Exercise: Using the Census API

Summary

This exercise uses the Census API to retrieve data on educational attainment by county in New York State. It then examines that data briefly. A subsequent exercise will use GIS to map the results.

Input Data

Most of the data will be obtained from the Census via its API. However, one CSV file, **variable-info.csv**, is provided that will be used to help manage the data retrieval and analysis. The file has three columns. The first, `variable`, contains the names of 25 Census variables, B15003_001E through B15003_025E, which give the number of people aged 25 and older whose final level of education falls in various categories. Be sure to keep in mind that it's the *final* year of education. For example, B15003_016E is the number of people whose final year of education was graduating from high school: it does not include anyone who finished a year of college or more.

The second column, `description`, has a short Census description for each variable. The third column, `group`, does not come from the Census. Rather, it was added as part of this exercise to facilitate aggregating the data. It takes on one of the following values, depending on the variable described in the row: `'total'`, for the total population 25 and over in the county; `'<hs'` for any level of education below high school graduation; `'hs'` for high school graduation or equivalent; `'some col'` for some college or an Associate's degree; `'ba+'` for a Bachelor's degree or beyond.

Deliverables

The deliverables are two scripts, **collect.py** and **analyze.py**, and a short Markdown file called **results.md**. Your CSV and PNG files will be uploaded as well when you push your repository but they'll be rebuilt when I run your scripts so they're not technically deliverables.

Instructions

A. Sign up for a Census API Key

1. Go to https://api.census.gov/data/key_signup.html and request an API key. Strictly speaking, the Census only requires a key if you might make a large number of requests per day. However, it's free and a professional courtesy to the Census to use one, so please sign up.

B. Script collect.py

1. Import `pandas` and `requests`.
2. Set `var_info` to the result of using `pd.read_csv()` to read `'variable-info.csv'`.
3. Set `var_name` to the `'variable'` column of `var_info` with the `.to_list()` method applied to the end. The `to_list()` method converts the Pandas Series into a simple list.
4. Set `var_list` to `['NAME']+var_name`. Adding `'NAME'` to the list of variables tells the API to return the official Census name of the geographic entity in each row of the results.
5. Set `var_string` to the result of using `join()` to concatenate the elements of `var_list` into a single, comma-separated string. This will be the variable list part of the API call.
6. Set variable `api` to the American Community Survey 5-Year API endpoint for 2018 as shown below:

   ```
   api = 'https://api.census.gov/data/2018/acs/acs5'
   ```

7. Set `for_clause` to `'county:*'`. The left side of the "for" clause in a Census query indicates what kind of geographic unit should be returned and the right side is used to select subsets of the possible records.

The asterisk here is a wildcard indicating that data for all eligible counties, subject to the "in" clause below, should be returned.

8. Set `in_clause` to `'state:36'`. The "in" clause in a Census query is used for limiting the selected geographic units to those that fall within a larger geographic entity. In this case, the clause indicates that only counties in state 36, which is New York, should be returned.
9. Set `key_value` to your Census API key in quotes.
10. Set `payload` to a new dictionary with the following keys and values: `'get':var_string`, `'for':for_clause`, `'in':in_clause`, and `'key':key_value`. If there's a delay in delivery of your API key and you don't have it yet, leave the `'key':key_value` piece out of the dictionary until it arrives. The API call will work without it.
11. Set `response` to the value of calling `requests.get()` with arguments `api` and `payload`. The call will build an HTTPS query string, send it to the API endpoint, and collect the response.
12. Use an `if` statement to test whether `response.status_code` is equal to 200, the HTTP status code for success. If so, print a message indicating that the request succeeded.
13. Add an `else` statement. Within the `else` block print `response.status_code` and then print `response.text`, which may provide a little more detail about what went wrong. See the Tips section below for more information about errors.
14. Still within the `else` block add the following statement:

    ```
    assert False
    ```

    This will cause the script to stop immediately if the statement is reached. Assert statements are a very useful tool for writing reliable scripts; see the Tips section for more information.
15. After the end of the `else` block, set `row_list` to the result of calling the `.json()` method of `response`. That will parse the JSON returned by the Census server and return a list of rows. Be sure to note that the call is a method build into `response`: you do not need to import the JSON module.
16. Set `colnames` to the first row of `row_list` via `row_list[0]`.
17. Set `datarows` to the remaining rows via `row_list[1:]`.
18. Convert the data into a Pandas dataframe by setting `attain` equal to the result of calling `pd.DataFrame()` with arguments `columns=colnames` and `data=datarows`.
19. Set the index of `attain` to the `"NAME"` column.
20. Write out `attain` to `'census-data.csv'` using `.to_csv()`. Be sure to look at the file to make sure it looks OK. It should have a list of county names in the first column and a bunch of additional columns with the Census variables.

C. Script analyze.py

1. Import modules as needed.
2. Increase the default resolution for plots by setting `plt.rcParams['figure.dpi']` to 300.
3. Call `sns.set_theme()` with argument `style="white"` to set the default style for Seaborn.
4. Use the line below to avoid having Pandas suppress rows when printing out data objects:

    ```
    pd.set_option('display.max_rows',None)
    ```

5. Set `var_info` to the result of using `pd.read_csv()` to read `'variable-info.csv'`. Unlike the first time, however, include an additional argument to set the index to the variable name: `index_col='variable'`. This will be convenient later when we aggregate the variables.

6. Set `var_group` to the `'group'` column of `var_info`. This will be a handy link between the names of the Census variables, which are the index of the series, and the aggregate educational attainment groups, which are the values of the series.
7. Print `var_group`. It's a Pandas series and it will be helpful to see it when you're setting up the `groupby()` call below.
8. Set `attain` to the result of using `pd.read_csv()` to read `'census-data.csv'`. Use the argument `index_col='NAME'` to set the index to the column of county names.
9. Set `group_by_level` to the result of applying the `.groupby()` method to `attain` using the arguments `var_group`, `axis='columns'`, and `sort=False`. The `axis='columns'` argument indicates that what's to be combined are columns. The first argument, `var_group`, will be used used like a dictionary to do the grouping: `groupby()` will look up each column of `attain` in the index of `var_group` and will then use the corresponding value of `var_group` (e.g., `"hs"`) as the column's group. The `sort` argument keeps the new groups in the original order rather than sorting them alphabetically. That's handy because it keeps the columns in increasing order of educational attainment, which is more intuitive than an alphabetical list.
10. Set `by_level` to the result of applying the `.sum()` method to `group_by_level`. That does the aggregation.
11. Set variable `total` equal to `by_level` column `"total"`. That makes a convenient series for use below.
12. Now drop `total` from `by_level`.
13. Next, check for problems by setting `error` to the result of applying the `.sum()` method to `by_level` with the argument `axis="columns"` and then subtracting `total`.
14. Print `error`. If all has gone well, it should be 0 for each county.
15. Compute the percent of the population in each aggregate group by setting `pct` to 100 times the result of calling `by_level.div()` with the arguments `total` (the series built above) and `axis='index'`.
16. Print `pct['<hs'].sort_values()`, the share of the population with less than a high school education sorted from lowest to highest.
17. Now set variable `lo` to the sum of the percentages in the two lowest educational levels: `pct['<hs'] + pct['hs']`.
18. Set variable `hi` to `pct['ba+']`.
19. Compare the ratio of the fraction of highly educated people in the county to the fraction with the least eduction by setting variable `ratio` equal to `hi/lo` rounded to 2 decimal places. A result of, say 1.2, would indicate that there are 1.2 highly educated people for each person with a high school education or below.
20. Print some appropriate text and then print the value of `ratio` for `'Onondaga County, New York'`.
21. Print some appropriate text and then print the full set of ratios sorted from lowest to highest via `ratio.sort_values()`.
22. Now add a new column to `pct` called `"quint"` that is equal to the result of calling the `pd.qcut()` function with arguments `total/1e6` and `5`. As before, `pd.qcut()` divides its argument into quantiles and then returns a series of numbers indicating the quantile of each row. Dividing the total by `1e6` converts the populations into millions, which doesn't change the quantiles but does improve the legend of a graph that will be drawn next.
23. Generate a matrix of scatter plots comparing the educational attainment levels by quintile by setting `pairs` equal to the result of calling `sns.pairplot()` with arguments `pct` and `hue="quint"`. The diagonal will be a set of overlapping density plots for each level of education in each quintile, and the off-diagonal elements will be pairwise scatter plots.
24. Save the figure by calling `.savefig()` on `pairs` with the argument `"scatter.png"`.

D. Markdown file results.md

1. Edit `results.md` and fill in answers to the questions.

Submitting

Once you're happy with everything and have committed all of the changes to your local repository, please push the changes to GitHub. At that point, you're done: you have submitted your answer.

Tips

- A successful API request that returns data will have a status code of 200. If the request is properly constructed but doesn't return any data the status code will be 204. That's almost certainly an indication that you asked for something that doesn't exist, such as state 136 instead of state 36. The API won't tell you that 136 is not a legal state code; rather, it will just return nothing. Other status codes, such as 400, occur when the structure of the request is invalid. An example would be using `'statex:36'` in the `in` clause, where `state` has been misspelled. In this case, `response.text` will provide a very terse error message.
- The general form of the Assert statement is `assert logical_condition` where `logical_condition` is a condition that is supposed to be true when the script is working correctly. If the condition is NOT true, the script stops with an assertion error. For example, if you just wanted the script to crash if the status code was not 200 (rather than printing out the messages discussed above), the following would do the trick:

  ```
  assert response.status_code == 200
  ```

  If the status code is, in fact, 200, the assertion is true and the script continues. Otherwise, the script stops. We set things up a little differently here because mistakes are very common when initially setting up API calls and it's helpful to have a little information about what went wrong.
- Using a CSV file to define the aggregation is very handy and much cleaner and less error-prone than hard-coding it in the script itself. For example, it's much easier to make sure that each variable is linked to exactly one aggregate group. It also makes it easy to change the aggregation later: it's just a matter of editing the CSV file and rerunning the script.