Big Data Lab – Week 3: Python Libraries & MapReduce

In this lab you will practice working with Python Libraries Also and try how to run Python MapReduce jobs jobs using Hadoop Streaming.

The python libraries you need in this lab include:

- Numpy - the fundamental package for scientific computing with Python (https://numpy.org/)
- Pandas – offers data structures and operations for manipulating numerical tables and time series (https://pandas.pydata.org/pandas-docs/stable/)
- Matplotlib - a Python 2D plotting library (https://matplotlib.org/)
- Seaborn – a Python visualization library based on Matplotlib (https://seaborn.pydata.org/)

All these libraries have been included in Anaconda 3 and Google Colab.

You can use Jupyter Notebook (which is the one utilized in task 1), Google Colab, Spyder IDE, Python shell, or any other python development environments you prefer to finish the lab task 1 & 2. For task 3, the Lint Mint VM is needed. While task 3 will be demonstrated in the live lecture session, in case some students may have difficulties to run the VM.

You can refer to the following sources:

- **Python documentation (**https://docs.python.org/3.8/)
- **Python tutorial (**https://www.tutorialspoint.com/python/**)**
- **Pandas documentation (**http://pandas.pydata.org/pandas-docs/stable/index.html)

Task 1: Import libraries into your environment

Python library is a collection of functions and methods that allows you to perform many actions without writing your code. There are many python libraries predefined which can make your coding easier. While, to use any library/package in your code, you must first make it accessible: by the process of **importing**. The **import** statement is the most common way of invoking the import machinery.

Try typing the following into a Python shell, an IDE or a Notebook:

```
nowTime = datetime.datetime.now()
```

run it and you will get a NameError:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-29bc7a89aef7> in <module>()
----> 1 nowTime = datetime.datetime.now()

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

Apparently Python does not know what *datetime* means: *datetime* is not defined.

For *datetime* (or anything else) to be considered defined, it has to be accessible from the current scope and has to satisfy one of the following conditions:

- It is a part of the default Python environment. Like *int*, *list* and *object*.
- It has been defined in the current program flow: as you wrote a *def* or a *class* or just a plain assignment statement to make it mean something.
- It exists as a separate package/library and you imported that package/library by executing a suitable import statement.

Now try the following scripts:

```
import datetime

nowTime = datetime.datetime.now()
print(nowTime.isoformat())
```

That works properly. As we imported *datetime* library in the first line, we can use the *now* function to create object and assigned it to the *nowTime* variable. Then we can access function and attributes attached to *nowTime*.

In fact, the ***import*** statement combines two operations:

- searches for the named module/ library/package
- binds the results of that search to a name in the local scope.

1. Practice with the calendar library. Enter the following code (you can ignore the comments if you want) and run it.

```
import calendar

# Sets the weekday (0 is Monday, 6 is Sunday) to start each week.
# The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
# and SUNDAY are provided for convenience.
calendar.setfirstweekday(calendar.SUNDAY)
print(calendar.firstweekday())

year = 2020
print(calendar.isleap(year))
```

2. Use the *factorial* functions in *math* library to calculate the factorial *n!* of a number *n*. For example, *5!* is *120*.

[Note: You can refer to https://docs.python.org/3.8/library/math.html for further information on *factorial, exp* and other mathematical functions in *math* library.]

3. Use the *exp* function in *math* library to calculate the value of e to the power of x, where e is the base of the natural logarithm, 2.718281828... For example, *exp(-1)= 0.367879441171*.

There are some other ways to import libraries in Python

- Define an alias to library, e.g.:

```
import math as m
```

You can now use various functions from *math* library (e.g. *factorial*) by referencing it using the alias, e.g., *m.factorial( )* as below:

```
import math as m

# refer to the "math" library with the alias "m"
print(m.factorial(5))
```

- Import the entire name space, e.g.:

```
from math import *
```

In this way, you can directly use *factorial()* without referring to *math*. However, it is not recommended to use this style of importing libraries.

4. Redo exercise 2 and 3 using the library importing ways mentioned above.

Task 2: Working with Python Libraries

The libraries you practiced above are from Python Standard Library. There are quite a few *3rd party Python libraries*. Following are some libraries you may need in data analysis.

- **NumPy** (http://www.numpy.org/) stands for Numerical Python. The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other low level languages like Fortran, C and C++

- **SciPy** (https://www.scipy.org/) stands for Scientific Python. SciPy is built on NumPy. It is one of the most useful library for variety of high level science and

engineering modules like discrete Fourier transform, Linear Algebra, Optimization and Sparse matrices.

- **Pandas** (http://pandas.pydata.org/) for structured data operations and manipulations. It is extensively used for data munging and preparation. Pandas were added relatively recently to Python and have been instrumental in boosting Python's usage in data scientist community.

- **Matplotlib** (http://matplotlib.org/) for plotting vast variety of graphs, starting from histograms to line plots to heat plots.

In this task, you will practice working with libraries from both Python standard library and 3rd parties. We will use reading and writing a CSV format file as examples.

1. Use the standard *csv* module to read tabular data in CSV format.

   First, copy the '*score.csv*' data to your working folder.

   If you are using Jupyter Notebook, you can run the following command from "Anaconda Prompt (Anaconda3)":

   ```
   jupyter notebook --notebook-dir=directory_name
   ```

   for example, the following command will change the working directory to "c:\Big_data\Labs"

   ```
   >jupyter notebook --notebook-dir="c:\Big_data\Labs"
   ```

   If you are using Google Colab, please refer to "*Mount Google drive and change working directory in Colab.pdf*" on GCUlearn to change working directory.

   Then, create the following scripts:

   ```
   import csv

   with open('score.csv', 'r') as csvfile:
       scorereader =  csv.reader(csvfile, delimiter=' ', quotechar='|')
       for row in scorereader:
           print(', '.join(row))
   ```

   Run the scripts, did you see the 'score.csv' data printed in the console window like below?

   ```
   id,first_name,last_name,age,preTestScore,postTestScore
   0,Harry, ,Potter,21,4,25000
   1,Micky,Mouse,19,24,94000
   2,Peter,,20,31,57
   3,Tina,Pong,22,,62
   4,Curz,Lee,21,,70
   ```

Note:

a. The *reader* function in *csv* module returns a reader object which will iterate over lines in the given csv file, which is 'score.csv' in this example. Each row read from the csv file is returned as a list of strings.
b. The *join* method is a string method and returns a string in which the elements of sequence have been joined by str separator, which is ', ' in this example.

2. Write data into delimited strings on the given file-like object.

Create the following scripts:

```
import csv

with open('score_2.csv', 'w') as csvfile:
    scorewriter = csv.writer(csvfile, delimiter=' ', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    scorewriter.writerow([5, 'Neil', 'Armstrong', 25, 42, 68])

with open('score_2.csv', 'r') as csvfile:
    scorereader =  csv.reader(csvfile, delimiter=' ', quotechar='|')
    for row in scorereader:
        print(', '.join(row))
```

Run the scripts, did you see the data saved in 'score_2.csv'?

Note: The *writer* function in *csv* module returns a writer object responsible for converting the user's data into delimited strings on the given file-like object., which is '*score_2.csv*' in this example.

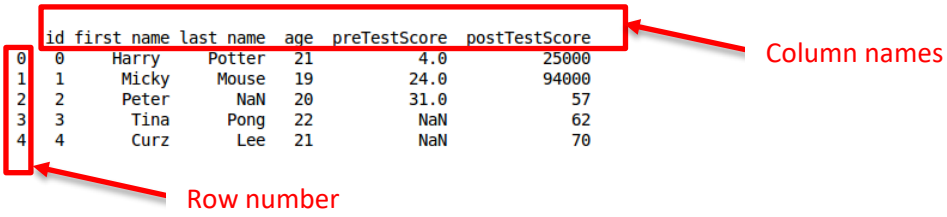
3. Load a CSV file into Pandas DataFrame.

DataFrame is a two-dimensional labelled data structure with columns of potentially different types. Arithmetic operations align on both row and column labels. You can think of it like a spreadsheet or SQL table, or a dict-like container of Series objects. It is generally the most commonly used Pandas object.

```
import pandas as pd

df = pd.read_csv('score.csv')
print(df)
```

Run the scripts, you should see output like:



	id	first_name	last_name	age	preTestScore	postTestScore
0	0	Harry	Potter	21	4.0	25000
1	1	Micky	Mouse	19	24.0	94000
2	2	Peter	NaN	20	31.0	57
3	3	Tina	Pong	22	NaN	62
4	4	Curz	Lee	21	NaN	70

Select one column or one element in the dataframe, try:

```
print(df['last_name'])
print(df['first_name'][2])
```

By default, the *header = 0* and column names are inferred from the first line of the file. If column names are not included and all the rows in the file are data, you can load the CSV file without header using '*header = None*':

```
df = pd.read_csv('score.csv', header = None)
print(df)
```

Check the difference between the print result of '*df*' below and the previous one.

```
    0           1          2    3             4             5
0  id  first_name  last_name  age  preTestScore  postTestScore
1   0       Harry     Potter   21             4          25000
2   1       Micky      Mouse   19            24          94000
3   2       Peter        NaN   20            31             57
4   3        Tina       Pong   22           NaN             62
5   4        Curz        Lee   21           NaN             70
```

4. Go through the online Pandas Tutorial and run the scripts in this tutorial: http://nbviewer.jupyter.org/urls/bitbucket.org/hrojas/learn-pandas/raw/master/lessons/01%20-%20Lesson.ipynb

   Note: depending on the version of Python you are running, you may not able to check the value of variable/object directly in the console window. One simple solution is to '*print*' the variable/object.

5. Copy the 'bank.csv' file to your working folder. Read the data in 'bank.csv' into Python. Find the maximum balance in the bank data. Print the job and age of the person who has the maximum balance.

Task 3: Running Python MapReduce jobs using Hadoop Streaming

Hadoop's native language is Java, but you can create and run MapReduce jobs with any executable or script as the mapper and/or the reducer using the Hadoop Streaming utility. In this task you will use Python to create mapper and reducer scripts. You will start by implementing a job to do a word count in text file(s), for which you are given the required Python scripts, and then you can try to solve another similar analysis problem.

The Lint Mint VM is needed to perform this task. While the task will be demonstrated in the live lecture session in case some students may have difficulties to run the VM.

Word count example - scripts

For this task you should create a directory in the local filesystem on the Linux Mint VM – these instructions assume the directory is in the *bdtech* user's home directory and is called ***mapreduce_example***.

1. Download the files ***mat.txt*** and ***holmes.txt*** from GCU Learn into that directory.

2. Create two Python scripts, ***mapper.py*** and ***reducer.py*** using the following code. Note that each of these scripts reads lines of text from the standard input, and prints some output (to the standard output).

   The mapper script splits each line into words, and for each word prints the word, as a key, to the output, followed by a tab character and 1.

   The reducer script can read each line from the output of the mapper, and splits the line into a key and a value (separated by the tab character). It counts how many times each distinct key is found, by adding the value for a key to a running total for that key if the key is the same as the previous line. This will work only if the input is sorted so that all occurrences of a particular key are in consecutive lines. The output from the reducer consists of each key with the total number of occurrences of that key.

mapper.py

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    line = line.strip()
    keys = line.split()
    for key in keys:
        value = 1
        print( "%s\t%d" % (key, value) )
```

reducer.py

```
#!/usr/bin/env python

import sys

last_key = None
running_total = 0

for input_line in sys.stdin:
    input_line = input_line.strip()
    this_key, value = input_line.split("\t", 1)
    value = int(value)

    if last_key == this_key:
        running_total += value
    else:
        if last_key:
            print( "%s\t%d" % (last_key, running_total) )
        running_total = value
        last_key = this_key

if last_key == this_key:
    print( "%s\t%d" % (last_key, running_total) )
```

Testing locally

These scripts (i.e., ***mapper.py*** and ***reducer.py***) can be run on the local filesystem without using Hadoop, which is useful to test that they work correctly. Once you are sure they work, you can then run on the Hadoop cluster (which will, of course, scale up much better if the input data is large).

3. In the terminal, change to the directory where the scripts and data are located

```
bdtech@osboxes ~ $ cd /home/bdtech/mapreduce_example/
```

4. Use the following command to list the contents of the file *mat.txt*

```
$ cat mat.txt
```

5. Enter the following command – this uses Unix pipes to pass the content of the file to the *mapper* script, pass the output of that to the *sort* command and pass the sorted output to the *reducer* script:

```
$ cat mat.txt | python mapper.py | sort | python reducer.py
```

The output should list each distinct word in the file and the count for that word (actually only one word in that file is repeated).

Running on the cluster

When you have confirmed that the scripts are working correctly, you can move the data to the cluster and run the same scripts as a MapReduce job.

6. In the terminal, change to the Hadoop application home directory.

```
$ cd /home/bdtech/Applications/hadoop-2.7.3/
```

7. Use an HDFS shell command to create a new directory ***wcinput*** on the cluster

```
$ hadoop fs -mkdir wcinput
```

then copy the text data files from the local filesystem into that directory using the command:

```
$ bin/hdfs dfs -put /home/bdtech/mapreduce_example/*.txt wcinput
```

Use HDFS shell commands to check that the files have been copied successfully.

```
$ hadoop fs -ls wcinput
```

8. Enter the following command to run a Hadoop Streaming job with your Python scripts as mapper and reducer and the file *holmes.txt* as the input. This input file is the complete text of the book "The Adventures of Sherlock Holmes", downloaded from Project Gutenberg. Actually, it's not so large that the Python scripts running locally would take too long to process, but it does give you a bit more sense of working with realistic data.

   Note that, in the following command, some lines are shown with "\" at the end which allows you to enter this rather long command over multiple lines to make it more readable as you type, but the line containing the path to the Hadoop Streaming jar can't be split in that way.

```
$ bin/hadoop \
  jar /home/bdtech/Applications/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
  -mapper "python /home/bdtech/mapreduce_example/mapper.py" \
  -reducer "python /home/bdtech/mapreduce_example/reducer.py" \
  -input "wcinput/holmes.txt" \
  -output "wcoutput02"
```

As before, you should see many lines of output logging the progress of the job, and the output of the job will be written to a directory *wcoutput02* in the

distributed filesystem (note that the *wcoutput02* folder is created as the job runs, and you should not try to direct output to a directory that already exists).

9. Use the following command to view the output (piping the output to less allows you to view it a page at a time, pressing spacebar to move on, q to exit).

```
$ bin/hdfs dfs -cat wcoutput02/* | less
```