CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #1

due at 5pm, Tue 31 Aug 2021

Precision
Follow the directions given *exactly:* your code will be graded automatically, so any deviation from the program spec can result in a significant loss of credit. If you are unsure about something, ask for clarification in Office Hours or Piazza.

Style
Please pay attention to the programming style guidelines for this class. For this assignment you will be notified of style violations but not penalized for them; style violations will be penalized in subsequent assignments.

Submitting Your Solution
This assignment requires that you submit multiple files. Because of the the way that the auto-grader script works, in order to get full credit for your work you have to submit **all** of these files each time that you want to resubmit a solution to either problem.

1 Overview

In this assignment, you will be working with a lot of basic Python syntax, structures, and types. You will write a series of Python programs, each one in its own file.

Each program must have a `main()` function, and you must not have any of your code (except for imports and comments) outside of functions. You **may** use additional functions if you wish.

Pay close attention to what this spec asks you to do: if you mix things up, you won't get credit for your code.

2 `strings_and_input.py`

Write a program, in a file named `strings_and_input.py` , which does the following:

Your program will read an input string from the keyboard. You will print a prompt, and save the value which the user types. To do this, you must call the following Python function, and save its return value into a variable:

```
input("input string: ")
```

After you read this input, print out the following information, one per line:

- The length of the string that `input()` returned.

- The second character in the string (you may assume that the string has at least two characters)
- The first 10 characters in the string - or less, if the string is shorter than that. (Use slicing, so that you don't have to worry about the string length)
- The last 5 characters in the string (again, use slicing)
- The entire string, with lowercase characters converted to uppercase
- Inspect the first character in the string, and classify it.

  If the first character is any one of the following characters (uppercase or lowercase): q,w,e,r,t,y, print `"QWERTY"` .

  If the first character is any one of the following characters (lowercase only): u,i,o,p, print `"UIOP"` .

  If the first character is any letter (other than the previous two cases), print `"LETTER"` .

  If the first character is a digit, print `"DIGIT"` .

  If it is anything else, print `"OTHER"` .

  HINTS: To make the tasks above easier, think about how the `in` operator works for strings. Also, read up on the `upper()`, `lower()`, and `is...()` methods of the `str` class. See also the python documentation here: `https://docs.python.org/3.9/library/stdtypes.html?highlight=str#str`

After you have printed out all of the information about the string in the list above, read two numbers (saving both into variables), but **without any prompts**. You can read them by calling the following function **twice:**

```
input()
```

Convert each number to an integer, multiply them together, and print the result. (You may assume that both inputs are integers, don't worry about writing any error handling code.)

HINT: Use `int()` to convert a string into an integer.

(the spec continues on the next page)

3 `sequence_len.py`

Write a program, in a file named `sequence_len.py`, which reads a series of integers. There might be multiple integers in a single line of input; if so, they will be separated by whitespace. Some lines of input might be blank.

The input intgers will be in some sort of order: maybe ascending, maybe descending (you won't be told which). Count how many integers are in order; stop counting when one of them is out of order. Print out how many integers there were that were in order.

You may assume that all input is integers; there will not be any non-numeric words. Also, you may assume that the sequence will get broken **before** you hit the end of the input; thus, you don't have to worry about the input getting exhausted (something known as `End of File`).

Remember: the sequence might include duplicates; that still counts as an ascending or descending sequence.

EXAMPLE:
If the input is this:

```
2 3 5 7 11 10
```

print out `5` because the first 5 numbers were in order, but the sixth was not.

EXAMPLE:
If the input is this:

```
-1 -1
-1
2 3
4 -5 6
7 8 9
```

print out `6` because the value `-5` was out of order.

(the spec continues on the next page)

EXAMPLE:
If the input is this:

```
    10
    10
    10
    10
         9
 8 8 7
      6     5
  4 3
3
3
3
              10
             9
            8
           7
```

print out `15` because the value `10` (after `3`) is out of order; until then, the sequence was descending. Also note, from this example, that whitespace in the input is ignored.

4 swap.py

Write a program, in a file named `swap.py` , which does the following:

Your program will read an input string from the keyboard. You will print a prompt (**different than the previous problem!**) and read an input string. The prompt, for this problem, must be:

```
Please give a string to swap:
```

You must remove all whitespace from the beginning and end of the input string. Do **not** remove any whitespace from the middle! (Remember, Python allows you to do this with a single function call. Review the documentation.)

After you have removed the whitespace, swap the front and back halves of the string, and print out the result. If the length of the string (after removing the whitespace) is even, then this is easy; just split it in half. If the length of the string is odd, then keep the middle character **in the middle of the result.** For example, if the input string is `"abcde"` , then the output should be `"decab"` .

(the spec continues on the next page)

5 population.py

Write a program, in a file named `population.py` , which does the following:

Your program will read an input string from the keyboard. You will print a prompt (`"file: "`) and then read an input string. Remove all leading and trailing whitespace from the input string, and then open a file with that name. You will then read through that file, reading the lines one at a time, and performing a calculation on the data. You **must** use a `for` loop to read the lines of the file; you may either directly use the `for` loop on the file, or read the lines into a list, and use a `for` loop over that.

Your program should ignore any blank lines within the input (including lines that have some whitespace; treat a line as blank if it contains **only** whitespace).

Likewise, your program should ignore any line within the input where the first non-whitespace character is `'#'` (so that we can place comments in the input file).

REMEMBER: `open(filename)` will open a file, and return a file object. You can call `readlines()` on the object to get the lines of the file as a list.

5.1 Input File Format

The input file is a text file where each line (except blank lines and comments, see above) consists of the name of a US state or territory, together with its estimated 2019 population. For example:

```
Oklahoma          3943079
Oregon          4190713
South Dakota          882235
Pennsylvania          12807060
Northern Mariana Islands  55194
```

To simplify your input processing, you may assume that, between the words of the placename, we will only use single spaces. However, note that there might be any type of whitespace - and any number of whitespace characters - between the placename and the population number.

5.2 The Calculation

Read each line of the input file. For each line (other than the blank or comment lines), print out the information on two lines, as follows:

```
State/Territory: Oklahoma
Population:      3943079
    <blank line>
State/Territory: Oregon
Population:      4190713
```

and so on.

At the very end of the output, print a blank line, followed by a summary of the information:

```
# of States/Territories: xxxxx
Total Population:        xxxxx
```

5.3 Mu Only: Getting the Right Directory

In many systems (including our autograder), it's very common to put the program, and its input files, into the same directory. Then, when you run your program, the operating system will set the "current working directory" of the program to point to the directory that contains the program. This is nice, since when you open a file, for instance

```
some_file = open("data.txt")
```

then the OS will look for `data.txt` in the same directory as the program.

But Mu doesn't work that way. It always uses the same directory as the "current working directory" - no matter where your program is. So I suggest, **for programs in this class only,** that you add the following code to the top of your program - to change the CWD.

First, you must import the `os` librrary. To do this, place the following line near the top of your file (just below the file docstring):

```
import os
```

Then, inside your `main()` function, add the following lines:

```
# chdir to the same directory as where this script is ... so
# that open() will open the file we expect.
this_script   = os.path.realpath(__file__)
dir_of_script = os.path.dirname(this_script)
os.chdir(dir_of_script)
```

I'm sorry for this trouble! But it's just a part of using Mu. :(

(the spec continues on the next page)

6 count_items.py

Write a program, in a file named `count_items.py` , which does the following:

As with the population program, use `input()` to read a filename from the keyboard; open the file, and read its contents. (Use the prompt

```
File to scan:
```

for this input.)

The input format is similar to the Population problem you just did; however, in this case, you can assume that the string will only have only one word (it will never have spaces in it). So make sure to remove leading and trailing whitespace, skip over comments, and ignore blank lines. But after you've handled all of this, each line will be

```
string integer
```

Your job will be to print out the totals, ordered first by the count, and (if there are duplicate counts), ordered by the word.

EXAMPLE
If your input looks like this:

```
asdf  10
jkl   3
asdf  -1
foobar  17
bbbb    17
```

then the **last step** of your output should be:

```
jkl 3
asdf 9
bbbb 17
foobar 17
```

(the spec continues on the next page)

6.1 The Required Steps

A few of you may already have some ideas about how to get this done - but many students will find it challenging. So I've provided step-by-step instructions for this program. And you will be required to print out some debugging information, for each step, to show that you are following along.

6.1.1 Step 1: The Dictionary

Scan through the file. Build a dictionary where the words are keys, and the values are integers - the sum total of all of the values that we've seen for that particular word.

Unfortunately, since the order in which dictionaries print out can be kind of hard to predict, you can't print out the dictionary directly. Instead, you will extract all of the keys from the dictionary, sort them, and then print out the key and value, one per line. In the example above, your output will be:

```
STEP 1: THE ORIGINAL DICTIONARY
  Key: asdf Value: 9
  Key: bbbb Value: 17
  Key: foobar Value: 17
  Key: jkl Value: 3
```

6.1.2 Step 2: List of Value,Key Pairs

Iterate through the dictionary, again doing it in sorted-key order. But this time, for every key, build a tuple

```
(value, key)
```

Add each new tuple that you build to an array. When you're done, print it out, and it will look like this:

```
STEP 2: A LIST OF VALUE->KEY TUPLES
[(9, 'asdf'), (17, 'bbbb'), (17, 'foobar'), (3, 'jkl')]
```

6.1.3 Step 3: Sorting

Sort the list, and print out the result. Did you notice that, because the values come before the keys in each tuple, the tuples will be sorted first by the values inside them? And that if there are any duplicate values, then they will be sorted by the keys?

```
STEP 3: AFTER SORTING
[(3, 'jkl'), (9, 'asdf'), (17, 'bbbb'), (17, 'foobar')]
```

6.1.4 Step 4: The Actual Output

Finally, iterate through the sorted list, and print out the values in the format we actually wanted.

```
STEP 4: THE ACTUAL OUTPUT
jkl 3
asdf 9
bbbb 17
foobar 17
```

6.2 Summary

In summary, if the input to your program is:

```
asdf  10
jkl   3
asdf  -1
foobar  17
bbbb    17
```

then the output should look like this:

```
STEP 1: THE ORIGINAL DICTIONARY
  Key: asdf Value: 9
  Key: bbbb Value: 17
  Key: foobar Value: 17
  Key: jkl Value: 3

STEP 2: A LIST OF VALUE->KEY TUPLES
[(9, 'asdf'), (17, 'bbbb'), (17, 'foobar'), (3, 'jkl')]

STEP 3: AFTER SORTING
[(3, 'jkl'), (9, 'asdf'), (17, 'bbbb'), (17, 'foobar')]

STEP 4: THE ACTUAL OUTPUT
jkl 3
asdf 9
bbbb 17
foobar 17
```

7 Turning in Your Solution

You must turn in your code using GradeScope.

8 Acknowledgements

Thanks to Saumya Debray for many resources that I used and adapted for this class.