Lab 4: Analyzing customer data

Parts of this lab are adapted from
http://www.cs.uni.edu/~diesburg/courses/cs1510_fa14/labs/lab13/index.htm by Sarah Diesburg with a few sentences and paragraphs copied verbatim from these sources.

Objectives

- Learn to access files in a program.
- Explore the types: lists and dictionaries.
- Analyze data from a data file.

Introduction

We will first learn how to open files and read information from them in a program. Then we will explore two new types: lists and dictionaries. Finally we will analyze fake customer data for an imaginary company.

As you work through this lab, you will read parts of sections 4.6, 5.2, 5.4 and 5.5 of "Introduction to Computation and Programming Using Python" by Guttag.

Part A: Accessing Files

We will see how Python helps us access information stored in other files. It is also possible to create files in Python but we will not do much of that. Read section 4.6 on pages 53-55 of Guttag's book. Now let us try to understand the `load_words` function from the `wordgame.py` file that we used in lab last week to write the word guessing game.
Download the file `createwordlist.py` from Lyceum. The file has the `load_words` function from last week. Save the file and run the program. You should get the following output.

```
Loading word list from file…
   55900 words loaded.
```

Now let us try to understand each line of the code.

Questions for discussion

What is the type of the objects associated with each of the following variables in the function: `infile, line, wordlist`?

Which file is associated with the variable `infile`?

Change the code `line=infile.read()` to `line = infile.readline()`. Does the function still give the same output as before? Why or why not? What is the type of the object associated with the variable `line` now?

Change the code to `line=inFile.readlines()`. Does the function still give the same output as before? Why or why not? What is the type of the object associated with the variable `line` now?

Now change the code back to the original `line=infile.read()` .

Exercise: The function `word_list` works with only one file – `words.txt` . Write a new function `load_words_modified` that accepts a text file as an argument and returns a list of words in the file. You do not need to start from scratch – just modify the `word_list` function.

Show the solution to the Exercise to the instructor.

Part B: Lists

Read section 5.2 and subsection 5.2.1 on pages 58-63 from Guttag's book.

Exercise: Add lines of code to the function `load_words` to do the following things:

Print the number of words in the file `words.txt` that start with the letter a.

Create a new list of words called `new_wordlist` that has the same words as the list `wordlist` except for the words that begin with the letter `a` or `A`. Make sure that you do not change the list `wordlist`. Print the length of `wordlist` and `new_wordlist`. Remember that lists are mutable objects. So you need to create `new_wordlist` carefully.

Show the solution to the Exercise to the instructor.

Python has a built-in type called `tuple` which is similar to the type `list`. Objects of type `tuple` are enclosed within parentheses instead of square brackets. For example, `(1, 'one', 2, 'two')` is an object of type `tuple` while `[1, 'one', 2, 'two']` is an object of type `list`. The big difference between the two types is that tuples are immutable while lists are mutable.

Read section 5.4 on pages 66-67 of Guttag's book to understand the similarities and differences between the types: `str`, `tuple` and `list`.

Part C: Dictionaries

Unlike most programming languages, Python has a built-in type called `dict` which allows for objects similar to lists but which can be indexed by immutable objects other than integers. Objects in an object of type list are indexed by integers. For example, the "indices" of the list

`L=[1, 'one', 2, 'two']` are the integers 0, 1, 2, and 3 and so we can refer to `L[0]`, `L[1]`, etc. In an object of type dict, the "indices" do not have to be integers.

Read section 5.5 on pages 67-69 of Guttag's book. You can skip the details of the example in Figure 5.9 in your reading.

Part D Analyzing Customer Data

We will now analyze data from a file of 30,000 fake customers of an imaginary company. The data is stored in a text file called `FakeCustomerData.txt` which as the name implies has randomly generated data that does not constitute any real information.

Download the file `FakeCustomerData.txt` from Lyceum and take a look at it. The first line of the file has the header information for each of the columns.

Open a new file in Python and call it `customerdata.py` . Write the functions for problems 1 and 2 below in this file. Save this file and the file `FakeCustomerData.txt` in the same folder.

Problem 1: Analyze data by state

In order to understand where the customers come from we want to know how many customers come from each state.

Write a function called `state_distribution()` which uses a dictionary to obtain information about the number of customers from each state. This function works on our one customer data file and so it accepts no arguments. The function prints a list of the names of the states sorted alphabetically along with the number of customers from that state.

Below is a sample run of the function which includes only a partial list from the output.

```
>>> state_distribution()
AK, 120
AL, 462
AR, 363
AZ, 453
CA, 3130
...
TX, 2322
UT, 251
VA, 690
VT, 126
WA, 634
WI, 643
WV, 177
WY, 75
```

Problem 2: Analyze data by birth year

Now we want to understand the distribution of customers by their age.

Write a function called `birth_year_distribution()` which uses a dictionary to obtain information about the number of customers for each birth year. This function works on our one customer data file and so it accepts no arguments. The function prints a sorted list of the birth years along with the number of customers with that year as the birth year.

Below is a sample run of the function which includes only a partial list from the output.

```
>>> birth_year_distribution()
1940, 651
1941, 677
1942, 660
1943, 663
...
1980, 637
1981, 638
1982, 693
1983, 722
1984, 660
1985, 622
```

Lab Submission

Email me the Python file you created in Part D by 2 p.m. on Friday, May 15. Rename the file name so that it contains the last names of each of the partners and the word `customerdata`. For example, if I am working with someone whose last name is Byron, our filename would be `byron_jayawant_customerdata.py`.

This submission will be graded out of ten points. Grading is based on correctness and completeness of the two functions you have written in part D and appropriate use of the coding standard.

Optional Additions

Consider implementing the following extensions if you have time and want to do it. This will not affect the grade in any way. Do not submit the extensions with your current submission. You may use these extensions in your week 5 work.

1. The function in Problem 1 was written to work with one specific data file, `FakeCustomerData.txt` and with one specific column, the column with the states in it. But what if we want to analyze the data from any column? Or from any data file? We need a more general function for that purpose that accepts two arguments – a filename and a column number and then prints the appropriate information pertaining to that column.

If the user gives a column number that doesn't exist in the data file, the function should print a statement explaining that no information is available for that column number.

A few sample runs of the function may look as follows.

```
>>> column_distribution("FakeCustomerData.txt", 20)
There is no column number 20
>>> column_distribution("FakeCustomerData.txt", 14)
MasterCard -> 14868
Visa -> 15132
```

2. Sometimes we might know the header name but not the column number for the column whose data we want to analyze. Revise the function from the previous extension. It should continue to work exactly as it did before. That is, it should work when the user passes in a string for a filename and an integer for the column number. However, in addition, it should also work if a string is input as a column header for the second argument. In this case, the function will look at all of the headers in the data file to see if the input string is one of the headers. If the column name exists, the function organizes the data from that column. If it does not exist, then the function prints an appropriate message.

A few sample runs of the function may look as follows.

```
>>> column_distribution("FakeCustomerData.txt", "State")
AK -> 120
AL -> 462
AR -> 363
AZ -> 453
>>> column_distribution("FakeCustomerData.txt", "CC")
There is no column labeled CC
>>> column_distribution("FakeCustomerData.txt", 5.2)
There is no column number 5.2.
>>> column_distribution("FakeCustomerData.txt", 20)
There is no column number 20
>>> column_distribution("FakeCustomerData.txt", 14)
MasterCard -> 14868
Visa -> 15132
```