

CS 120 (Spring 22): Introduction to Computer Programming II

Long Project #1

due at 5pm, Tue 25 Jan 2022

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 running_total.py

Write a program, in a file named `running_total.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.

Break out of the loop, and terminate the program, if you ever read the value -1. Do not print out the line for the -1. (Note: The -1 might be in the middle of a line; if so, don't process the numbers on the rest of the line.)

I recommend that you structure program as follows:

```
keep_going = True
while keep_going:
    ... read one line of input ...
    for each word in this one line of input:
        ... convert the word to an integer ...
        if it is time to end the program:
            keep_going = False
            break
    else:
        ... update and print out current status ...
```

Each time that you read a new integer, you will print out one line of input. The primary purpose is to keep track of a running total of all of the integers; however, you will also print out other information about the loop.

Each line will look like this:

```
Total:1234 - numbers:10 total_lines:4 blank_lines:1
```

SPECIAL RESTRICTION: (this program only)

For this program only, you must use f-strings to format your output. Not familiar with them? Watch the Python Review videos that I've posted! On other programs, you have complete freedom - but I think that you will love f-strings once you've tried them.

BANNED FUNCTIONS: (throughout this course)

Some resources online will tell you that, to terminate a Python program, you can call `os._exit()` or `sys.exit()`. While it is technically true that you can do this, it is **ugly and error-prone**, and so it is **banned** in this course. To end a program, please work out how to break out **cleanly** from your loops. Don't just kill the program!

(the spec continues on the next page)

4 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.

4.1 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.

(the spec continues on the next page)

4.2 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
```

4.3 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` library. 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. :(

5 count_items.py

WARNING: Read this entire section (including the required debugging information) before you start writing your code. There are a number of ways to solve this problem, but since I'm requiring you to print out debugging information, your options will be constrained.

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)

5.1 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.

I apologize if you had another idea for how to solve this problem - you will need to follow along with my recommended algorithm.

5.1.1 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
```

5.1.2 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')]
```

5.1.3 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')]
```

5.1.4 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
```

5.2 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
```

6 Turning in Your Solution

You must turn in your code using GradeScope.

7 Acknowledgements

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