Trabajar con Scala desde Jupyter Notebook.

El formato notebook de Jupyter es ideal para el aprendizaje de lenguajes de programación ya que nos permite ejecutar el código y obtener las salidas por partes y hace sencillo tanto el seguimiento del proceso como la detección de fallos. En este documento veremos cómo configurar nuestro Jupyter Notebook para programar en Scala.

Prerequisitos:

- Instalación de Apache Spark (en nuestro caso v3.0.1).
- Jupyter Notebook (recomiendo instalar Anaconda que nos permite hacer una gestión sencilla de nuestros paquetes y viene con Jupyter Notebook incluido).
- Python 3.5+ (si hacemos la descarga de las versiones más recientes de Anaconda como 4.2.0+ viene con versiones de Python válidas preinstaladas).

En esta guía gestionaremos nuestros paquetes desde Anaconda.

Paso 1: Instalación del kernel "spylon-kernel".

Desde la consola de Anaconda (Anaconda prompt) ejecutaremos el comando:

```
pip install spylon-kernel
```

* En ocasiones podemos encontrar un error al ejecutar comandos en la consola de Anaconda, para prevenirlos una posible solución es ejecutar como administrador.

Paso 2: Habilitar el acceso al kernel desde Jupyter Notebook.

En la misma consola ejecutamos:

```
python -m spylon_kernel install
```

Paso 3: Abrimos Jupyter Notebook.

Podemos hacerlo desde el gestor de Anaconda o insertando el comando `jupyter notebook` en la consola de Anaconda.

Paso 4: Iniciar el kernel y abrir un nuevo notebook.

Una vez en el gestor de archivos de Jupyter Notebook hacemos click en el recuadro "New" y seleccionamos "spylon-kernel". Esto debería abrirnos un nuevo notebook que nos permite ejecutar comandos en Scala.

¡Importante!

Para que todo funcione tenemos que asegurarnos de que tenemos configurada la variable SPARK-HOME en nuestro sistema y si estamos en Windows asegurarnos de tener instalados los winutils para nuestra versión de Hadoop con su respectiva variable de entorno configurada. Vamos a cubrir estos dos asuntos en Windows 10:

Configuración de la variable de entorno:

-Localizar la carpeta donde están los archivos de Spark, en nuestro caso Spark 3.0.1.

-Abrir el explorador de archivos, hacer click secundario en Equipo.

-A continuación, seleccionamos:

"Configuración avanzada del sistema" -> "Variables de entorno" -> En variables del sistema creamos una nueva con nombre SPARK_HOME y en "Valor de la variable" introducimos el directorio que nos dirija al nivel superior de la carpeta de los archivos de Spark.

Una vez aceptado deberíamos ver una nueva variable de sistema que en nuestro caso luce de la siguiente manera:

SPARK_HOME	C:\spark-3.0.1-bin-hadoop2.7

Es importante asegurarnos que no hay directorios con espacios en el nombre. La manera más sencilla de garantizarlo es mover la carpeta de Spark a "C:\".

Instalación de Winutils:

Podemos descargar las diferentes versiones de winutils en el siguiente enlace:

https://github.com/steveloughran/winutils

Una vez localizada la carpeta con el nombre de nuestra versión de Hadoop, en nuestro caso 2.7.1, crearemos una carpeta "Hadoop" en el directorio "C:/". Dentro de "C:/Hadoop/" pegamos la carpeta con los archivos winutils, en nuestro caso "hadoop-2.7.1".

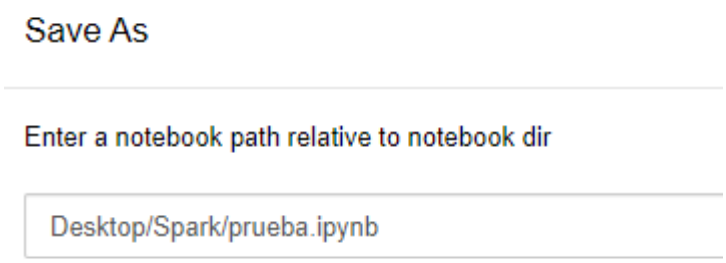
Cuando tengamos esto completado volvemos al panel de configuración de variables de entorno y creamos una variable de sistema que quede de la siguiente manera:

HADOOP_HOME	C:\Hadoop\hadoop-2.7.1\

Hecho esto tenemos los archivos winutils instalados y configurados para funcionar.

Nota sobre guardar el Notebook:

Al seleccionar la opción del notebook "Guardar como" nos pide introducir el directorio de guardado y tras él el nombre con el que queramos guardar el archivo. Es importante guardarlo con la extensión .ipynb ya que por defecto lo guarda sin extensión.



Nota sobre ejecutar comandos en Python desde el kernel "spylon-kernel":

Es interesante saber que podemos ejecutar comandos en lenguaje Python en este mismo notebook. Lo ilustramos con el siguiente ejemplo:

Podemos ejecutar el código en Scala:

```
val a = 5
val b = 7
a+b
```

Una instrucción similar en Python sería de la siguiente manera:

```
c = 3
d = 4
c+d
```

Como era de esperar, esta instrucción nos dará error ya que Scala no la interpreta. La solución es añadir `%%python` al inicio del código:

```
%%python
c = 3
d = 4
c+d
```

Y debería darnos el resultado correcto sin errores.

Notar que una variable definida con Python no funcionará cuando intentemos interpretarla con Scala ya que siguen procesos de definición diferentes por lo que los resultados de los chunks en Python por lo general sólo podremos utilizarlos en otros chunks en Python.

Nota sobre ejecutar comandos en Scala desde el kernel "Python 3":

A su vez, podemos evaluar expresiones escritas en Scala en el kernel de Python importando el intérprete de Scala. Lo ilustramos con un ejemplo práctico:

Imaginemos que queremos interpretar la expresión de Scala:

```
val x = 8
x
```

En el kernel Python 3 lo haremos de la siguiente manera:

Importamos el intérprete tratando spylon-kernel como una librería.

```
from spylon_kernel import get_scala_interpreter
```

Le asignamos un nombre al intérprete para llamarlo fácilmente.

```
interp = get_scala_interpreter()
```

Ahora la utilizamos sobre el bloque de código en Scala que deseemos interpretar.

```
interp.interpret("""
val x = 8
x
""")
```

Observamos que la función *"interpret"* es la que se encarga de hacer el trabajo.

Y estamos en condiciones de hacer el print del último resultado arrojado por el intérprete con la función *"last_result"*.

```
interp.last_result()
```

```
8
```

Ya que "x" ha sido almacenado como una variable definida por Scala no podemos hacer algo como lo siguiente en el kernel "Python 3".

```
print(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-fc17d851ef81> in <module>
----> 1 print(x)

NameError: name 'x' is not defined
```

Pero podemos resolver esto guardando como una variable en Python el resultado de la interpretación del código Scala de la siguiente manera:

```
result = interp.last_result()
```

Y ahora podemos utilizar el resultado como si fuera una variable en Python:

```
a = result + 7

a
```

```
15
```