DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Extra Credit Assignment

Efficiency of Python Code

Submitted to:

Professor Joseph Picone
ECE 3822: Software Tools for Engineers
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

April 5, 2018

Prepared by:

Abdulrahman Almusaileem
Email: tug46204@temple.edu
Jermy Joy
Email: tug02925@temple.edu

A. DESCRIPTION

In this assignment we can measure the CPU and memory efficiency of different Python scripts. The Python scripts objective was to determine the number of duplicate elements in an array. We have a total of six different scripts which complete the same task but have a different approach. Shown below are the different scripts which are tested.

```
def first_trail(numbers):
    #print "in first_trail..."
    d = {}
    for val in numbers:
        d[val] = d.get(val, 0) + 1
    return sum(d[i] > 1 for i in d)
```

```
def second_trail(numbers):
    #print "in second_trail..."
    dupVals = []
    for i in range(0, len(numbers)):
        for j in range(i+1, len(numbers)):
            if numbers[j] == numbers[i] and
numbers[j] not in dupVals:
                dupVals.append(numbers[j])
    return len(dupVals)
```

```
def third_trail(numbers):
    #print "in third_trail..."
    temp = []
    foo = 0
    numberz = set(numbers)
    for num in numberz:
        temp.append(numbers.count(num))
    for num in temp:
        if num > 1:
            foo += 1
    return foo
```

```
def forth_trail(numbers):
    #print "in forth_trail..."
    x=[]
    for n in set(numbers):
        count = numbers.count(n)
        if count > 1:
            x.append(n)
    return (len(x))
```

```
def fifth_trail(numbers):
    counter = 0
    if len(numbers) < 2:
        return counter
    else:
        numbers.sort()
        dup = 0
        for i in range(1,len(numbers)):
            if ((numbers[i-1] == numbers[i]) &
(dup == 0)):
                counter = counter + 1
                dup = 1
            elif numbers[i-1] != numbers[i]:
                dup = 0
    return counter
```

```
def sixth_trail(numbers):
    #print "in sixth_trail..."
    allDupes = [x for x in numbers if
numbers.count(x) >= 2]
    uniqueDupes = list(set(allDupes))
    numberOfDupes = len(uniqueDupes)
    return numberOfDupes
```

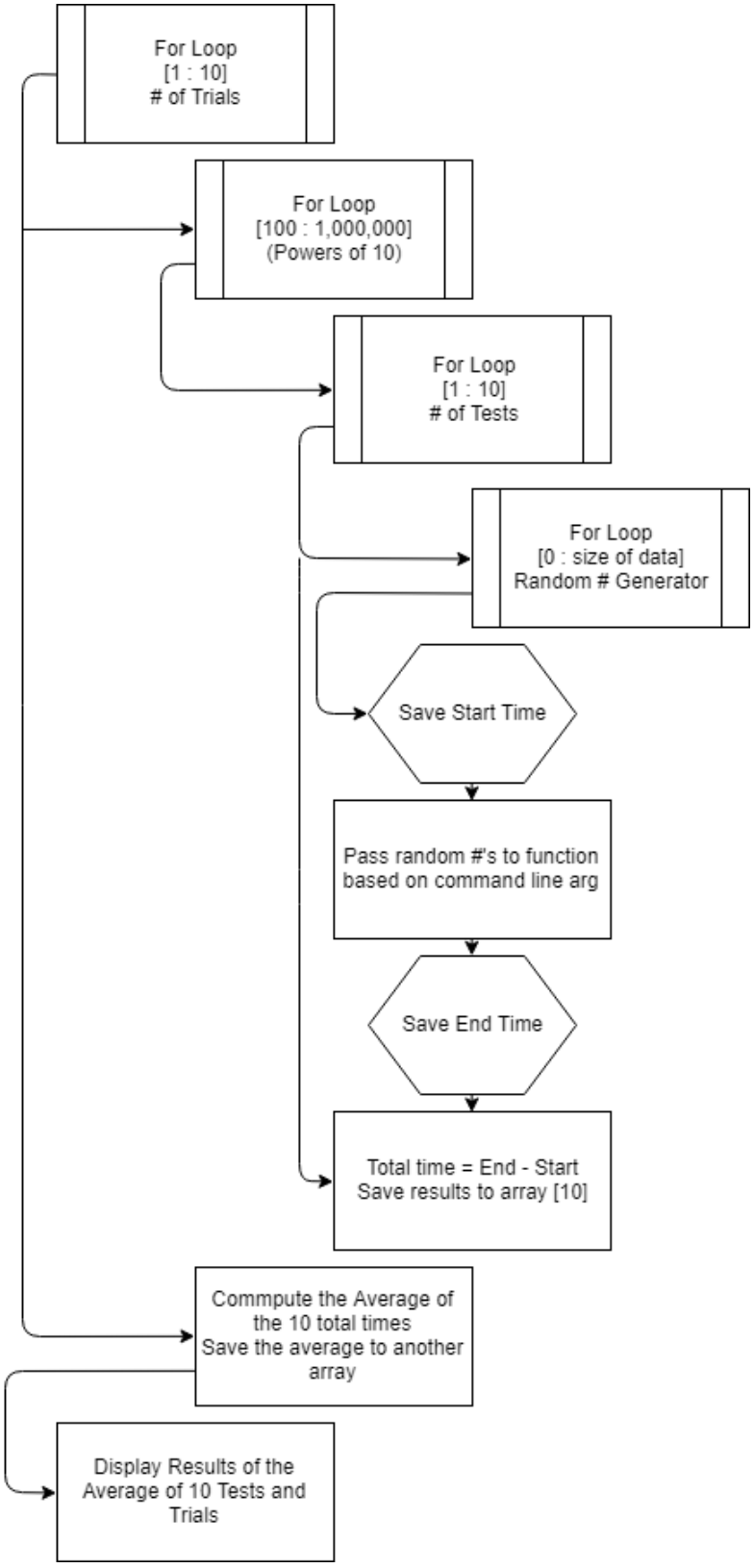
In order to produce meaningful data points, we vary the data size, one hundred to one million by powers of ten, and execute the code a total of ten times and compute the average time of the total trials. The time was computed by saving the time prior to execution and then saving the time after execution of the script was complete. The initial time was then subtracted from the final time and the resulting value was the time which was spent executing the script.

Each script was saved into a single python script and was saved as six different functions which can be called using a command line argument. For example, the first example code was called by entering the command below. The results then yield the averages of the ten trials for the five different data sizes.

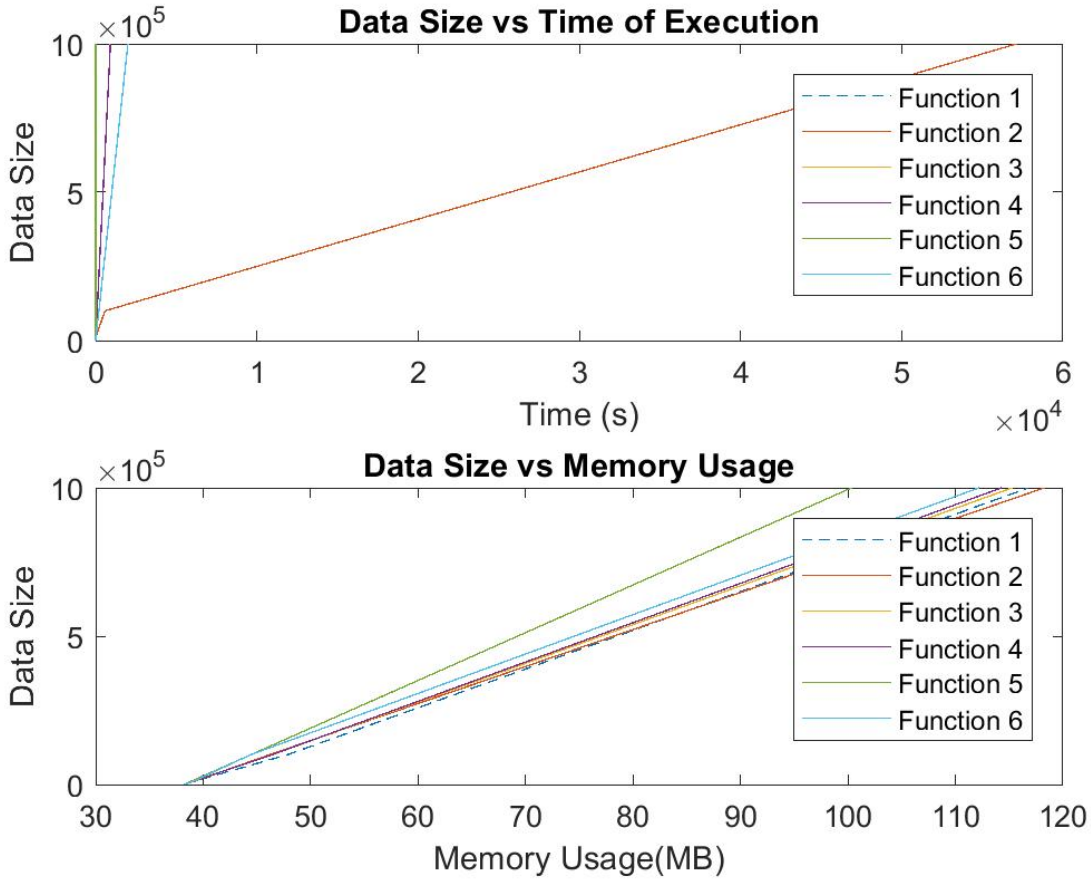
```
nedc_000_[1]: nohup python extra.py first_trail >> first.txt &
```

Using data sizes larger than 100,000 caused the execution times to be large. We executed the code on the Neuronix cluster and in the background and used the “nohup” command to ensure the script was compiling even after logging off. The results were then saved to a text file which can be read after the code is executed to completion.

The main frame of this code consisted of four nested ‘for’ loops. Shown below is an example flowchart for the process. The first loop executed the whole experiment a total of ten times. The second loop selected a data size from a corresponding array with the desired data sizes as elements. The third loop ensured there was a different data set for the function to execute ten times, timed the execution of the other function, and saved the time results to an array. The final loop was used for the random number generator ranging from 0 to the data size divided by four, this was done to ensure there are a fair number of repeats in the data set. The exact code which was used can be found in the appendix section of the report.


For Loop
[1 : 10]
# of Trials
For Loop
[100 : 1,000,000]
(Powers of 10)
For Loop
[1 : 10]
# of Tests
For Loop
[0 : size of data]
Random # Generator
Save Start Time
Pass random #'s to function based on command line arg
Save End Time
Total time = End - Start
Save results to array [10]
Commpute the Average of the 10 total times
Save the average to another array
Display Results of the Average of 10 Tests and Trials

B. RESULTS



With the use of MATLAB, we graphed the plots and it shows that function two initially had an efficient start for small data sizes, representing the steep incline at the start, and then the slope drastically decreases representing a great increase in time around the 150,000 to 200,000 random number data size mark. Furthermore, on the second subplot there represents the results of the memory data points. There was a slight reduction of memory using function code but the other functions all represented around the same amount of memory. As an employer, function 1 is the better choice when selecting a script even though it shows a slight increase in the memory sector because of the increased speed of execution. Below is an excel data sheet representing the results data size vs time.

Size	100 (S)	1000 (S)	10000 (S)	100000 (S)	1000000 (S)
Function 1	5.03E-05	0.0003307271004	0.006919734478	0.08501665831	0.9209700084
Function 2	0.000549530983	0.05136265755	5.491262841	5.71E+02	5.71E+04
Function 3	0.01472270489	0.1345752239	1.93723731	90.91240913	9.11E+02
Function 4	0.01178379059	0.1107995987	1.68309114	91.40230715	9.15E+02
Function 5	5.72E-05	0.0005646133423	0.01623624802	0.1983497071	2.712822492
Function 6	0.01450824738	0.1467009783	3.706346703	2.00E+02	2.00E+03

Size	100 (MB)	1000 (MB)	10000 (MB)	100000 (MB)	1000000 (MB)
Function 1	38.153125	38.28515625	39.04648437	47.82617188	116.8179688
Function 2	38.25554688	38.27488281	39.13054688	46.13592052	118.2918474
Function 3	38.25554688	38.26726563	38.96257813	46.38714844	115.2918383
Function 4	38.25457031	38.25066406	38.95769531	46.38851563	114.2918375
Function 5	38.25457031	38.25457031	38.83660156	44.40304688	100.3442578
Function 6	38.25984375	38.26765625	44.40304688	44.40304688	112.1948272

Above is a representation of the memory vs the data size on a spread sheet. As the size of the data increases, the value of the data size, measured in megabytes, increases.

C. SUMMARY

After completing this assignment, we as a group concluded that of these six codes, the first was the most efficient considering executing speeds for all sizes of the data provided. The second most efficient code was the fifth case. Function two, was ONLY efficient for small data sets. The second code executed faster than the third, fourth, and sixth codes for the data sizes varying from (100,1000) numbers. This code is only good for the quiz, where the random numbers are apart of only a small set. After increasing the data size to 10,000+, the time for execution for the second code increased drastically, deeming to be the most inefficient of the six codes for larger data sets. The third most efficient code provided, considering all data sets, was the third code. The fourth most efficient function provided was the fourth, the fifth most efficient was the third and finally, the least efficient code which was executed was the second.

Memory wise, these functions do not vary drastically amongst each other as the size increases. As the size increases the memory also increases. Each of the functions take up about the same amount of memory when executing the code for the different data sizes.

When analyzing the code for the second code, there are two nested ‘for’ loops and an if statement within those loops. This takes a much longer time to execute, especially when the script is expected to index through an array which is roughly one million elements long. Furthermore, the first code implements the use of a dictionary which can index much faster considering it does not index through each element necessarily. The dictionary method quickly executes since there is no comparing every individual element within the stored data.

In conclusion, when coding in python, coding efficiently is what creates a separation between expert coders and amateur coders, this assignment being an example. These six codes all accomplished the same goal but timing wise, as the data set became larger, the code began to take an exponentially longer time.

D. APPENDIX

```python
# /usr/bin/env python
import os
import sys
import random
from timeit import default_timer as time
import numpy as np
from memory_profiler import profile, memory_usage

def first_trail(numbers): # first function
    d = {}
    for val in numbers:
        d[val] = d.get(val, 0) + 1
    return sum(d[i] > 1 for i in d)

def second_trail(numbers): # second function
    dupVals = []
    for i in range(0, len(numbers)):
        for j in range(i+1, len(numbers)):
            if numbers[j] == numbers[i] and numbers[j] not in dupVals:
                dupVals.append(numbers[j])
    return len(dupVals)

def third_trail(numbers): # third function
    temp = []
    foo = 0
    numberz = set(numbers)
    for num in numberz:
        temp.append(numbers.count(num))
    for num in temp:
        if num > 1:
            foo += 1
    return foo

def forth_trail(numbers): # fourth function
    x=[]
    for n in set(numbers):
        count = numbers.count(n)
        if count > 1:
            x.append(n)
    return (len(x))

def fifth_trail(numbers): # fifth function
    counter = 0
    if len(numbers) < 2:
        return counter
    else:
        numbers.sort()
        dup = 0
        for i in range(1,len(numbers)):
            if ((numbers[i-1] == numbers[i]) & (dup == 0)):
                counter = counter + 1
                dup = 1
            elif numbers[i-1] != numbers[i]:
                dup = 0
    return counter

def sixth_trail(numbers): # sixth function
    allDupes = [x for x in numbers if numbers.count(x) >= 2]
    uniqueDupes = list(set(allDupes))
    numberOfDupes = len(uniqueDupes)
    return numberOfDupes
```

```
def main(argv): # main code
    code_dict ={ # dictionary for calling the functions based on the input arg
        "first_trail" : first_trail,   # first code
        "second_trail" : second_trail, # second code
        "third_trail": third_trail,    # third code
        "forth_trail" : forth_trail,   # forth code
        "fifth_trail" : fifth_trail,   # fifth code
        "sixth_trail" : sixth_trail    # sixth code
    }
    all_avr = [] # array for storing the final results
    data = [100, 1000, 10000, 100000, 1000000] # The data sizes for the tests
    for trail in range(0,10): # Loop for the ten tests
        tmp_avr = [] # array for storing the average of the average times
        for data_size in data: # for loop collecting the data size from the array
                avr = [] # array for storing the average of the times
                for test in range(0,10): # for loop for 10trials
                    rand_data = [] # array for storing the random data
                    for index in range(0,data_size): # create (data_size) random numbers
                        rand_data.append(random.randint(0,data_size/4)) # Generate random numbers
                    print len(rand_data) # printing the data size
                    print("Timer turned on!") # Showing timer turned on
                    ti = time() # Get the time of the start of execution
                    code_dict[argv[1]](rand_data) # dictionary for the function call
                    tf = time() # Get the time of the end of execution
                    pid = os.getpid() # memory function
                    mem_usage = memory_usage(pid) # memory
                    mem.append(mem_usage) # memory usage
                    print("Timer turned off!") # Showing timer turned off
                    inner_t = tf - ti # Expression for the time of execution
                    avr.append(inner_t) # saving the time to an array
                tmp_avr.append(np.mean(avr)) # save the average of the ten trial values to array
                 mem_avr.append(np.mean(mem))# array for storing the average memory value
        print "tmp {}".format(tmp_avr)
        print len(tmp_avr) # print the length of the time array
        all_avr.append(np.mean(tmp_avr)) # save the results of the average to an array
    print "\nall {}".format(all_avr) # print the results
    print len(all_avr) # print length of the answer array
if __name__ == "__main__":
    main(sys.argv)
```

Below is the code which was used on MATLAB to plot the results

```
clc;
clear;
% Software tools extra credit efficiency plots
%% given the average of the times from the python code
% Time result vectors
% 100, 1000, 10000, 100000, 1000000
time_first_code = [5.03063201904e-
05,0.000330727100372,0.006919734478,0.0850166583061,0.920970008373]; % All data collected
time_second_code = [0.00054953098297119143, 0.051362657546997072,
5.4912628412246702,5.712591812224671e+02,5.712804126347671e+04]; % All data collected
time_third_code = [0.014722704887390137, 0.1345752239227295,
1.9372373104095459,90.912409129142745,9.112475566145275e+02]; % All data collected
time_fourth_code = [0.011783790588378907, 0.11079959869384766,
1.6830911397933961,91.402307152748122,9.151470526574811e+02]; % All data collected
time_fifth_code = [5.72299957275e-
05,0.000564613342285,0.0162362480164,0.198349707127,2.71282249212]; % All data collected
time_sixth_code = [0.014508247375488281, 0.14670097827911377,
3.7063467025756838,1.997105785822868e+02,1.998979512256610e+03]; % All data collected
% Memory result vectors

mem_first_code = [38.153125000000003, 38.28515625, 39.046484374999999, 47.826171875,
116.81796875000001]; % All data collected
mem_second_code =
[38.255546875,38.2748828125,39.130546875,46.1359205203,118.291847362718]; % all data
collected
mem_third_code = [38.255546875,38.267265625,38.962578125,46.3871484375,115.2918382759]; %
all data collected
mem_fourth_code = [38.2545703125,38.2506640625,38.9576953125,46.388515625,114.2918374628];
% all data collected
mem_fifth_code = [38.2545703125,38.2545703125,38.8366015625,44.403046875,100.344257813]; %
All data collected
mem_sixth_code = [38.25984375,38.26765625,39.045,44.403046875,112.1948271938]; % all data
collected
% Data Set Size

Data_size_5 = [100,1000,10000,100000,1000000];

figure(1);
clf;
subplot(2,1,1)
plot(time_first_code,Data_size_5,time_second_code,Data_size_5,time_third_code,Data_size_5,
time_fourth_code,Data_size_5,time_fifth_code,Data_size_5,time_sixth_code,Data_size_5)
title('Data Size vs Time of Execution');
xlabel('Time (s)')
ylabel('Data Size')
legend('Function 1','Function 2','Function 3','Function 4','Function 5','Function 6')

subplot(2,1,2)
plot(mem_first_code,Data_size_5,mem_second_code,Data_size_5,mem_third_code,Data_size_5,mem
_fourth_code,Data_size_5,mem_fifth_code,Data_size_5,mem_sixth_code,Data_size_5)
title('Memory Usage vs Time of Execution');
xlabel('Memory Usage(MB)');
ylabel('Data Size');
legend('Function 1','Function 2','Function 3','Function 4','Function 5','Function 6')
```