Handling Files

**Introduction**

For this lab, imagine you are an IT Specialist at a medium-sized company. The Human Resources Department at your company wants you to find out how many people are in each department. You need to write a Python script that reads a CSV file containing a list of the employees in the organization, counts how many people are in each department, and then generates a report using this information. The output of this script will be a plain text file. We will guide you through each step of the lab.

**Prerequisites**

We have created the employee list for you. Navigate to the data directory using the following command:

cd data

To find the data, list the files using the following command:

ls

You can now see a file called **employees.csv**, where you will find your data. You can also see a directory called scripts. We will write the python script in this directory.

To view the contents of the file, enter the following command:

cat employees.csv

Let's start by writing the script. You will write this python script in the scripts directory. Go to the scripts directory by using the following command:

cd ~/scripts

Create a file named **generate\_report.py** using the following command:

nano generate\_report.py

You will write your python script in this **generate\_report.py** file. This script begins with a line containing the #! character combination, which is commonly called hash bang or shebang, and continues with the path to the interpreter. If the kernel finds that the first two bytes are #! then it uses the rest of the line as an interpreter and passes the file as an argument. We will use the following shebang in this script:

#!/usr/bin/env python3

**Convert employee data to dictionary**

The goal of the script is to read the CSV file and generate a report with the total number of people in each department. To achieve this, we will divide the script into three functions.

Let's start with the first function: *read\_employees()*. This function receives a CSV file as a parameter and returns a list of dictionaries from that file. For this, we will use the CSV module.

The CSV module uses classes to read and write tabular data in CSV format. The CSV library allows us to both read from and write to CSV files.

Now, import the CSV module.

import csv

Define the function *read\_employees*. This function takes file\_location (path to employees.csv) as a parameter.

def read\_employees(csv\_file\_location):

Open the CSV file by calling **open** and then **csv.DictReader**.

DictReader creates an object that operates like a regular reader (an object that iterates over lines in the given CSV file), but also maps the information it reads into a dictionary where keys are given by the optional *fieldnames* parameter. If we omit the *fieldnames* parameter, the values in the first row of the CSV file will be used as the keys. So, in this case, the first line of the CSV file has the keys and so there's no need to pass *fieldnames* as a parameter.

We also need to pass a dialect as a parameter to this function. There isn't a well-defined standard for comma-separated value files, so the parser needs to be flexible. Flexibility here means that there are many parameters to control how csv parses or writes data. Rather than passing each of these parameters to the reader and writer separately, we group them together conveniently into a dialect object.

Dialect classes can be registered by name so that callers of the CSV module don't need to know the parameter settings in advance. We will now register a dialect **empDialect**.

csv.register\_dialect('empDialect', skipinitialspace=True, strict=True)

The main purpose of this dialect is to remove any leading spaces while parsing the CSV file.

The function will look similar to:

employee\_file = csv.DictReader(open(csv\_file\_location), dialect = 'empDialect')

You now need to iterate over the CSV file that you opened, i.e., employee\_file. When you iterate over a CSV file, each iteration of the loop produces a dictionary from strings (key) to strings (value).

Append the dictionaries to an empty initialised list **employee\_list** as you iterate over the CSV file.

employee\_list = []

for data in employee\_file:

employee\_list.append(data)

Now return this list.

return employee\_list

To test the function, call the function and save it to a variable called employee\_list. Pass the path to employees.csv as a parameter to the function. Print the variable employee\_list to check whether it returns a list of dictionaries.

employee\_list = read\_employees('<file\_location>')

print(employee\_list)

Replace <file\_location> with the path to the employees.csv (this should look similar to the path */home/<username>/data/employees.csv*). Replace <username> with the one mentioned in Connection Details Panel at left hand side.

Save the file by clicking Ctrl-o, Enter, and Ctrl-x.

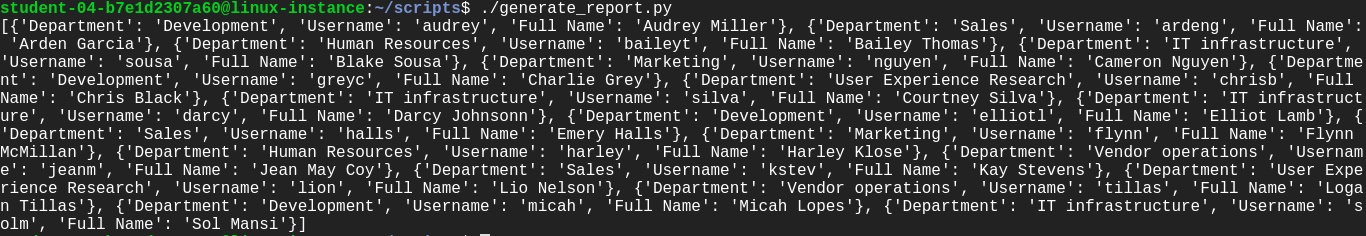
For the file to run it needs to have execute permission (x). Let's update the file permissions and then try running the file. Use the following command to add execute permission to the file:

chmod +x generate\_report.py

Now test the function by running the file using the following command:

./generate\_report.py

The list employees\_list within the script should return the list of dictionaries as shown below.



**Process employee data**

The second function *process\_data()* should now receive the list of dictionaries, i.e., employee\_list as a parameter and return a dictionary of **department:amount**.

Open the file generate\_report.py to define the function.

nano generate\_report.py

def process\_data(employee\_list):

This function needs to pass the employee\_list, received from the previous section, as a parameter to the function.

Now, initialize a new list called department\_list, iterate over employee\_list, and add only the departments into the **department\_list**.

department\_list = []

for employee\_data in employee\_list:

department\_list.append(employee\_data['Department'])

The department\_list should now have a redundant list of all the department names. We now have to remove the redundancy and return a dictionary. We will return this dicationary in the format **department:amount**, where amount is the number of employees in that particular department.

department\_data = {}

for department\_name in set(department\_list):

department\_data[department\_name] = department\_list.count(department\_name)

return department\_data

This uses the set() method, which converts iterable elements to distinct elements.

Now, call this function by passing the employee\_list from the previous section. Then, save the output in a variable called dictionary. Print the variable dictionary.

dictionary = process\_data(employee\_list)

print(dictionary)

Save the file by clicking Ctrl-o, Enter, and Ctrl-x.

Now test the function by running the file using the following command:

./generate\_report.py

This should return a dictionary in the format **department: amount**, as shown below.



**Generate a report**

Next, we will write the function *write\_report*. This function writes a dictionary of **department: amount** to a file.

The report should have the format:

<department1>: <amount1>

<department2>: <amount2>

Lets open **generate\_report.py** file to define the function.

nano generate\_report.py

def write\_report(dictionary, report\_file):

This function requires a dictionary, from the previous section, and report\_file, an output file to generate report, to both be passed as parameters.

You will use the open() function to open a file and return a corresponding file object. This function requires file path and file mode to be passed as parameters. The file mode is 'r' (reading) by default, so you should now explicitly pass 'w+' mode (open for reading and writing, overwriting a file) as a parameter.

Once you open the file for writing, iterate through the dictionary and use write() on the file to store the data.

with open(report\_file, "w+") as f:

for k in sorted(dictionary):

f.write(str(k)+':'+str(dictionary[k])+'\n')

f.close()

Now call the function *write\_report()* by passing a dictionary variable from the previous section and also passing a report\_file. The report\_file passed within this function should be similar to */home/<username>/data/report.txt*. Replace <username> with the one mentioned in Connection Details Panel at left-hand side.

write\_report(dictionary, '<report\_file>')

Save the file by clicking Ctrl-o, Enter, and Ctrl-x.

Let's execute the script now.

./generate\_report.py

This script does not generate any output, but it creates a new file named **report.txt** within the **data** directory. This report.txt file should now have the count of people in each department.

Navigate to the data directory and list the files. You should see a new file named **report.txt**.

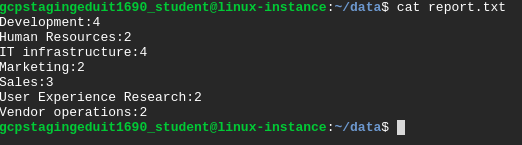
cd ~/data

ls

To view the generated report file, use the following command:

cat report.txt

The report file should be similar to the below image.



**Congratulations!**

You successfully wrote a Python script that achieves two tasks. First, it reads a CSV file containing a list of the employees in the organization. Second, it generates a report of the number of people in each department in a plain text file.

Creating reports using Python is a very useful tool in IT support. You will likely complete similar tasks regularly throughout your career, so feel free to go through this lab more than once. Remember, practice makes perfect.