Exercise: Residential Solar in NY

Summary

This exercises explores data on residential solar installations in New York by county. However, the main purpose is really to illustrate some key features of the `pandas` module.

Input Data

The input data is contained in **res_solar_by_county.csv**. It contains information on completed residential solar installations in New York State between 2000 and 2019. It has five fields: `"County"`; `"projects"`, the number of installations in the county; `"total_cost"`, the total cost of all the installations; `"total_incentive"`, the total subsidy provided by New York State; and `"total_kw"`, the total capacity built, in kilowatts. Note that `"total_cost"`, `"total_incentive"` and `"total_kw"` are sums over all the projects in the corresponding county.

Deliverables

Please prepare a script called **solar.py** that carries out the steps described below.

Instructions

1. Import the `pandas` module as `pd`
2. By default, Pandas limits the number of rows it displays when a dataframe and or series is printed. That's convenient for very large datasets when no one would want to see every row but we're going to override it here. Set the maximum number of rows to print to `None` as shown below to remove the limit.

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

3. Create a variable called `solar` by calling `pd.read_csv()` with argument `"res_solar_by_county.csv"`. It will be a Pandas DataFrame object.
4. By default, the index of `solar` will be row numbers starting with 0. We'll change it to be the county instead, which is much more useful. To do that, add a line setting `solar` equal to the result of calling the `.set_index()` method on `solar` with the argument `"County"`. See the tips section for a note about how to avoid this step by setting the index during `read_csv()`.
5. Print an appropriate heading and then print the list of column names for `solar`. The column names will be a Pandas Index object; you may want to use `list()` to turn that into a simple list, which will look a little clearer when printed.
6. Print an appropriate heading and then print the index for `solar`. As with the columns, you may want to clean up the output using `list()`.
7. Create a variable called `count` that is equal to the `"projects"` column of `solar`. It will be a Pandas Series object, which is much like a dictionary with the counties as keys and the number of projects as the values.
8. Print an appropriate heading and then print `count`.
9. Now create a list called `some_cny_counties` that contains `"Onondaga"`, `"Oswego"`, and `"Wayne"`.
10. Print an appropriate heading and then print `count[some_cny_counties]`. Notice that it's a Series object since several index values (the counties) were given.

11. Now print an appropriate heading and the print the value of `count` for `"Albany"`. Notice that the value is a scalar, not a Series, since only a single element was requested.
12. Now create a variable called `high_to_low` that is the result of calling the `.sort_values()` method on `count` with the argument `ascending=False`.
13. Then create a variable called `top_five` that is equal to the result of using the selector `.iloc[0:5]` on `high_to_low`. The result will be a new series of the five counties with the most projects. If the notation isn't clear recall that `0:5` in a subscript context is shorthand for a list of 5 values starting with 0 and ending with 4. The `.iloc` selector ignores the index and picks out elements based entirely on order. Also, be sure to note that `.iloc` uses square brackets, not parentheses: it's a subscripting operation, not a function or method call.
14. Print an appropriate heading and then print `top_five`.
15. Create a variable called `means` that is the result of calling the `.div()` method on `solar` using the arguments `count` and `axis="index"`. That will divide each column of `solar` by the `count` Series.

    This is a good demonstration of the power of the Pandas: a single short line can create a whole new dataframe where all of the columns in the original dataframe have been divided by the number of projects in the corresponding county.

    That will be true even if the rows of `count` and `solar` are not in the same order: Pandas will automatically match, or "align", the indexes of `count` and `solar` during the calculation. If you want to check this, try the same calculation dividing by `high_to_low` instead of `count`. Even though the rows of `high_to_low` are in a different order from the rows of `solar`, the result will be exactly the same.

    The Tips section below explains what the `axis='index'` argument does and why it's needed.
16. Print an appropriate heading and then print `means`.
17. Now we'll compare the means for all the counties to those for Onondaga. As a first step, create a variable called `onondaga_row` that is equal to the result of using the selector `.loc['Onondaga']` on `means` to pull out the Onondaga row.
18. Now create a variable called `relative` by dividing `means` by `onondaga_row` using a plain division sign. The result will be a new dataframe giving each county's mean variables as ratios to those for Onondaga. For example, the ratio for mean total cost in the Bronx will be 0.82, or 82%, of the mean cost in Onondaga. The `.div()` method and the `axis` keyword are not needed in this context; see the tips section for an explanation.
19. Round the results to two decimal places by setting `relative` equal to the result of applying the `.round()` method to `relative` using the argument 2. Note that the only argument is the number of digits since the thing to be rounded is given by the object itself.
20. Create a variable called `rel_incent` that is equal to the `"total_incentive"` column of `relative`.
21. Print a suitable heading and then print `rel_incent` sorted in ascending order using its `.sort_values()` method.
22. There's no Markdown deliverable for this assignment but it's interesting to note that there are large differences in the mean incentive across counties. That's partially, but not completely, explained by differences in the average size of the projects.

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

- Reading the file and setting the index are done as two separate steps here to illustrate how they work. However, it's possible to do both at once by adding `index_col="County"` to the call to `read_csv()`. That's often a very good idea.
- When carrying out an operation that involves two objects that aren't the same shape, such as a dataframe and a series, Pandas will automatically sweep the smaller object across the large one in a process known as "broadcasting". However, it has to deal with an inherent ambiguity: should the smaller object be treated like a column, and swept across the columns of the large object, or should it be treated like a row and swept down the larger object's rows.

  The `.div(count,axis='index')` call resolves the ambiguity by telling Pandas to treat `count` as a column and line it up with the index of the dataframe. To have Pandas treat something as a row, as was the case with `onondaga_row`, the call would be `.div(onondaga_row,axis='columns')`.

  However, when given an expression that links two objects by a simple operator, such as `means/onondaga_row`, rather than a call like `.div()`, Pandas will *assume* that it should use row-wise broadcasting. That's why `means/onondaga_row` worked as intended.

  Be aware of this behavior: it's handy but if you forget and accidentally use something like `some_data_frame/some_column` you won't get what you expect and the result will usually be full of missing data.