Option    Description
         ------    -----------
         --limit   The number of records to return
                     (default: 10)
```

```
    --output  The type of the output: html, csv,
                json (default: html)
```

Now here's the previous query again, with the a `WHERE` clause added for good measure:

```
In [51]: val topLocations = sqlContext.sql("""
             SELECT word,  total_count, locations[0] AS top_location, counts[0] AS top_count
             FROM inverted_index
             WHERE word LIKE '%love%' OR word LIKE '%hate%'
         """)
```

Now use the `%%dataframe` *magic*.

```
In [52]: %%dataframe --limit 100
         topLocations
```

A *natural language processing* (NLP) expert might tell you that *love*, *loved*, *loves*, etc. are really the same word, because they are different conjugations of the verb *to love* and *love* is a noun, too. Similarly, should *gloves* (plural) and *glove* (singular) be handled differently?

What we really should do is extract the *stems* of these words and use those instead. NLP toolkits handle this *stemming* for you.

There's also a useful `show` method on `DataFrames`.

```
In [53]: topLocations.show

+-------+-----------+--------------------+---------+
|   word|total_count|        top_location|top_count|
+-------+-----------+--------------------+---------+
|beloved|         11|    tamingoftheshrew|        4|
| cloven|          1|    loveslabourslost|        1|
| cloves|          1|    loveslabourslost|        1|
|  glove|          3|    loveslabourslost|        2|
| glover|          1| merrywivesofwindsor|        1|
| gloves|          5| merrywivesofwindsor|        3|
|   hate|         22|midsummersnightsd...|        9|
|  hated|          6|midsummersnightsd...|        4|
|hateful|          5|midsummersnightsd...|        3|
|  hates|          5|         asyoulikeit|        2|
| hateth|          1|midsummersnightsd...|        1|
|   love|        662|    loveslabourslost|      121|
|  loved|         38|         asyoulikeit|       13|
| lovely|         15|midsummersnightsd...|        7|
|  lover|         33|         asyoulikeit|       14|
| lovers|         31|midsummersnightsd...|       17|
|  loves|         51| muchadoaboutnothing|       10|
| lovest|          8|    tamingoftheshrew|        3|
| loveth|          2|    loveslabourslost|        1|
|unloved|          1|midsummersnightsd...|        1|
+-------+-----------+--------------------+---------+
only showing top 20 rows
```

By default, it truncates column widths and only prints 20 rows. You can override both:

```
In [54]: topLocations.show(numRows = 40, truncate = false)
```

```
+--------+-----------+---------------------+---------+
|word    |total_count|top_location         |top_count|
+--------+-----------+---------------------+---------+
|beloved |11         |tamingoftheshrew     |4        |
|cloven  |1          |loveslabourslost     |1        |
|cloves  |1          |loveslabourslost     |1        |
|glove   |3          |loveslabourslost     |2        |
|glover  |1          |merrywivesofwindsor  |1        |
|gloves  |5          |merrywivesofwindsor  |3        |
|hate    |22         |midsummersnightsdream|9        |
|hated   |6          |midsummersnightsdream|4        |
|hateful |5          |midsummersnightsdream|3        |
|hates   |5          |asyoulikeit          |2        |
|hateth  |1          |midsummersnightsdream|1        |
|love    |662        |loveslabourslost     |121      |
|loved   |38         |asyoulikeit          |13       |
|lovely  |15         |midsummersnightsdream|7        |
|lover   |33         |asyoulikeit          |14       |
|lovers  |31         |midsummersnightsdream|17       |
|loves   |51         |muchadoaboutnothing  |10       |
|lovest  |8          |tamingoftheshrew     |3        |
|loveth  |2          |loveslabourslost     |1        |
|unloved |1          |midsummersnightsdream|1        |
|whate   |4          |tamingoftheshrew     |3        |
|whatever|1          |tamingoftheshrew     |1        |
+--------+-----------+---------------------+---------+
```

Note: Named Parameters

I used *named parameters* here, `show(numRows = 40, truncate = false)`, for legibility. They are optional in Scala, as long as you pass the values in the same order as the parameters are declared. You can also use named parameters to write the arguments in any order you want, not just declaration order. So, I could have just written `(40, false)`, but then you would rightly wonder what `false` means in this context.

Exercises:
See the Appendix for the solutions to the first two exercises.

- The glove, gloves, whate and whatever aren't really the love and hate we wanted ;) How might you change the query so be more specific.
- Modify the query to return the top two locations and counts.
- Before moving on, try writing other queries. Edit the query in the following cell:

```
In [55]: val sql1 = sqlContext.sql("""
             SELECT * FROM inverted_index
         """)
         sql1.show(10, false)
```

```
+-----------+-----------+-------------------------------------------------------------------------
|word       |total_count|locations
+-----------+-----------+-------------------------------------------------------------------------
|a          |3350       |[loveslabourslost, merrywivesofwindsor, muchadoaboutnothing, asyoulikei
|abandon    |6          |[asyoulikeit, tamingoftheshrew, twelfthnight]
|abate      |3          |[loveslabourslost, midsummersnightsdream, tamingoftheshrew]
|abatement  |1          |[twelfthnight]
|abbess     |8          |[comedyoferrors]
|abbey      |9          |[comedyoferrors]
|abbominable|1          |[loveslabourslost]
|abbreviated|1          |[loveslabourslost]
|abed       |2          |[asyoulikeit, twelfthnight]
|abetting   |1          |[comedyoferrors]
+-----------+-----------+-------------------------------------------------------------------------
only showing top 10 rows
```

Removing the "Stop Words" Did you notice that one record we saw above was for the word "a". Not very useful if you're using this data for text searching, *sentiment mining*, etc. So called *stop words*, like *a*, *an*, *the*, *he*, *she*, *it*, etc., could also be removed.

Recall the filter logic I added to remove "", word => word.size > 0. I could replace it with word => keep(word), where keep is a method that does any additional filtering I want, like removing stop words.

Exercise:

- Implement the keep(word: String):Boolean method and change the filter function to use it. Have keep return false for a small, hard-coded list of stop words (make up your own list or search for one). (See the Appendix for the solution.)

2.4 More on Pattern Matching Syntax

We've only scratched the surface of pattern matching. Let's explore it some more.

Here's another anonymous function using pattern matching that extends the previous function we passed to flatMap:

```
{
    case (location, "") =>
```

```
        Array.empty[((String, String), Int)]  // Return an empty array
    case (location, contents) =>
        val words = contents.split("""\W+""")
        val fileName = location.split(pathSep).last
        words.map(word => ((word, fileName), 1))
}.
```

You can have multiple case clauses, some of which might match on specific literal values ("" in this case) and others which are more general. The first case clause handles files with no content. The second clause is the same as before.

Pattern matching is *eager*. The first successful match in the order as written will win. If you reversed the order here, the `case (location, "")` would never match and the compiler would throw an "unreachable code" warning for it.

Note that you don't have to put the lines after the => inside braces, `{...}` (although you can). The => and `case` keywords (or the final `}`) are sufficient to mark these blocks. Also, for a single-expression block, like the one for the first case clause, you can put the expression on the same line after the => if you want (and it fits).

Finally, if none of the case clauses matches, then a MatchError exception is thrown. In our case, we *always* know we'll have two-element tuples, so the examples so far are fine.

Here's a final contrived example to illustrate what's possible, using a sequence of objects of different types:

```
In [56]: val stuff = Seq(1, 3.14159, 2L, 4.4F, ("one", 1), (404F, "boo"), ((11, 12), 21, 31), "h

         stuff.foreach {
             case i: Int                 => println(s"Found an Int:   $i")
             case l: Long                => println(s"Found a Long:   $l")
             case f: Float               => println(s"Found a Float:  $f")
             case d: Double              => println(s"Found a Double: $d")
             case (x1, x2) =>
                 println(s"Found a two-element tuple with elements of arbitrary type: ($x1, $x2)
             case ((x1a, x1b), _, x3) =>
                 println(s"Found a three-element tuple with 1st and 3th elements: ($x1a, $x1b) a
             case default                => println(s"Found something else: $default")
         }
```

```
Found an Int:   1
Found a Double: 3.14159
Found a Long:   2
Found a Float:  4.4
Found a two-element tuple with elements of arbitrary type: (one, 1)
Found a two-element tuple with elements of arbitrary type: (404.0, boo)
Found a three-element tuple with 1st and 3th elements: (11, 12) and 31
Found something else: hello
```

A few notes. * A literal like `1` is inferred to be `Int`, while `3.14159` is inferred to be `Double`. Add `L` or `F`, respectively, to infer `Long` or `Float` instead. * Note how we mixed specific type checking, e.g., `i: Int`, with more loosely-typed expressions, e.g., `(x1, x2)`, which expects a two-element tuple, but the element types are unconstrained. * All the words `i`, `l`, `f`, `d`, `x1`, `x2`, `x3`, and `default` are arbitrary variable names. Yes `default` is not a keyword, but an arbitrary choice for a variable name. We could use anything we want. * The last `default` clause specifies a variable with no type information. Hence, it matches *anything*, which is why this clause must appear last. This is the idiom to use when you aren't sure about the types of things you're matching against and you want to avoid a possible MatchError. * If you want to match that something *exists*, but you don't need to bind it to a variable, then use `_`, as in the three-element tuple example. * The three-element tuple example also demonstrates that arbitrary nesting of expressions is supported, where the first element is expected to be a two-element tuple.

All the anonymous functions we've seen that use these pattern matching clauses have this format:

`scala {       case firstCase => ...      case secondCase => ...      ...  }`

This format has a special name. It's called a *partial function*. All that means is that we only "promise" to accept arguments that match at least one of our `case` clauses, not any possible input.

The other kind of anonymous function we've seen is a *total function*, to be precise.

Recall we said that for total functions you can use either `(...)` or `{...}` around them, depending on the "look" you want. For *partial functions*, you *must* use `{...}`.

Also, recall that we used pattern matching with assignment:

```
val (locations, counts) = vect.unzip
```

Vector.unzip returns a two-element tuple, where each element is a collection. We matched on that tuple and assigned each piece to a variable. Here's another contrived example, with nested tuple elements:

```
In [57]: val (a, (b, (c1, c2), d)) = ("A", ("B", ("C1", "C2"), "D"))
         println(s" $a, $b, $c1, $c2, $d")

 A, B, C1, C2, D
```

Try adding an `"E"` element to the tuple on the right-hand side, without changing the left-hand side. What happens? Try removing the `"D"` and `"E"` elements. What happens now?

We'll come back to one last example of pattern matching when we discuss *case classes*.

2.5 Scala's Object Model

Scala is a *hybrid*, object-oriented and functional programming language. The philosophy of Scala is that you exploit object orientation for encapsulation of details, i.e., *modularity*, but use functional programming for its logical precision when implementing those details. Most of what we've seen so far falls into the functional programming camp. Much of data manipulation and analysis is really Mathematics. Functional programming tries to stay close to how functions and values work in Mathematics.

However, when writing non-trivial Spark programs, it's occasionally useful to exploit the object-oriented features.

2.5.1 Classes vs. Instances

Scala uses the same distinction between classes and instances that you find in Java. Classes are like *templates* used to create instances.

We've talked about the *types* of things, like `word` is a `String` and `totalCount` is an `Int`. A class defines a *type* in the same sense.

Here is an example class that we might use to represent the inverted index records we just created:

```
In [58]: class IIRecord1(
             word: String,
             total_count: Int,
             locations: Array[String],
             counts: Array[Int]) {

             /** CSV formatted string, but use [a,b,c] for the arrays */
             override def toString: String = {
                 val locStr = locations.mkString("[", ",", "]")  // i.e., "[a,b,c]"
                 val cntStr = counts.mkString("[", ",", "]")  // i.e., "[1,2,3]"
                 s"$word,$total_count,$locStr,$cntStr"
             }
         }

         new IIRecord1("hello", 3, Array("one", "two"), Array(1, 2))

Out[58]: IIRecord1 = hello,3,[one,two],[1,2]
```

When defining a class, the argument list after the class name is the argument list for the *primary constructor*. You can define secondary constructors, too, but it's not very common, in part for reasons we'll see shortly.

Note that when you override a method that's defined in a parent class, like Java's `Object.toString`, Scala requires you to add the `override` keyword.

We created an *instance* of `IIRecord1` using `new`, just like in Java.

Finally, as a side note, we've been using `Ints` (integers) all along for the various counts, but really for "big data", we should probably use `Longs`.

2.5.2 Objects

I've been careful to use the word *instance* for things we create from classes. That's because Scala has built-in support for the Singleton Design Pattern, i.e., when we only want one instance of a class. We use the `object` keyword.

For example, in Java, you define a class with a `static void main(String[] arguments)` method as your entry point into your program. In Scala, you use an `object` to hold `main`, as follows:

```
In [59]: object MySparkJob {

             val greeting = "Hello Spark!"
```

```
        def main(arguments: Array[String]) = {
            println(greeting)

            // Create your SparkContext, etc., etc.
        }
    }
```

Just as for classes, the name of the object can be anything you want. There is no `static` keyword in Scala. Instead of adding `static` methods and fields to classes as in Java, you put them in objects instead, as here.

> **NOTE:** Because the Scala compiler must generate valid JVM byte code, these definitions are converted into the equivalent, Java-like static definitions in the output byte code.

2.5.3 Case Classes

Tuples are handy for representing records and for decomposing them with pattern matching. However, it would be nice if the fields were *named*, as well as *typed*. A good use for a class, like our `IIRecord1` above, us to represent this structure and give us named fields. Let's now refine that class definition to exploit some extra, very useful features in Scala.

Consider the following definition of a *case class* that represents our final record type.

```
In [60]: case class IIRecord(
             word: String,
             total_count: Int = 0,
             locations: Array[String] = Array.empty,
             counts: Array[Int] = Array.empty) {

             /**
              * Different than our CSV output above, but see toCSV.
              * Array.toString is useless, so format these ourselves.
              */
             override def toString: String =
                 s"""IIRecord($word, $total_count, $locStr, $cntStr)"""

             /** CSV-formatted string, but use [a,b,c] for the arrays */
             def toCSV: String =
                 s"$word,$total_count,$locStr,$cntStr"

             /** Return a JSON-formatted string for the instance. */
             def toJSONString: String =
                 s"""{
                 |  "word":        "$word",
                 |  "total_count": $total_count,
                 |  "locations":   ${toJSONArrayString(locations)},
                 |  "counts"       ${toArrayString(counts, ", ")}
                 |}
                 |""".stripMargin
```

```
        private def locStr = toArrayString(locations)
        private def cntStr = toArrayString(counts)

        // "[_]" means we don't care what type of elements; we're just
        // calling toString on them!
        private def toArrayString(array: Array[_], delim: String = ","): String =
            array.mkString("[", delim, "]")  // i.e., "[a,b,c]"

        private def toJSONArrayString(array: Array[String]): String =
            toArrayString(array.map(quote), ", ")

        private def quote(word: String): String = "\"" + word + "\""
    }
```

I said that defining secondary constructors is not very common. In part, it's because I used a convenient feature, the ability to define default values for arguments to methods, including the primary constructor. The default values mean that I can create instances without providing all the arguments explicitly, as long as there is a default value defined, and similarly for calling methods. Consider these two examples:

```
In [61]: val hello = new IIRecord("hello")
         val world = new IIRecord("world!", 3, Array("one", "two"), Array(1, 2))

         println("\n`toString` output:")
         println(hello)
         println(world)

         println("\n`toJSONString` output:")
         println(hello.toJSONString)
         println(world.toJSONString)

         println("\n`toCSV` output:")
         println(hello.toCSV)
         println(world.toCSV)
```

```

`toString` output:
IIRecord(hello, 0, [], [])
IIRecord(world!, 3, [one,two], [1,2])

`toJSONString` output:
{
  "word":        "hello",
  "total_count": 0,
  "locations":   [],
  "counts"       []
}
```

```
{
  "word":        "world!",
  "total_count": 3,
  "locations":   ["one", "two"],
  "counts"       [1, 2]
}


`toCSV` output:
hello,0,[],[]
world!,3,[one,two],[1,2]
```

I added `toJSONString` to illustrate adding *public* methods, the default visibility, and *private* methods to a class definition. By the way, when there are no methods or non-field variables to define, I can omit the body complete; no empty `{}` required.

Recall that the `override` keyword is required when redefining `toString`.

Okay, what about that `case` keyword? It tells the compiler to do several useful things for us, eliminating a lot of boilerplate that we would have to write for ourselves with other languages, especially Java:

1. Treat each constructor argument as an immutable (`val`) private field of the instance.
2. Generate a public reader method for the field with the same name (e.g., `word`).
3. Generate *correct* implementations of the `equals` and `hashCode` methods, which people often implement incorrectly, as well as a default `toString` method. You can use your own definitions by adding them explicitly to the body. We did this for `toString`, to format the arrays in a nicer way than the default `Array[_].toString` method.
4. Generate an `object IIRecord`, i.e., with the same name. The object is called the *companion object*.
5. Generate a "factory" method in the companion object that takes the same argument list and instantiates an instance.
6. Generate helper methods in the companion object that support pattern matching.

Points 1 and 2 make each argument behave as if they are public, read-only fields of the instance, but they are actually implemented as described.

Point 3 is important for correct behavior. Case class instances are often used as keys in Maps and Sets, Spark RDD and DataFrame methods, etc. In fact, you should *only* use your case classes or Scala's built-in types with well-defined `hashCode` and `equals` methods (like `Int` and other number types, `String`, tuples, etc.) as keys.

For point 4, the *companion object* is generated automatically by the compiler. It adds the "factory" method discussed in point 5, and methods that support pattern matching, point 6. You can explicitly define these methods and others yourself, as well as fields to hold state. The compiler will still insert these other methods. However, see Ambiguities with Companion Objects. The bottom line is that you shouldn't define case classes in notebooks like this with extra methods in the companion object, due to parsing ambiguities.

Point 5 means you actually rarely use `new` when creating instances. That is, the following are effectively equivalent:

```
In [62]: val hello1 = new IIRecord("hello1")
         val hello2 = IIRecord("hello2")
```

What actually happens in the second case, without `new`? The "factory" method is actually called `apply`. In Scala, whenever you put an argument list after any *instance*, including these `objects`, as in the `hello2` case, Scala looks for an `apply` method to call. The arguments have to match the argument list for apply (number of arguments, types of arguments, accounting for default argument values, etc.). Hence, the `hello2` declaration is really this:

```
In [63]: val hello2b = IIRecord.apply("hello2b")
```

You can exploit this feature, too, in your other classes. We talked about word stemming above. Suppose you write a stemming library and declare an object for as the entry point. Here, I'll just do something simple; assume a trailing "s" means the word is a plural and remove it (a bad assumption...):

```
In [64]: object stem {
             def apply(word: String): String = word.replaceFirst("s$", "") // insert real implem
         }

         println(stem("dog"))
         println(stem("dogs"))

dog
dog
```

Note how it looks like I'm calling a function or method named `stem`. Scala allows object and class names to start with a lower case letter.

Finally, point 6 means we can use our custom case classes in pattern matching expressions. I won't go into the methods actually implemented in the companion object and how they support pattern matching. I'll just use the "magic" in the following example that "parses" or previously-defined `hello` and `world` instances.

```
In [65]: Seq(hello, world).map {
             case IIRecord(word, 0, _, _) => s"$word with no occurrences."
             case IIRecord(word, cnt, locs, cnts) =>
                 s"$word occurs $cnt times: ${locs.zip(cnts).mkString(", ")}"
         }

Out[65]: Seq[String] = List(hello with no occurrences., world! occurs 3 times: (one,1), (two,2))
```

The first case clause ignores the locations and counts, because I know they will be empty arrays if the total count is 0!

The second case clause uses the `zip` method to put the locations and counts back together. Recall we used `unzip` to create the separate collections.

2.6 Datasets and DataFrames

So far, we've mostly used Spark's RDD API. It's common to use case classes to represent the "schema" of records when working with RDDs, but also with a new type, Dataset[T], analogous to RDD[T], where the T represents the type of records.

A problem with DataFrames is the fact that the fields are untyped until you try to access them. Datasets restore the type safety of RDDs by using a case class as the definition of the schema.

Datasets were introduced in Spark 1.6.0, but they are somewhat incomplete in the 1.6.X releases. In Spark 2.0.0, Dataset becomes the "parent" class of DataFrame. This means that you'll be encouraged to use the greater type safety of Dataset, but you can still use DataFrame if you want. Now, DataFrame will be the equivalent of Dataset[Row], where Row is the loosely-typed representation of the row and its columns.

Let's try it out. But first, we need to import some SparkSQL-related code. Scala lets you import code almost anywhere, whereas Java requires imports at the beginning of source files. Scala also lets you import members of instances, not just the static imports supported by Java.

So, the next cell imports some "implicits" from the SQLContext instance already in scope. Unfortunately, due to a scoping ambiguity involving notebooks and the Scala interpreter, we need to assign sqlContext to a new variable, *then* import from that:

```
In [66]: val sqlc = sqlContext
         import sqlc.implicits._
```

We'll explain what "implicits" are later. For now, suffice it to say that they are used to "allow" us to call the as method on our iiDF DataFrame, which converts it to a Dataset[IIRecord].

```
In [67]: val iiDS = iiDF.as[IIRecord]
         iiDS

Out[67]: org.apache.spark.sql.Dataset[IIRecord] = [word: string, total_count: int ... 2 more fie

In [68]: iiDS.show

+-----------+-----------+--------------------+--------------------+
|       word|total_count|           locations|              counts|
+-----------+-----------+--------------------+--------------------+
|          a|       3350|[loveslabourslost...|[507, 494, 492, 4...|
|    abandon|          6|[asyoulikeit, tam...|        [4, 1, 1]|
|      abate|          3|[loveslabourslost...|        [1, 1, 1]|
|  abatement|          1|       [twelfthnight]|                 [1]|
|     abbess|          8|     [comedyoferrors]|                 [8]|
|      abbey|          9|     [comedyoferrors]|                 [9]|
|abbominable|          1|   [loveslabourslost]|                 [1]|
|abbreviated|          1|   [loveslabourslost]|                 [1]|
|       abed|          2|[asyoulikeit, twe...|              [1, 1]|
|   abetting|          1|     [comedyoferrors]|                 [1]|
|abhominable|          1|   [loveslabourslost]|                 [1]|
|      abhor|          5|[asyoulikeit, com...|     [1, 1, 1, 1, 1]|
|     abhors|          2|       [twelfthnight]|                 [2]|
|      abide|          5|[merrywivesofwind...|              [3, 2]|
|     abides|          1|[muchadoaboutnoth...|                 [1]|
|    ability|          2|[muchadoaboutnoth...|              [1, 1]|
|     abject|          2|[comedyoferrors, ...|              [1, 1]|
|     abjure|          1|[midsummersnights...|                 [1]|
|    abjured|          2|[tamingoftheshrew...|              [1, 1]|
|       able|          9|[merrywivesofwind...|     [4, 2, 1, 1, 1]|
+-----------+-----------+--------------------+--------------------+
only showing top 20 rows
```

3 "Scala for Spark 102"

We've covered a lot already in this notebook, focusing on the most important topics you need to know about Scala for daily use. Let's call them the "Scala for Spark 101" material.

At this point, I suggest you create a new notebook and play with Spark using what you've learned so far, then come back to this point if you run into something we didn't cover already. Chances are you're ready to learn the next bits of useful Scala, the "102" material.

3.0.1 Importing Everything in a Package

In Java, `import foo.bar.*;` means import everything in the `bar` package.

In Scala, `*` is actually a legal method name; think of defining multiplication for custom numeric types, like `Matrix`. Hence, this import statement in Scala would be ambigious. Therefore, Scala uses `_` instead of `*`, `import foo.bar._` (with the semicolon inferred).

Incidentally, what would that `*` method definition look like? Something like this:

```
case class Matrix(rows: Array[Array[Double]]) {  // Each row is an Array[Double]

    /** Multiple this matrix by another. */
    def *(other: Matrix): Matrix = ...

    /** Add this matrix by another. */
    def +(other: Matrix): Matrix = ...

    ...
}

val row1: Array[Array[Double]] = ...
val row2: Array[Array[Double]] = ...
val m1 = Matrix(rows1)
val m2 = Matrix(rows2)
val m1_times_m2 = m1 * m2
val m1_plus_m2 = m1 + m2
```

3.0.2 Operator Syntax

Wait!! What's this `m1 * m2` stuff?? Shouldn't it be `m1.*(m2)`. It would be really convenient to use "operator syntax", more precisely called *infix operator notation* for many methods like `*` and `+` here. The Scala parser supports this with a simple relaxation of the rules; when a method takes a single argument, you can omit the period `.` and parentheses `(...)`. Hence the following really is equivalent:

```
val m1_times_m2 = m1.*(m2)
val m1_times_m2 = m1 * m2
```

This convenience can lead to confusing code, especially for beginners to Scala, so use it cautiously.

3.0.3 Traits

Traits are similar to Java 8 *interfaces*, used to define abstractions, but with the ability to provide "default" implementations of the methods declared. Unlike Java 8 interfaces, traits can also have fields representing "state" information about instances. There is a blury line between traits and *abstract classes*, again where some member methods or fields are not defined. In both cases, a subtype of a trait and/or an abstract class must define any undefined members if you want to construct instances of it.

So, why have both traits and abstract classes? It's because Java only allows *single inheritance*; there can be only one *parent* type, which is normally where you would use an abstract class, but Scala lets you "mix in" one or more additional traits (or use a trait as the parent class - yes, confusing). A great example "mix in" trait is one that implements logging. Any "service" type can mix in the logging trait to get "instant" access to this reusable functionality. Schematically, it looks like the following:

```
// Assume severity `Level` and `Logger` types defined elsewhere...
trait Logging {

    def log(level: Level, message: String): Unit = logger.log(level, message)

    private logger: Logger = ...
}

abstract class Service {
    def run(): Unit   // No body, so abstract!
}

class MyService extends Service with Logging {
    def run(): Unit = {
        log(INFO, "Staring MyService...")
        ...
        log(INFO, "Finished MyService")
    }
}
```

`Unit` is Scala's equivalent to Java's `void`. It actually is a true type with a single return value, unlike `void`, but we use it in the same sense of "nothing useful will be returned".

3.0.4 Ranges

What if you want some numbers between a start and end value? Use a Range, which has a nice literal syntax, e.g., `1 until 100`, `2 to 200 by 3`.

The `Range` always includes the lower bound. Using `to` in a `Range` makes it *inclusive* at the upper bound. Using `until` makes it *exclusive* at the upper bound. Use `by` to specify a delta, which defaults to `1`.

```
In [69]: 1 until 10

Out[69]: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

In [70]: 1 to 10

Out[70]: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

In [71]: 1 to 10 by 3

Out[71]: scala.collection.immutable.Range = Range(1, 4, 7, 10)
```

When you need a small test data set to play with Spark, ranges can be convenient.

```
In [72]: val rdd7 = sc.parallelize(1 to 50).
             map(i => (i, i%7)).
             groupBy{ case (i, seven) => seven }.
             sortByKey()
         rdd7.take(7).foreach(println)

(0,CompactBuffer((7,0), (14,0), (21,0), (28,0), (35,0), (42,0), (49,0)))
(1,CompactBuffer((1,1), (8,1), (15,1), (22,1), (29,1), (36,1), (43,1), (50,1)))
(2,CompactBuffer((2,2), (9,2), (16,2), (23,2), (30,2), (37,2), (44,2)))
(3,CompactBuffer((3,3), (10,3), (17,3), (24,3), (31,3), (38,3), (45,3)))
(4,CompactBuffer((4,4), (11,4), (18,4), (25,4), (32,4), (39,4), (46,4)))
(5,CompactBuffer((5,5), (12,5), (19,5), (26,5), (33,5), (40,5), (47,5)))
(6,CompactBuffer((6,6), (13,6), (20,6), (27,6), (34,6), (41,6), (48,6)))
```

SparkContext also has a `range` method that effectively does the same thing as `sc.parallelize(some_range)`.

3.0.5 Scala Interpreter (REPL) vs. Notebooks vs. Scala Compiler

This notebook has been using a running Scala interpreter, a.k.a. *REPL* ("read, eval, print, loop") to parse the Scala code. The Spark distribution comes with a `spark-shell` script that also lets you use the interpreter from the command line, but without the nice notebook UI.

If you use `spark-shell`, there are a few other behavior changes you should know about.