Map Reduce Example using Hadoop & Python

Overview

MapReduce is a data processing job which splits the input data into independent chunks (large pieces), which are then processed by the map function and then reduced by grouping similar sets of the data.

Using Hadoop, the MapReduce framework can allow code to be executed on multiple servers  called nodes without having to worry about single machine performance. Nodes can be grouped into clusters, dispersing processing and memory constraints, for faster access to datasets.

Here is an overview of how the MapReduce works.

# Sample Superhero Dataset

# Looks like we have a few duplicates: Iron Man and Thor.

# While Hawkeye, Spiderman, and The Hulk, all appear once.

>>

Iron Man

Thor

Hawkeye

Iron Man

The Hulk

Spiderman

Thor

The first part of the processing is our Map job, which takes subsets of your data and isolates the entries in it. For each entry we assign a key and a value, and create a tuple (k\_i, v\_i). For each key (k\_i), our example contains Superhero names , we assign the(v\_i) as 1, because in our example we want to count the number of times the name appears.

# Part 1 - Map and subset

# Subset 1

Iron Man → Key: 'Iron Man’, Value: ‘1’

Thor → Key: ‘Thor’, Value: ‘1’

Hawkeye → Key: ‘Hawkeye’, Value: ‘1’

Iron Man → Key: ‘Iron Man’, Value: ‘1’

# Subset 2

The Hulk → Key: ‘The Hulk’, Value: ‘1’

Spiderman → Key: ‘Spiderman’, Value: ‘1’

Thor → Key: ‘Thor’, Value: ‘1’

The second is the Reduce job, that converts the tuple data from the Map job, and reduces the tuples to only contain unique keys while adding their values together.

# Part 2 - Reduce

# Our tuples

((Iron Man, 1), (Thor, 1), (Hawkeye, 1), (Iron Man, 1), (The Hulk, 1), (Spiderman, 1), (Thor, 1))

# Our reduced tuples

((Iron Man, 2), (Thor, 2), (Hawkeye, 1), (The Hulk, 1), (Spiderman, 1))

What we are left with is a reduced subset of our original data, with the number of times each unique key appears. So we’ve taken the list of 7 Superheroes and reduced it down to 5 tuples.

Why Not Entirely Java?

Working with Hadoop using Python and not Java can be done with a bundle of the different libraries available. The main reason is not having to compile the code. Which makes development and debugging much faster. We are going to be using Java to execute part of the code, but the **mapper.py** and **reducer.py** will all be written in Python.

Here is the typical steps of how to use Hadoop, and exactly what we’re going to be doing below

1. Load your data in a cluster
2. Use MapReduce to analyze the data
3. Store the results in the cluster
4. Read the results from the cluster

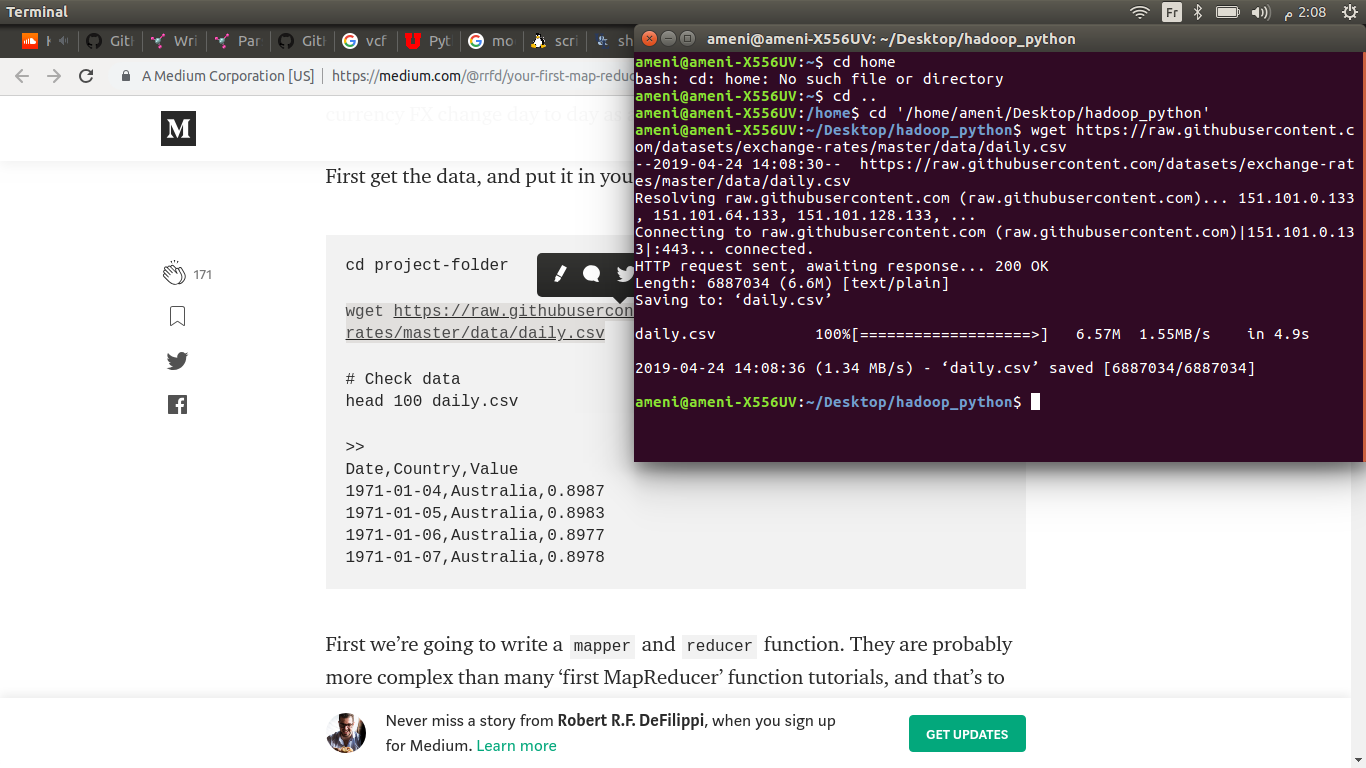
**Step 1**

Let’s take this example of [dataset](https://github.com/datasets/exchange-rates/blob/master/data/daily.csv) which shows the daily exchange rate for most countries in the world against the US dollar. We will be implementing a MapReduce application to see how currency FX change day to day as a percentage.

* First get the data and put it in your project root.

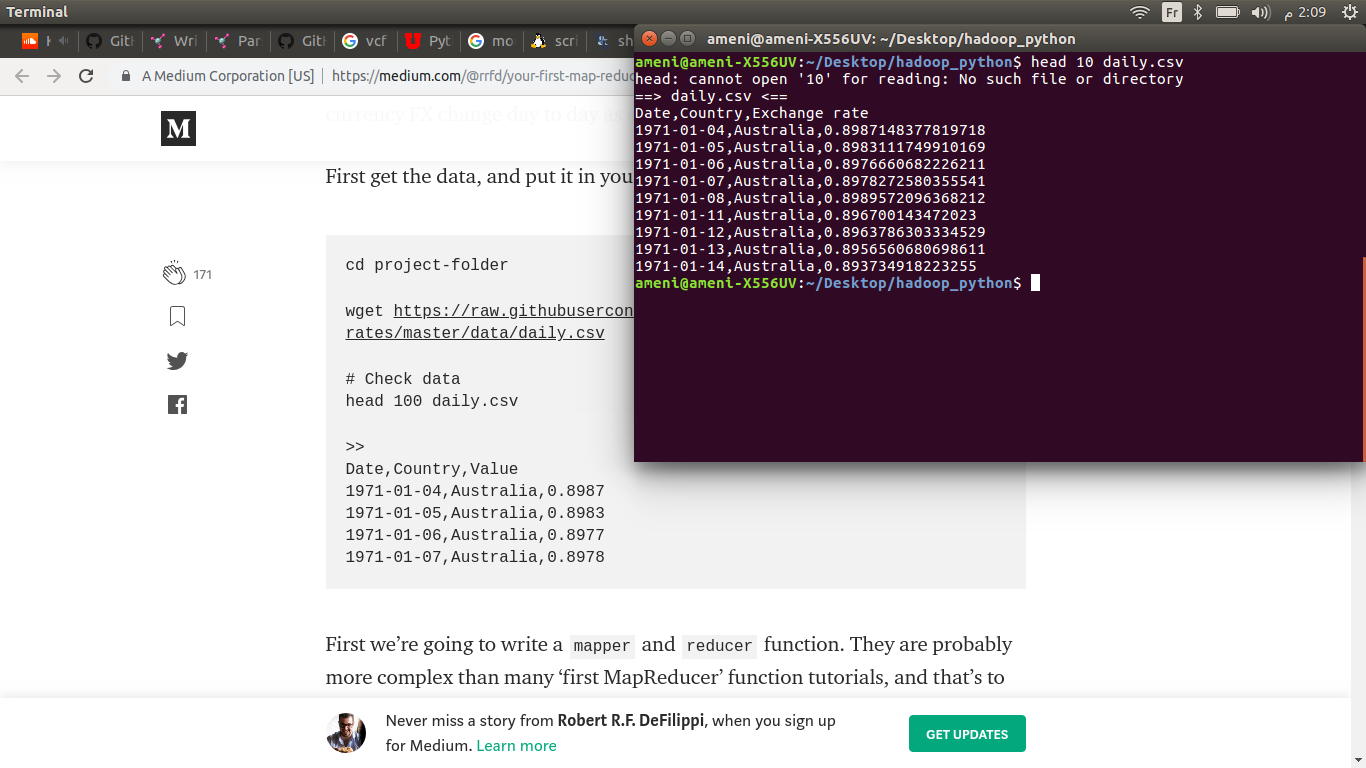
cd <project-folder>

wget <https://raw.githubusercontent.com/datasets/exchange-rates/master/data/daily.csv>



* To check data, use this command:

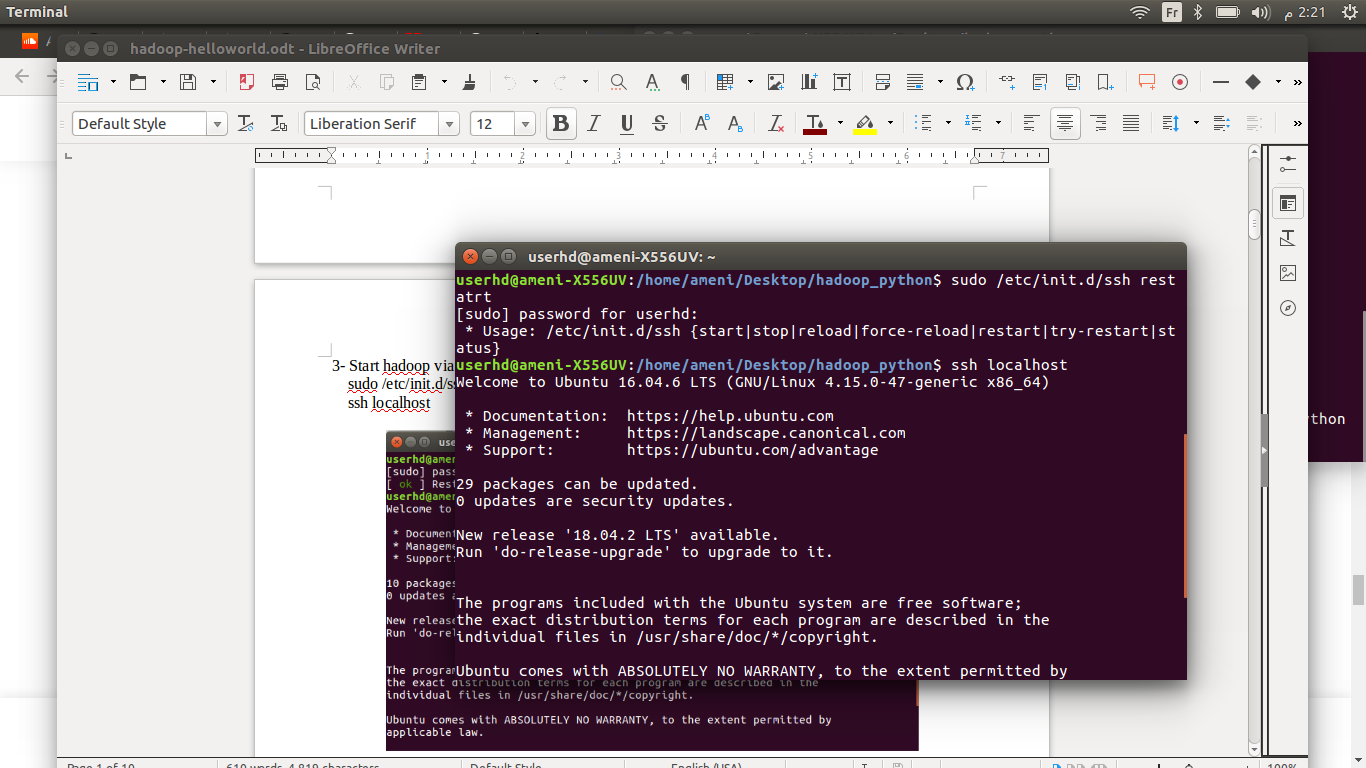
head 10 daily.csv



* Start Hadoop via ssh

sudo /etc/init.d/ssh restart

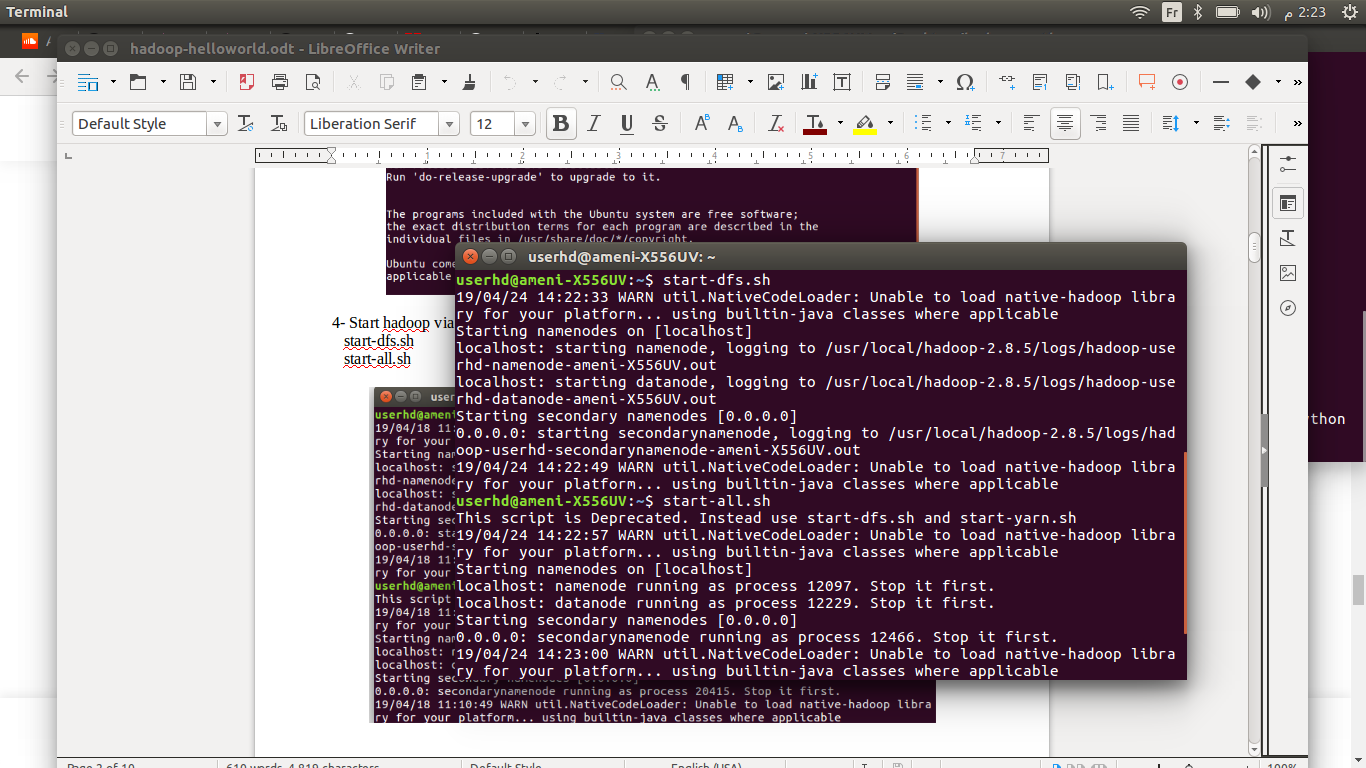
ssh localhost



* Start Hadoop via :

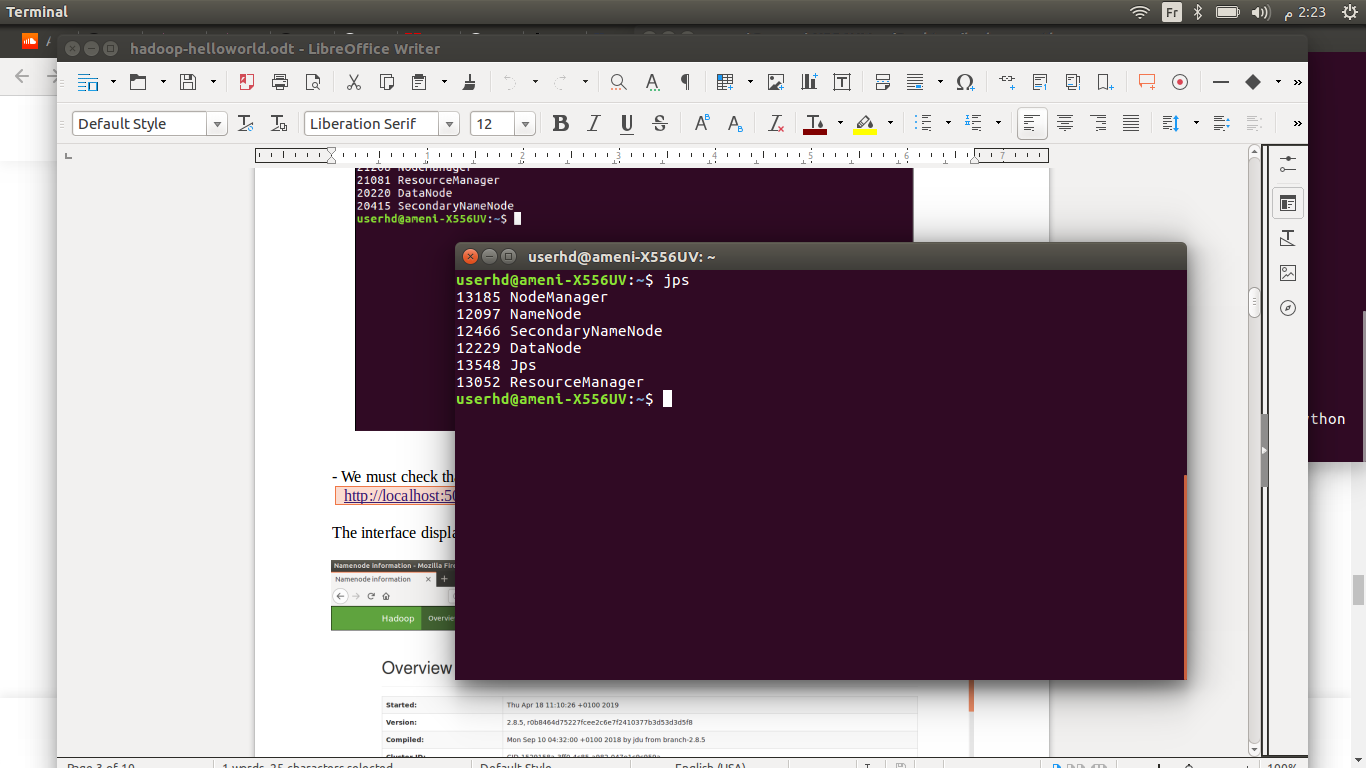
start-dfs.sh

start-all.sh



* Check Hadoop services are running :

jps



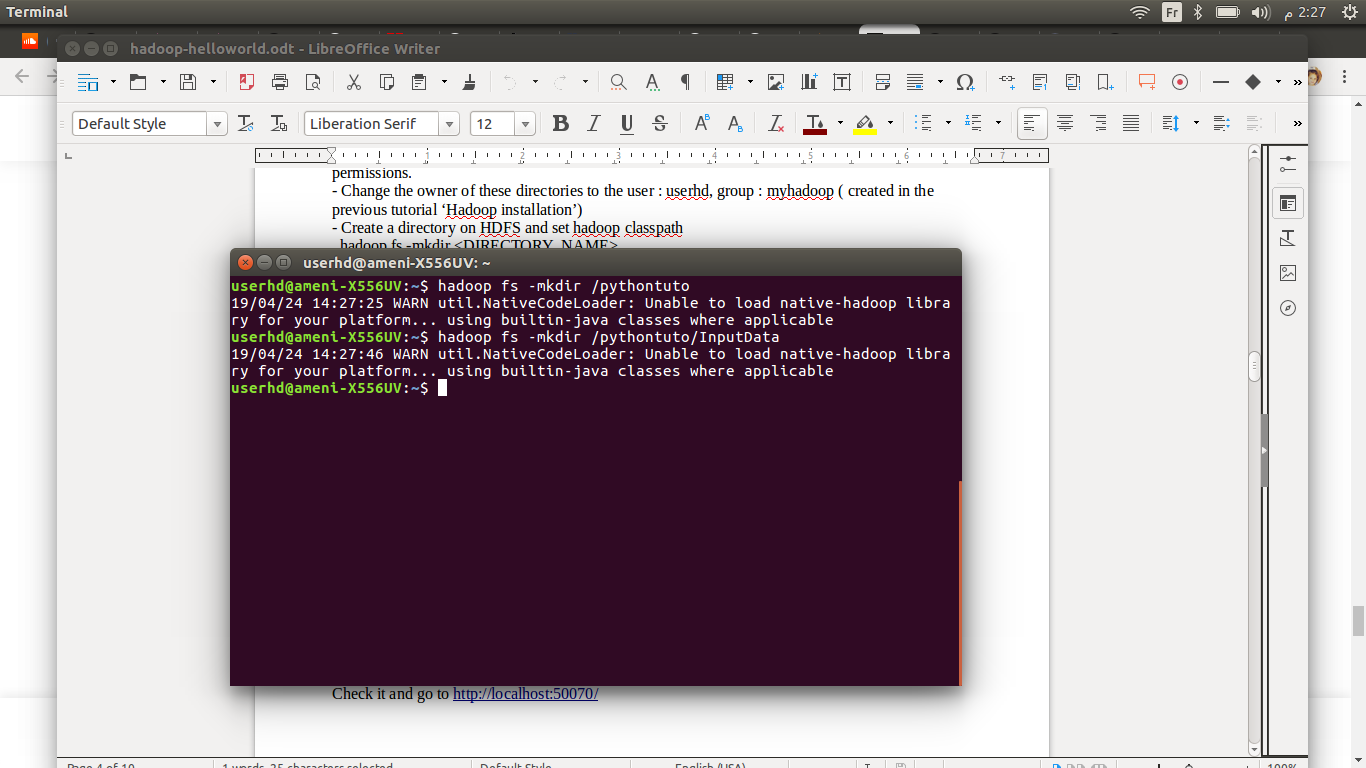
* Create a directory on **HDFS** and set Hadoop classpath :

export HADOOP\_CLASSPATH=$(hadoop classpath)

hadoop fs -mkdir <DIRECTORY\_NAME>

* Create a directory inside it for the input :

hadoop fs -mkdir <HDFS\_INPUT\_DIRECTORY>

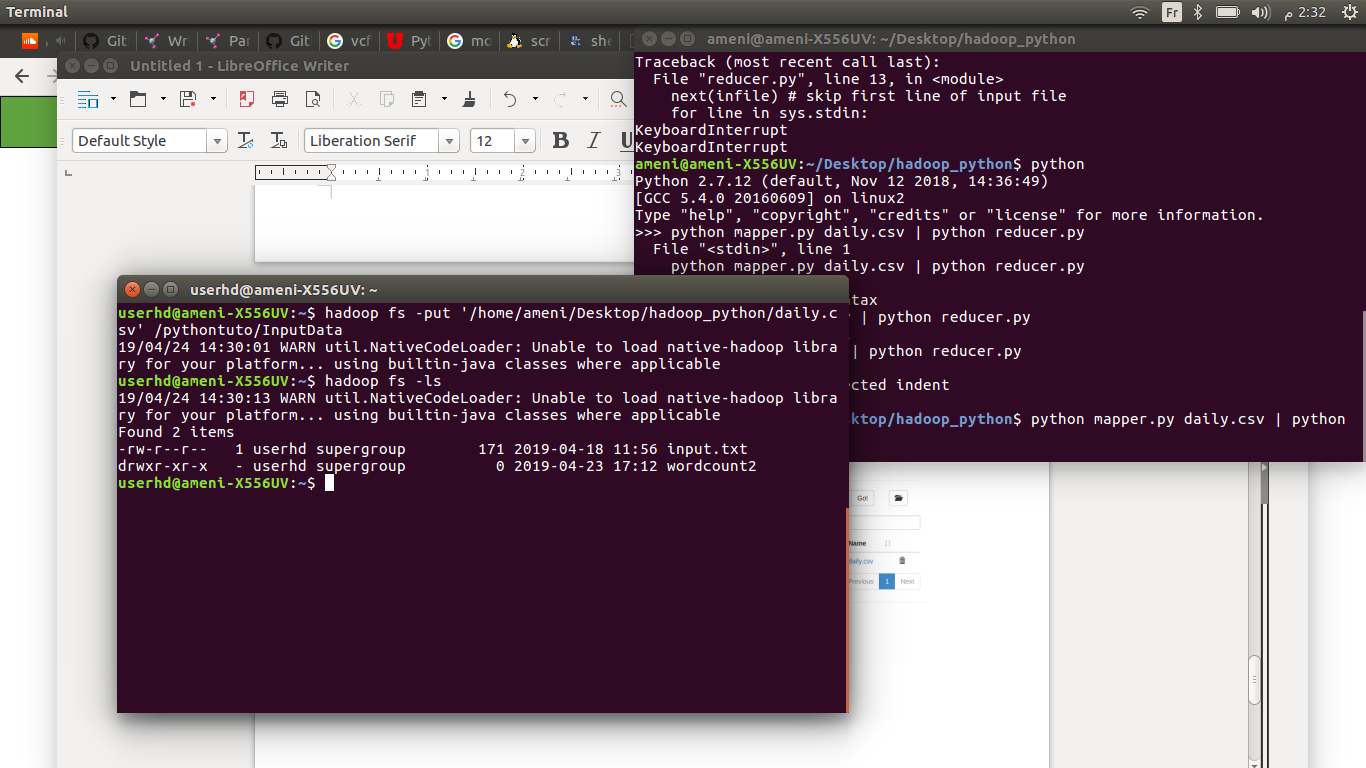


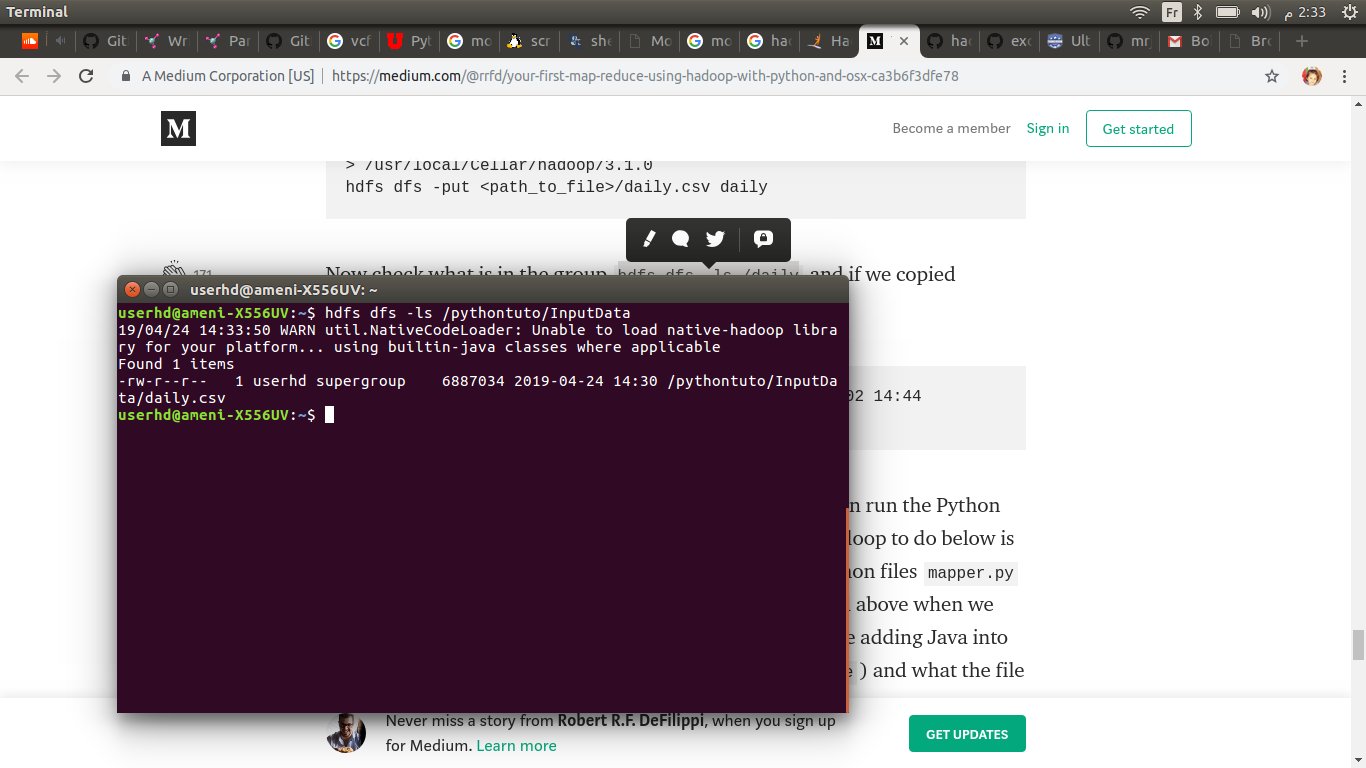
* Navigate to the folder containing the dataset and execute the command:

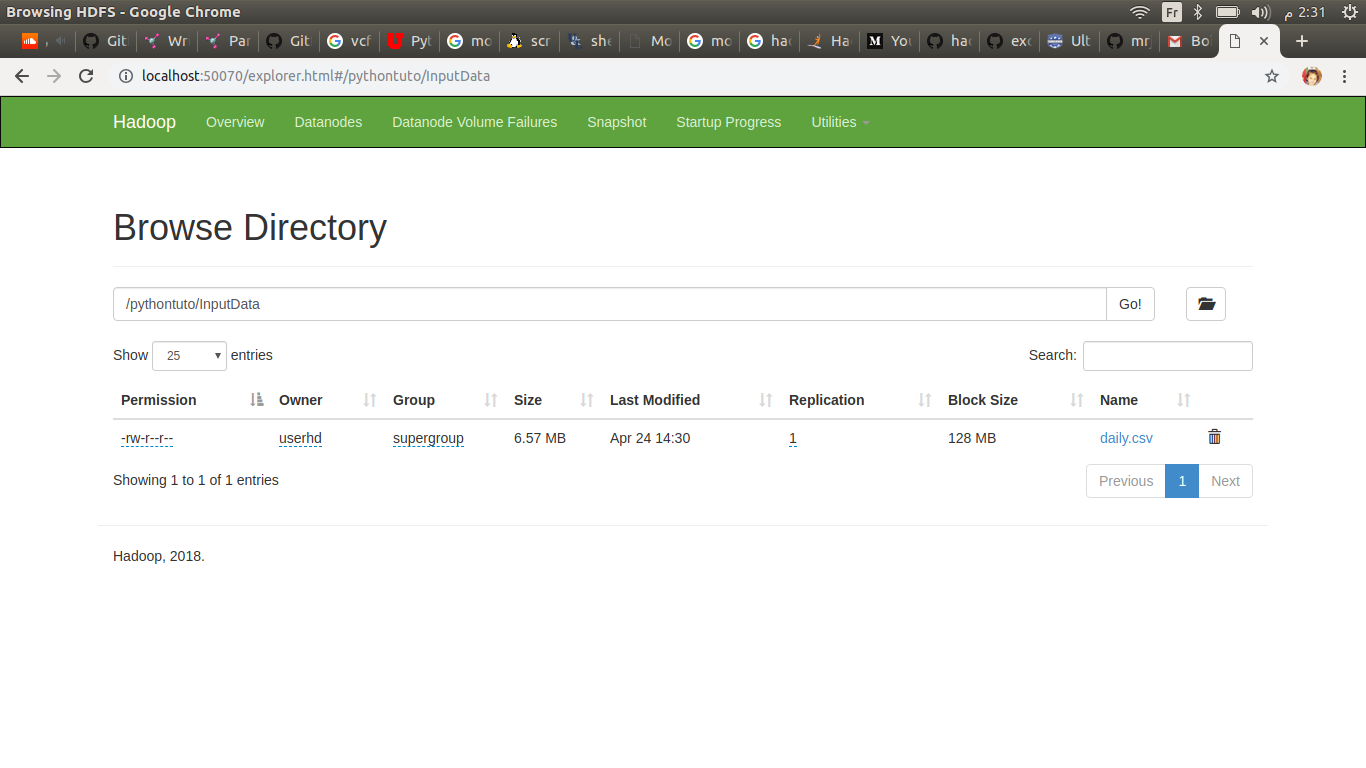
hadoop fs -put <INPUT\_FILE> <HDFS\_INPUT\_DIRECTORY>

* Check via:

hadoop fs -ls





* Check it again and go to [**http://localhost:50070/**](http://localhost:50070/)

**Step 2**

* First, we’re going to write a mapper and reducer function.
* Keep in mind, our Exchange rate column has null values, which we will take into consideration in our **mapper.py**. As well, the country is not static  so when we are evaluating the changes in the FX for a new country, we need to ignore the first appearance or else it will take the FX from the previous country and compare that with the new country.
* Let’s look at our **mapper.py** and make sure to include **#!/usr/bin/python** at the top of the script. It allows the script to be executed like a standalone executable without typing python in front of it. (I’ve included it to ensure our script runs, because sometimes Hadoop can be difficult with executables)

#!/usr/bin/python

# The Mapper

import sys

import csv

# Set local variables

iteration = 0

currentCountry = None

previousCountry = None

currentFx = None

previousFx = None

percentChange = None

currentKey = None

fxMap = []

# print "Starting mapper.py"

infile = sys.stdin

next(infile) # skip first line of input file

for line in infile:

line = line.strip()

line = line.split(',', 2)

try:

# Get data from line

currentCountry = line[1].rstrip()

if len(line[2]) == 0:

continue

currentFx = float(line[2])

if currentCountry != previousCountry:

previousCountry = currentCountry

previousFx = currentFx

previousLine = line

continue

# If country same as previous, add to map

elif currentCountry == previousCountry:

percentChange = ((currentFx - previousFx) / previousFx) \* 100.00

percentChange = round(percentChange, 2)

percentChange = percentChange

currentKey = "%s: %6.2f%%" % (currentCountry, percentChange)

# Set the array with tuple keys

fxMap.append(tuple([currentKey, 1]))

# Update Values

previousCountry = currentCountry

previousFx = currentFx

previousLine = line

# Uncomment if you want to see the output

# if iteration % 50000 == 0:

# print "Current iteration is %d" % iteration

# iteration += 1

# Handle unexpected errors

except Exception as e:

template = "An exception of type {0} occurred. Arguments:\n{1!r}"

message = template.format(type(e).\_\_name\_\_, e.args)

print "currentFx: %.2f previousFx: %.2f" % (currentFx, previousFx)

print message

sys.exit(0)

# print "mapper.py has completed with %d iterations" % (iteration - 1)

# Show the returned values

for i in sorted(fxMap):

print "%-20s - %d" % (i[0], i[1])

* And the reducer.py

#!/usr/bin/python

# The reducer

from operator import itemgetter

import sys

current\_key = None

current\_count = 0

key = None

# Import the mapped FX data data

for line in sys.stdin:

# parse the input we got from mapper.py

key, count = line.split('- ', 1)

key = key.strip()

try:

count = int(count)

except ValueError:

continue

if current\_key == key:

current\_count += count

else:

if current\_key:

print '%s\t%s' % (current\_key, current\_count)

current\_count = count

current\_key = key

if current\_key == key:

print '%s\t%s' % (current\_key, current\_count)

**Step 3**

* We have everything in its place, and we can run the Python MapReduce on our Hadoop cluster.
* What we are telling Hadoop to do below is to run Java class **hadoop-streaming** but using our python files **mapper.py** and **reduce.py** as the MapReduce process. Same as we did above when we just performed the MapReduce with Python, but now we are adding Java into the mix.
* Next, we are telling Hadoop where to find the input data (daily.csv) is located on the **hdfs** and where to put the output after the process is done. Because the output does not exist, it will be created at the time of execution.

hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.8.5.jar \

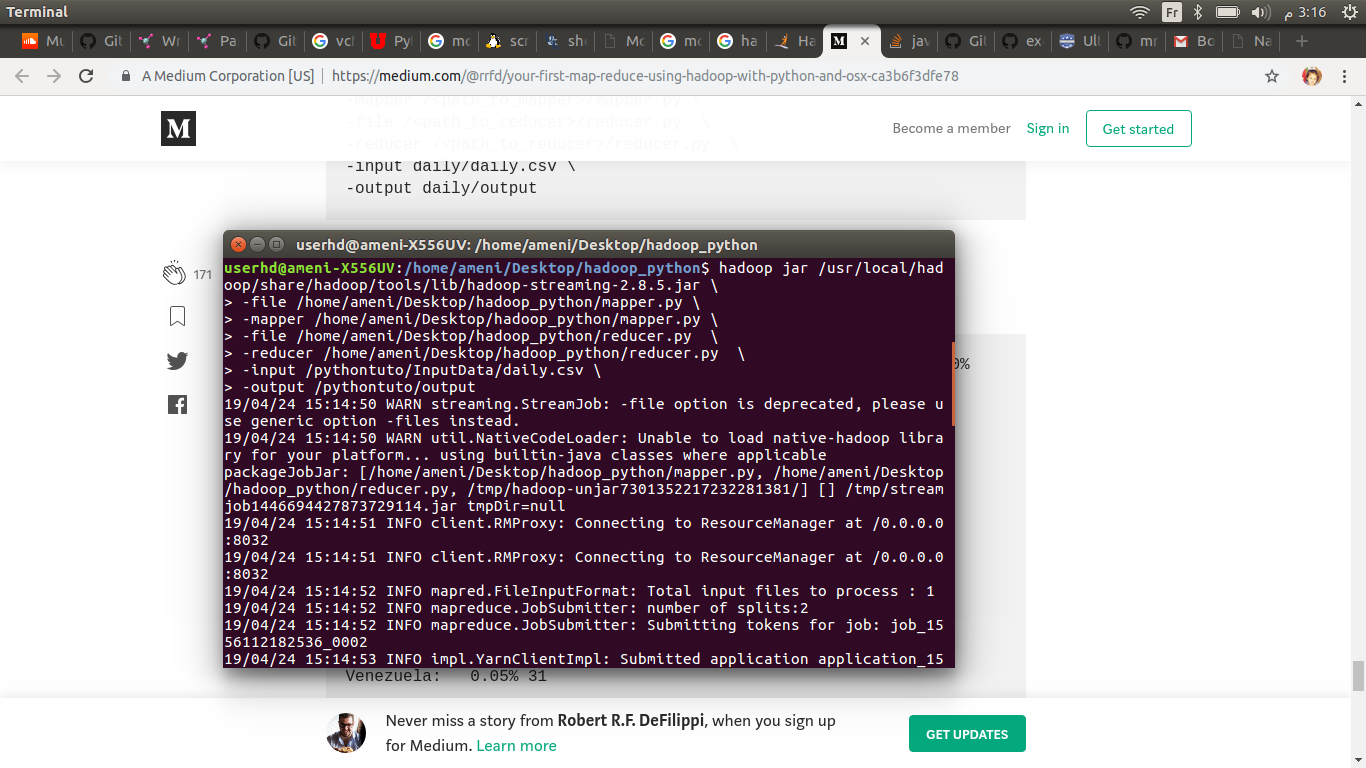
-file /<path\_to\_mapper>/mapper.py \

-mapper /<path\_to\_mapper>/mapper.py \

-file /<path\_to\_reducer>/reducer.py \

-reducer /<path\_to\_reducer>/reducer.py \-input /pythontuto/InputData/daily.csv \

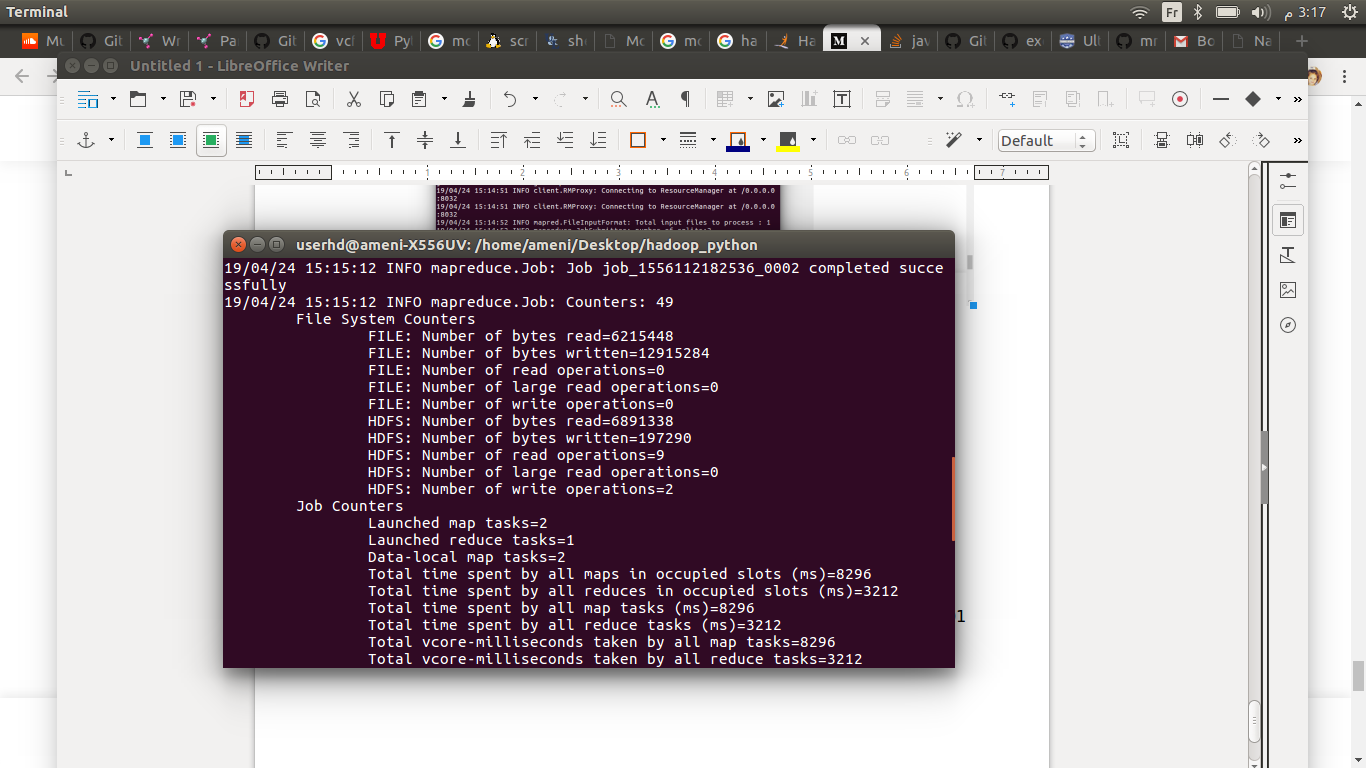
-output /pythontuto/output



* Let’s check the output to see if everything ran correctly:

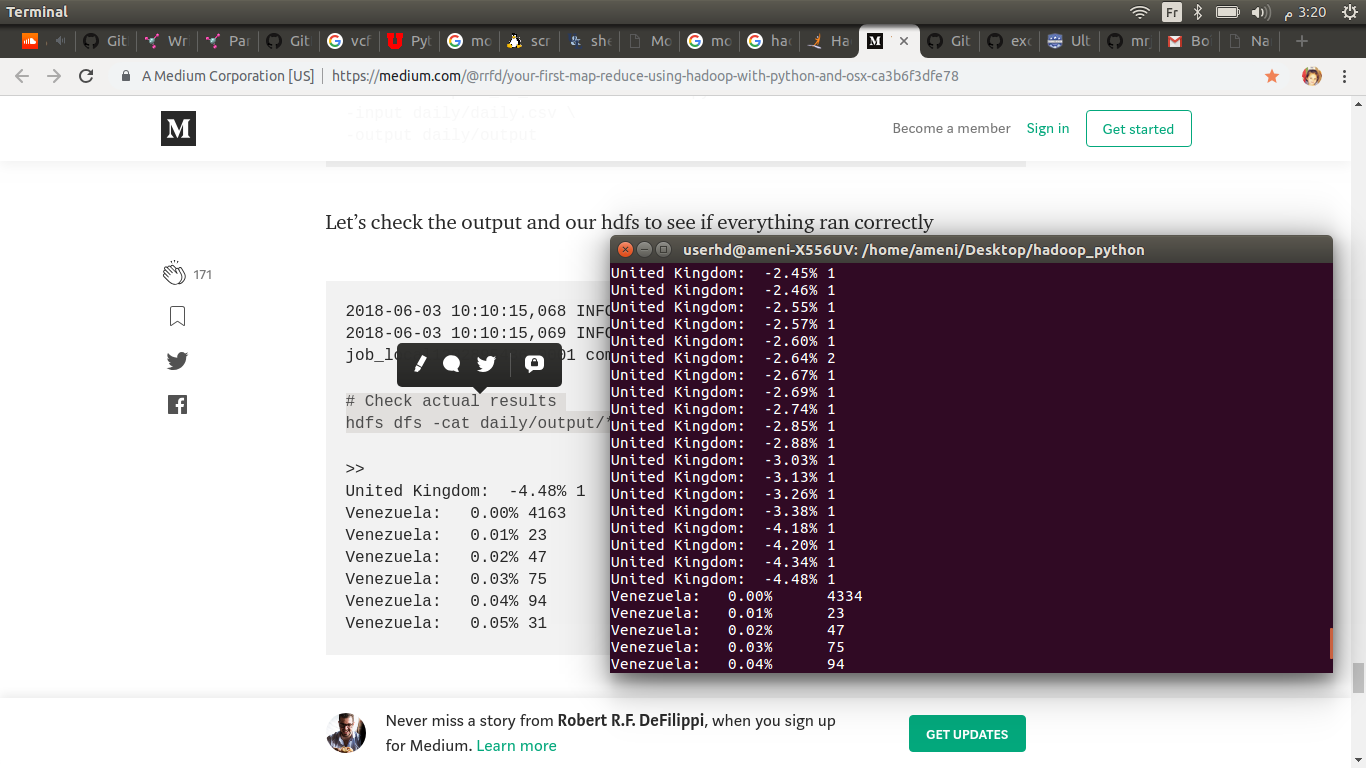
2018-06-03 10:10:15,068 INFO mapreduce.Job: map 100% reduce 100%

2018-06-03 10:10:15,069 INFO mapreduce.Job: Job job\_local1202895436\_0001 completed successfully



**Step 4**

* Now check actual results

 hdfs dfs -cat /pythontuto/output/\*