Pyspark Functions, Anonymous and Otherwise

Simple operations in python

Try typing the following lines at the spark-shell:

```
>>> 2 + 2                    # Should be Int = 2
4
>>> 2.0 + 2                  # Should be Double = 2.0
4.0
>>> "This is a string"       # 'This is a string'
'This is a string'
```

Function Definitions In Pyspark

In this little exercise, we'll define a simple function in the most straightforward, obvious way and then show how Scala's ability to infer context and types allows us to be more succinct.

Let's start by defining a simple add function and testing it.

```
def add(x, y):
  return x + y

add(42,13)

# Shorter version all on one line
def add(x, y): return x + y
```

You should see 55 as your result.

We don't have a lot of other fancy definition forms for Python as we did for Scala.

Anonymous Functions or Function Literals

First, we'll define a named greeting function named `greeting` that just prepends a cheery "Hello ".

```
# a named function greeting
def greeting(x): return "Hello " + x

greeting("Joe")
'Hello Joe'
>>>
```

Now we'll do the same with an anonymous function, which we'll assign to the variable greeting. We get the same cheery greeting.

```
# an anonymous function whose definition is assigned to variable greeting
greeting = lambda x: "Hello " + x
greeting("Joe")
'Hello Joe'
>>>
```

Now let's create a list of names and apply our anonymous function stored in `greeting` to that list. Unfortunately, we can't use `map` to do this because `map` isn't defined for local lists. Instead, we'll use a Python comprehension.

```
names = ["Joe", "Mary", "Barbara"]
# comprehension for a local list;
# local lists don't have the map method defined.
[greeting(x) for x in names]
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>
```

Now do this in Pyspark

If we parallelize our list, we can then use the `map` function on our list.

```
names = sc.parallelize(["Joe", "Mary", "Barbara"])
names.map(greeting).collect()
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>

# Now let's get rid of the name greeting

names.map(lambda x: "Hello " + x).collect()
['Hello Joe', 'Hello Mary', 'Hello Barbara']
>>>
```

Here are a collection of other function definitions to try out to get a feel for how anonymous functions can shorten our code and make it more readable at the same time.

```
maximize = lambda a, b: a if (a > b) else b
maximize(5, 3)

# Define doubler as an immutable variable whose value is a function (note we aren't using def)
doubler = lambda x: x * 2                        # This assigns a function literal to doubler
doubler(4)

# We can also use this for our even number tester
nums = sc.parallelize([1, 2, 3, 4, 5])
nums.map(lambda x: x % 2 == 0).collect()


# These literal defintions are incredibly valuable in Spark
# because many of the Spark operations
# take functional arguments. Most of these will never be assigned
# to a variable. For example
# the following finds list entries that have an "f"
# in them using an anonymous function/function literal

mylist = sc.parallelize(["foo", "bar", "fight"])
mylist.filter(lambda x: "f" in x).collect()

# We could have written that as follows,
# but it's longer and less clear

mylist = sc.parallelize(["foo", "bar", "fight"])
def ffilter(s):
    return "f" in s   # define a named function to search for f

mylist.filter(ffilter).collect()      # will generate the same result as previous
```

A Full PySpark Example

Load error messages from a log into memory, then interactively search for various patterns.

The file `log.txt` has the following 5 lines:

```
ERROR       php: dying for unknown reasons
WARN        dave, are you angry at me?
ERROR       did mysql just barf?
WARN        xylons approaching
ERROR       mysql cluster: replace with spark cluster
```

Our objective is to count all the error messages (not warnings) that have reference *mysql* or *php*.

```
lines = sc.textFile("hdfs:///data/logs/log.txt")

# transformed RDDs
errors = lines.filter(lambda line: line.startswith("ERROR"))
fields = errors.map(lambda message: message.split("\t"))
messages = fields.map(lambda r: r[1])
messages.cache()

# actions
messages.filter(lambda m: "mysql" in m).count()
messages.filter(lambda m: "php" in m).count()
```

You should see 2 messages that have "mysql" in them, and one message that has "php".

Now Wordcount

The periods in this dataflow pipeline are at the ends of lines so that we can execute this code in interactive pyspark without modification.

Type this into pyspark and see how it works for you.

```
text_file = sc.textFile("hdfs:///data/shakespeare/input")
word_counts = text_file \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

# Let's sort the results by count

sorted_counts = word_counts.sortBy(lambda x: x[1], ascending=False)
sorted_counts.saveAsTextFile("hdfs:///tmp/shakespeare-wc-python")
```

You can examine the results using `hdfs dfs`. We've omitted the permissions information so that the lines are visible in our printed output.

```
hdfs dfs -ls /tmp/shakespeare-wc-python
Found 3 items
/tmp/shakespeare-wc-python/_SUCCESS
/tmp/shakespeare-wc-python/part-00000
/tmp/shakespeare-wc-python/part-00001
[vagrant@edge ~]$ hdfs dfs -cat /tmp/shakespeare-wc-python/part-00000 | head -5
hdfs dfs -cat /tmp/shakespeare-wc-python/part-00000 | head -5
(u'the', 25815)
(u'I', 20402)
(u'and', 19249)
(u'to', 17222)
(u'of', 16526)
```

Compute Pi

Estimate Pi in Pyspark

This program generates 100,000 x and y variables between 0 and 1. It then counts the ratio of those that fall within a unit circle over the number of total samples; that value should approximate pi/4.

Try various values of NUM_SAMPLES to see how the computed value and runtimes vary.

```
import sys
from random import random
from operator import add
NUM_SAMPLES = 10000
def f(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0

count = sc.parallelize(range(1, NUM_SAMPLES + 1)).map(f).reduce(add)
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))

Pi is roughly 3.136000
>>>
```

It's not a great estimate, but you can improve it by increasing the number of samples.

This step concludes the lab.