Big Data Fundamentals with PySpark (Introduction to Big Data analysis with Spark)

Understanding SparkContext

A SparkContext represents the entry point to Spark functionality. It's like a key to your car. PySpark automatically creates a `SparkContext` for you in the PySpark shell (so you don't have to create it by yourself) and is exposed via a variable `sc` .

In this simple exercise, you'll find out the attributes of the `SparkContext` in your PySpark shell which you'll be using for the rest of the course.

```
# Print the version of SparkContext
print("The version of Spark Context in the PySpark shell is", sc.version)

# Print the Python version of SparkContext
print("The Python version of Spark Context in the PySpark shell is", sc.pythonVer)

# Print the master of SparkContext
print("The master of Spark Context in the PySpark shell is", sc.master)
```

Interactive Use of PySpark

Spark comes with an interactive python shell in which PySpark is already installed in it. PySpark shell is useful for basic testing and debugging and it is quite powerful. The easiest way to demonstrate the power of PySpark's shell is to start using it. In this example, you'll load a simple list containing numbers ranging from 1 to 100 in the PySpark shell.

The most important thing to understand here is that we are not creating any SparkContext object because PySpark automatically creates the SparkContext object named `sc` , by default in the PySpark shell.

```
# Create a python list of numbers from 1 to 100
numb = range(1, 100)

```

```
# Load the list into PySpark
spark_data = sc.parallelize(numb)
```

Loading data in PySpark Shell

In PySpark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. In the previous exercise, you have seen an example of loading a list as parallelized collections and in this exercise, you'll load the data from a local file in PySpark shell.

Remember you already have a SparkContext `sc` and `file_path` variable (which is the path to the `README.md` file) already available in your workspace.

```
# Load a local file into PySpark shell
lines = sc.textFile(file_path)
```

Use lambda with map()

The `map()` function in Python returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.). The general syntax of `map()` function is `map(fun, iter)` . We can also use lambda functions with `map()` . The general syntax of `map()` function with `lambda()` is `map(lambda <agument>:<expression>, iter)` . Refer to slide 5 of video 1.7 for general help of `map()` function with `lambda()` .

In this exercise, you'll be using `lambda` function inside the `map()` built-in function to square all numbers in the list.

```
# Print my_list in the console
print("Input list is", my_list)

# Square all numbers in my_list
squared_list_lambda = list(map(lambda x: x**2, my_list))

# Print the result of the map function
print("The squared numbers are", squared_list_lambda)
```

Use lambda with filter()

Another function that is used extensively in Python is the `filter()` function. The `filter()` function in Python takes in a function and a list as arguments. The general syntax of the `filter()` function is `filter(function, list_of_input)` . Similar to the `map()` , `filter()` can be used with `lambda()` function. The general syntax of the `filter()`

function with `lambda()` is `filter(lambda <argument>:<expression>, list)` . Refer to slide 6 of video 1.7 for general help of the `filter()` function with `lambda()` .

In this exercise, you'll be using `lambda()` function inside the `filter()` built-in function to find all the numbers divisible by 10 in the list.

```
# Print my_list2 in the console
print("Input list is:", my_list2)

# Filter numbers divisible by 10
filtered_list = list(filter(lambda x: (x%10 == 0), my_list2))

# Print the numbers divisible by 10
print("Numbers divisible by 10 are:", filtered_list)
```