CSE 337: Scripting Languages (Fall 2019)

Unit Homework #1

Assignment Due: Tuesday, December 3, 2019, by 11:59 PM

Assignment Objectives

By the end of this assignment you should be able to design, code, run and test original Python programs that use multiple functions (spread across multiple files) to analyze and report on the contents of various data files.

Note: This assignment describes several functions that work together to form a final, cohesive program. The starter code for these functions is specially designed to work together. Specifically, Parts III and IV may use your solutions to Parts I and II in order to operate correctly. We have already set up the starter code files to correctly link everything together; **you MUST keep ALL of those .py files, as well as the sample data files, together in the same folder.**

Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct (**note that the test cases we give you in the starter files are just examples; we will use different test inputs to grade your work!**). **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change anything related to a function header, the grading program will reject your solution as incorrect.

Directions

Solve each of the following problems to the best of your ability. The automated grading system will execute your solution to each problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 4 problems, and is worth a total of 60 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your work in a single file. If you do so, we will not grade your work and you will receive a 0.
- ⚠ Every function **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Every function must explicitly ***return*** its final answer; the grading program will ignore anything that your code prints out. Likewise, do **NOT** use `input()` anywhere before the `if __name__ == "__main__":` statement); your functions should get all of their input from their parameters. Programs that crash will likely receive a failing grade, so test your code thoroughly before you submit it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your completed work as directed by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time, or that is submitted before you have signed the course's Academic Honesty agreement.
- ⚠ **ALL** code that you submit (for each problem) **MUST** be your own work! You may not receive assistance from or share any materials with anyone else, except for the instructor and the (current) course TAs.

Log File Analysis

Many servers online face constant probes from would-be attackers trying to find and exploit security vulnerabilities. One such attack vector attempts to connect to a server via the SSH (Secure SHell) protocol, with the goal of discovering a user account that has a weak or easily-guessed password. Fortunately, most servers keep a log of unsuccessful connection attempts. For this problem, we will develop a program that extracts this information from a log file and analyzes it to determine the most common origin points for these connection failures, as well as the server's overall percentage of failed connection attempts.

Part I: Finding the Highest-Valued Dictionary Key (10 points)

Function Summary

Parameters: This function takes one argument: a non-empty dictionary. The dictionary's keys are (assumed to be) strings; more importantly, every key has a non-negative (meaning 0 or greater) integer as its corresponding value.

Return Type: If one key has the largest value, then this function returns that key. If two or more keys are tied for the largest value, this function returns a list that contains **ALL** of the keys that are tied for the largest value (and only those keys).

(Place your answer to this problem inside the file named "highest_value_key.py")

Complete the `highest_value_key()` function, which takes a non-empty Python dictionary (where every key corresponds to a non-negative (0 or greater) integer value) as its only argument. The function examines the contents of the dictionary and returns the key or keys with the highest value. You may assume that the dictionary will never contain two or more different "highest values".

If only one key has the highest value, the function returns only that key; if two or more keys are tied for the highest value, the function returns a list that contains **ALL** of the tied keys (and **ONLY** those keys). Note that, for grading purposes, the specific ordering of the keys within your list doesn't matter.

Examples:

Function Call	Return Value
`highest_value_key({18: 8, "Zeus": 8, 1: 4, "Hermes": 25, "Ares": 7, 2: 7})`	`Hermes`
`highest_value_key({10: 23, "Hera": 8, 11: 23, 2: 13})`	`[10, 11]`
`highest_value_key({12: 21, 19: 2, "Hermes": 15, 6: 11, 17: 10, 16: 2, 13: 22, 18: 25, 11: 4, 15: 15, 2: 20, 9: 10, 20: 17})`	`18`
`highest_value_key({15: 20, 6: 22, "Ares": 22, 11: 21})`	`[6, "Ares"]`

Part II: Extracting Data from a System Log File (20 points)

Function Summary

Parameters: A string representing the name of a (plain text) file. This string also includes any necessary path information for that data file.

Return Type: A dictionary whose keys are strings corresponding to IP addresses in *dotted-quad* format. Each key has a positive integer (i.e., ≥ 1) as its value.

(Place your answer to this problem inside the file named "count_failed_addresses.py")

Every refused or declined connection creates a series of (consecutive) error or status messages in the system log. For example, here is a (line-wrapped) log fragment that describes two failed connection attempts (the first part of the first failed connection is unfortunately cut off):

```
Sep 22 07:35:03 allv23 sshd[4996]: Received disconnect from 36.108.170.241 port
    53004:11: Bye Bye [preauth]
Sep 22 07:35:03 allv23 sshd[4996]: Disconnected from 36.108.170.241 port 53004
    [preauth]
Sep 22 07:35:15 allv23 sshd[5139]: Invalid user gmod from 149.56.46.220
Sep 22 07:35:15 allv23 sshd[5139]: input_userauth_request: invalid user gmod
    [preauth]
Sep 22 07:35:15 allv23 sshd[5139]: pam_unix(sshd:auth): check pass; user unknown
Sep 22 07:35:15 allv23 sshd[5139]: pam_unix(sshd:auth): authentication failure;
    logname= uid=0 euid=0 tty=ssh ruser= rhost=149.56.46.220
Sep 22 07:35:17 allv23 sshd[5139]: Failed password for invalid user gmod
    from 149.56.46.220 port 53788 ssh2
Sep 22 07:35:17 allv23 sshd[5139]: Received disconnect from 149.56.46.220
    port 53788:11: Bye Bye [preauth]
Sep 22 07:35:17 allv23 sshd[5139]: Disconnected from 149.56.46.220 port 53788
    [preauth]
```

In both cases, we are only interested in the IP address that tried (and failed) to connect. The exact series of console messages will vary based on the specific details of the connection attempt, but every logged failure features a final line with the substring "`Disconnected from <numeric IP address>`" (ignore anything after the address).

Complete the `count_failed_addresses()`, which takes a single string argument representing the name of a log file (or fraction thereof). This function processes the file one line at a time, looking for lines containing the substring listed in the preceding paragraph; for each such line, the function should extract the dotted-quad IP address (e.g., 149.56.46.220) from the line. The function should maintain an initially-empty dictionary to track the number of times each IP address has been seen in a failed connection attempt. If this is the first time that this address has been seen, add it to a dictionary as a new key, with a value of 1. If this address has been seen before, add 1 to its value in the dictionary (Python's `setdefault()` method may be helpful). Upon finishing, the function should return the dictionary.

Examples:

Function Call	Return Value
`count_failed_addresses ("test-log-1.txt")`	`{"149.56.46.220": 8, "114.33.233.226": 12, "213.32.52.1": 4, "36.108.170.241": 30, "82.196.4.46": 3, "68.183.161.41": 12, "180.168.141.246": 1, "112.186.77.74": 1, "175.211.116.238": 1, "167.71.194.222": 1}`
`count_failed_addresses ("test-log-2.txt")`	`{"36.108.170.241": 34, "68.183.161.41": 17, "114.33.233.226": 14, "167.71.194.222": 14, "112.186.77.74": 2, "175.211.116.238": 2, "180.172.186.102": 1, "195.201.143.162": 1}`
`count_failed_addresses ("test-log-3.txt")`	`{"114.33.233.226": 36, "36.108.170.241": 13, "195.201.143.162": 44, "167.71.194.222": 33, "68.183.161.41": 23, "175.211.116.238": 1, "112.186.77.74": 1, "159.89.225.82": 28, "185.105.238.199": 5, "91.222.195.26": 7, "85.169.71.119": 1, "14.215.46.94": 1, "159.89.10.77": 3}`

Part III: Consolidating and Identifying Attack Sources (20 points)

Function Summary

Parameters: A single string argument representing the name of a log file (or fraction thereof)

Return Type: A dictionary whose keys are strings representing three "octets" of a dotted-quad IP address, and whose values are integers.

(Place your answer to this problem inside the file named "consolidate_sources.py")

Complete the `consolidate_sources()` function, which takes a single argument: a single string argument representing the name of a log file (or fraction thereof). This function creates and returns a dictionary like the one generated by `count_failed_addresses()` from the previous problem (you may reuse your code from that problem in your solution), except the keys from the source dictionary have been combined on the basis of their first three numbers ("octets") and their values have been added together.

For example, if the source dictionary contained the key-value pairs `"159.89.10.82"`: 28 and `"159.89.10.77"`: 3, we would ignore the final value in the address (the machine number) and combine them into the single key-value pair `"159.89.10"`: 31 (to record a total of 31 attacks from the 159.89.10.* network block.

Part IV: Identifying the Most Frequent Attacker (10 points)

Function Summary

Parameters: A string representing the name of a (plain text) file. This string also includes any necessary path information for that data file.

Return Type: If one key has the largest value, then this function returns that key. If two or more keys are tied for the largest value, this function returns a list that contains **ALL** of the keys that are tied for the largest value (and only those keys).

(Place your answer to this problem inside the file named "most_frequent_attacker.py")

Complete the `most_frequent_attacker()` function, which takes one argument: a string representing the name of a (plain text) file. This string also includes any necessary path information for that data file. This function returns the network block (or blocks) with the single highest number of attacks (failed connection attempts). If there is a single "highest" value, this function just returns that partial address as a string; if two or more partial addresses are tied for the highest value, the function should return a list containing all of the keys that are tied for the "top" position. **HINT:** Use your solutions to Parts I and III to drastically simplify this problem!