Due 11:59:00pm Friday April 15

Last chance to use tokens! (P6 won't allow late submissions)

- The purpose of this assignment is to explore dictionaries and file reading and writing. We will be reading / creating some dictionaries about presidents and unemployment, and then checking for some statistics in various ways.
- You will turn in a single python file following our naming convention: `gmason76_2xx_P5.py`
- Include the following in comments at the top of your file: Name, G#, lecture and lab sections, and any references or comments we ought to know. You don't have to use the template from previous projects.
- Use the Piazza discussion forums (and professor/TA office hours) to obtain assistance.
- Remember, do not publicly post code for assignments on the forum! Use private posts as needed (or when you're not sure). Have a specific question ready, and both show what you're thinking and show what you've tried independently before you got stuck. We will prod for more details if needed.

Background:

Dictionaries give us an enriched way to store values by much more than just sequential indexes (as lists gave us); we identify key-value pairs, and treat keys like indexes of various other types. Though the keys are unordered, dictionaries help us simplify many tasks by keeping those key-value associations. Each key can only be paired with one value at a time in a dictionary.

When a file contains ASCII text in it, we can readily write programs to open the file and compute with its contents. It turns out that reading and writing text files gives our programs far more longevity than open-to-quit; we can store data and results for later, save user preferences, and all sorts of things. We will be reading text files that happen to be in the CSV format.

What's allowed?

Here is the exhaustive list of things you can use on the project.

- all basic expressions and statements
- functions: `len(), range(), min(), max(), enumerate(), int(), list(), tuple(),dict(),` and any functions you write
- file reading: `open(), close(), read(), readline(), readlines(), with` syntax
- dictionaries: `len, get, clear, copy, keys, values, items, pop, popitem, update`
- methods: lists: `remove(), insert(), append(), extend(), pop(), index()`
  strings: `split(), endswith(), startswith(), join(), replace(), insert()`
- `sorted(), sort(), reversed(), reverse()`
- **all sets and set operations; all exception handling blocks/raises.**

This means that…

- you can't call anything not listed above. Focus on applying these functions to solve the task.
- you can't import any modules for this project. (so you can't `import csv` either – it isn't that helpful)

Procedure

You will complete the function definitions as described; you can test your code with this sample data and testing file:

- testing files(tester5p.py and some sample csv files): http://cs.gmu.edu/~yzhong/112/p5testfiles.zip (to be posted)
    - leave the .csv files in the same directory as the tester and your code.
    - Invoke it as with prior assignments: `python3 tester5p.py yourcode.py just these funcs`

Scenario

We define two kinds of dictionaries in this project. Many functions deal with just one dictionary but there are some that interact with both kinds of dictionaries. Make sure you get comfortable with individual dictionaries before moving forward to those that play with two.

`President database`: a "database" of presidents defined as a dictionary. Names of presidents are used as keys while the values of the dictionary are tuples with basic presidential information in this order: year of inauguration, years in office, age at inauguration, state elected from, and political party. Note that state and party of a president are represented as strings while the other three are represented as integers.

```
sample_president_tuple = (year_inauguration, years_in_office, age, state, party)
sample_p_db = {
                "Ronald Reagan": (1981, 8, 69,"California","Republican"),
                "George Bush": (1989, 4, 64,"Texas","Republican"),
                "Bill Clinton": (1993, 8, 46,"Arkansas","Democrat")       }
```

`CSV file`: We've got some president data in a comma-separated-values file; only `read_file` needs to interact with files and build the initial database. All other functions will be given a database to work with. This is a file containing ascii text where each line in the file represents one record of information; each piece of info is surrounded by double quotes, and each of these quoted things is separated by a single comma. The very first line is the "header" row, which names the columns but is not part of the data. Here is a very small sample file that can is used to build the sample president database above. Note: a file's extension has no actual effect on its contents. These are ascii files, so you can edit them with your code editor just as easily as a .txt or .py file.
We recommend not using Excel to open the files! It saves in a slightly different format, causing trouble.

```
"President","Year inaugurated","Years in office","Age","State","Political Party"
"Ronald Reagan","1981","8","69","California","Republican"
"George Bush","1989","4","64","Texas","Republican"
"Bill Clinton","1993","8","46","Arkansas","Democrat"
```

`Unemployment Database`: a second "database" includes the unemployed data for multiple years. This database is a dictionary whose keys are years (as integers), and whose values are tuples of 12 integers, each representing one month's unemployment number in thousands. You will not need to build up any unemployment database from a file.

```
      sample_unemployment_db = {
1989: (6682, 6359, 6205, 6468, 6375, 6577, 6495, 6511, 6590, 6630, 6725, 6667),
1990: (6752, 6651, 6598, 6797, 6742, 6590, 6922, 7188, 7368, 7459, 7764, 7901),
1991: (8015, 8265, 8586, 8439, 8736, 8692, 8586, 8666, 8722, 8842, 8931, 9198),
1992: (9283, 9454, 9460, 9415, 9744, 10040, 9850, 9787, 9781, 9398, 9565, 9557) }
```

Functions dealing with president databases only

These functions work on, or create, president databases.

- `read_file(filename)`: This is the *only* function that needs to deal with reading a file. It will accept the file name as a string, and assumes it is a `CSV file` as described above (with our president data in the same format as the example, but with any number of rows after the header row). It will open the file, read all the name entries, and correctly create the `president database`.
  - Return the created dictionary.
  - **Hint:** How can you break this task down into multiple phases, each one taking a pass over the data and making something slightly more useful towards getting the result? How can you use any functions that you have to write, or could write? String methods could be helpful.
  - **if you get stuck on this one**, you can still attempt others – just test with the tester or with manually built database values.

- `youngest_at_inauguration(p_db)`: This function accepts an existing president database `p_db`, finds presidents with the youngest age at inauguration, and returns a tuple as `(age, list_of_names)`. If there are multiple names (with a tie), they must be alphabetically sorted.
  - ```
    sample_p_db = {
                    "Ronald Reagan": (1981, 8, 69,"California","Republican"),
                    "George Bush": (1989, 4, 64,"Texas","Republican"),
                    "Bill Clinton": (1993, 8, 46,"Arkansas","Democrat")       }
    ```
  - `youngest_at_inauguraton(sample_p_db)` → `(46: ['Bill Clinton'])`

- `oldest_at_retirement(p_db)`: This function accepts an existing president database `p_db`, finds presidents that were oldest at retirement, and returns a tuple as `(age, list_of_names)`. If there are multiple names, again they must be alphabetically sorted.
  - `oldest_at_retirement (sample_p_db)` → `(77, ['Ronald Reagan'])`

- `presidents_by_state(p_db, state)`: This function accepts an existing president database `p_db` and a string `state` name. It searches for all presidents whose state matches the `state` argument, and builds/returns another database that only contains those matching presidents.
  - ```
    presidents_by_state(sample_p_db,"Texas") →
              {'George Bush': (1989, 4, 64, 'Texas', 'Republican')}
    ```

- `presidents_by_party(p_db, party)`: This function accepts an existing president database `p_db` and a string `party` name. It searches for all presidents whose political party matches the `party` argument, and builds/returns another database that only contains those matching presidents.
  - ```
    presidents_by_party(sample_p_db," Republican") →
        {'George Bush': (1989, 4, 64, 'Texas', 'Republican'),
         'Ronald Reagan': (1981, 8, 69, 'California', 'Republican')}
    ```

    ```

Functions dealing with unemployment databases only

- `total_unemployment_for_year(u_db, year)`: This function accepts an existing database `u_db` and an integer `year`. It calculates and returns the total unemployment number for that year.
  - Return `None` if the given `year` is not included in the database.
  - `sample_unemployment_db = {`
  `1989: (6682, 6359, 6205, 6468, 6375, 6577, 6495, 6511, 6590, 6630, 6725, 6667),`
  `1990: (6752, 6651, 6598, 6797, 6742, 6590, 6922, 7188, 7368, 7459, 7764, 7901),`
  `1991: (8015, 8265, 8586, 8439, 8736, 8692, 8586, 8666, 8722, 8842, 8931, 9198),`
  `1992: (9283, 9454, 9460, 9415, 9744, 10040, 9850, 9787, 9781, 9398, 9565, 9557) }`
  - `total_unemployment_for_year(sample_unemployment_db,1989)` → `78284`
  `#total of the 12 month data from the db`

- `avg_unemployment_for_month(u_db, month)`: This function accepts an existing database `u_db` and an integer month. It calculates and returns the average unemployment number for that month across all years in the database.
  - The argument `month` uses integer `0` to `11` to represent months from January to December.
  - Return `None` if the given `month` is negative or greater than `11`.
  - Only keep the integer division result for the average calculation (no fractional or floating point result).
  - `avg_unemployment_for_month(sample_unemployment_db,1)` → `7682`
  `# avg of Feb data for 1989-1992`

Functions dealing with both president and unemployment databases

- `avg_unemployment_for_president(p_db,u_db,president)`: This function accepts a president database `p_db`, an unemployment database `u_db`, and a string `president` name. It calculates and returns the average unemployment number across all months of the years the given `president` was in office.
  - We are ignoring the inauguration date of presidents and use all 12 months of the year in calculation if the given president is in office for that year according to our president database `p_db`.
  - Return `None` if the given `president` is not included in the president database `p_db`.
  - Return `None` if the any year the given `president` in office is not included in the unemployment database `u_db`.
  - Only keep the integer division result for the average calculation (no fractional or floating point result).
  - `avg_unemployment_for_president(sample_p_db, sample_unemployment_db," "George Bush")`
  → `7958 # avg of all monthly data for 1989-1992`

- `unemployment_change_for_president(p_db,u_db,president)` :This function accepts a president database `p_db`, an unemployment database `u_db`, and a string `president` name. It calculates and returns the change of unemployment number from the first month to the last month when the given `president` was in office. Note: a positive number means increased unemployment.
  - We are ignoring the inauguration date of presidents. Hence for any given president, the first month is always January of the starting year in office; and the last month is always December of the ending year in office.
  - Return `None` if the given `president` is not included in the president database `p_db`.
  - Return `None` if the starting year or ending year of the given `president` is not included in the unemployment database `u_db`.
  - `unemployment_change_for_president(sample_p_db, sample_unemployment_db,"George Bush")` → `2875 # unemployment number of Dec. 1992 - unemployment number of Jan. 1989`

- `president_lowest_avg_unemployment(p_db, u_db)`: This function accepts a president database `p_db` and an unemployment database `u_db`. It compares the average unemployment for all presidents and reports the one with lowest monthly average number.
  - You can assume that there is no tie and only one name needs to be returned for this function
  - **Hint:** use your `avg_unemployment_for_president` definition!
  - Check the tester for examples

- `president_lower_unemployment_most(p_db, u_db)`: This function accepts a president database `p_db` and an unemployment database `u_db`. It compares the average unemployment for all presidents and reports the one that reduced the unemployment the most.
  - You can assume that there is no tie and only one name needs to be returned for this function
  - **Hint:** use your `unemployment_change_for_president` definition!
  - Check the tester for examples

- `expand_database(p_db, u_db)`: This function accepts a president database `p_db` and an unemployment database `u_db`. It calculates average monthly unemployment number for every president and adds this info into the president database `p_db`. The dictionary `p_db` should be expanded by adding an average unemployment number into every value tuple.
  - If a president's average number cannot be calculated due to missing information from the unemployment database, use `None` as the average.
  - The function updates the president database `p_db` in place; it should return `None`.
  - **Hint:** use your `avg_unemployment_for_president` definition!
  - Sample run:

```
>>> expand_database(sample_p_db,sample_unemployment_db)
>>> sample_p_db
      {'Ronald Reagan': (1981, 8, 69, 'California', 'Republican', None), 'Bill Clinton':
      (1993, 8, 46, 'Arkansas', 'Democrat', None), 'George Bush': (1989, 4, 64, 'Texas',
      'Republican', 7958)}      # no data for Reagan or Clinton
```

Extra Credit

- `total_unemployment_by_party(p_db, ue_db)`: This function accepts a president database and an unemployment database. It calculates the total unemployment number for all presidents of a political party and returns a list of tuples `(party, total_unemployment)`. The list should include one tuple for every party in the president database `p_db`. It should be sorted alphabetically according to the party names.
  - If a president/year's total number cannot be calculated due to missing information from the unemployment database, treat it as zero.
  - `total_unemployment_by_party(sample_p_db, sample_unemployment_db)` →
    `[('Democrat', 0), ('Republican', 382028)]` `# no data for Reagan or Clinton`

Grading Rubric

```
Code passes shared tests:     80
Well-documented/submitted:    10
No globals used (just def's): 10
-------------------------------
TOTAL:                       100  +5 extra credit
```

Note

- You should always test your code not only with the provided testing script, but also by directly calling your functions. If you store sample message strings to variables after all the definitions, you can use them in these interactive calls. ***Just be sure to remove them before turning in your work – they are globals, which are not allowed in your final submission.*** Consider the file below on the left, named **`shouter.py`**, which you can run as shown below on the right using interactive mode (-i).

```
def shout(msg):
     print(msg.upper())

mystring1 = "hello"
mystring2 = "another one"
```

```
demo$ python3 -i shouter.py

>>> shout(mystring1)
'HELLO'
>>>
```

Reminders on Turning In Projects

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available, just turn in your work (perhaps again) between 0 and 48 hours late to use them.

You can turn in your code as many times as you want; we will only grade the last submission that is <=48 hours late. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment, you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least for part of the program.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

Use a backup service. Do future you an enormous favor, and just keep all of your code in some automatically synced location, such as a Dropbox or Google Drive folder. Every semester someone's computer is lost/drowned/dead, or their USB drive is lost/destroyed, or their hard drive busts. Don't give these situations the chance to doom your project work!