Web Scraping II

ISTA 350 Hw7, Due 4/23/2020 at 11:59 pm

Introduction. This homework is the second part of an introduction to the art of web scraping. In it, you will take the three html documents you scraped and saved in hw6, parse them to extract their data and convert that data into usable form, clean the data, and save the results in `json` objects. Along the way, you will meet the `json` module and dive much deeper into `bs4`.

Instructions. Copy your `hw6.py` module and rename the copy `hw7.py`. You will also need to copy the html files you created in hw6 into the folder with your hw7 module because you will be picking up where you left off. Below is the spec for six new functions and some additions to `main`. Implement them and upload your module to the D2L Assignments folder.

Testing. Download `hw7_test.py` and the associated files necessary for testing and put them in the same folder as your `hw7.py` module. Run it from the command line to see your current correctness score. Each of the six new functions in `hw7.py` is worth 16.7% of your correctness score. The test file we will use to grade your program will be different and may uncover failings in your work not evident upon testing with the provided file. Add any necessary tests to make sure your code works in all cases.

Documentation. Your modules must contain a header docstring with your name, your section leader's name, the date, `ISTA 350 Hw7`, and a brief summary of the module. Each method/function must contain a docstring. Each docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

Grading. Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code (not necessary on this assignment).

Collaboration. Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources/collaborators in your header docstring. Leaving this out is dishonest.

Resources.
https://pypi.python.org/pypi/beautifulsoup4/
http://www.crummy.com/software/BeautifulSoup/bs4/doc/
https://docs.python.org/3/library/json.html

`hw7.py`:

`is_num`: this Boolean function takes a string and determines whether or not it represents a number (either integer or floating point).

`load_lists`: this function takes a soup object and a flag as arguments and returns a list of lists containing the useful data in the soup object. The soup object contains an html parse tree that describes a table of data. You will extract the data from the parse tree and store it in the list of lists. In the process of doing so, you will transpose the data so that the columns in the table are rows in the list of lists and vice versa. You may approach this however you wish, but I suggest nested `for` loops. In the

outer loop, you traverse the document's table rows (using the technique presented in class) and in the inner loop you traverse each row's table data fields. The first datum in each row should be a year, which you will need to convert to an `int`. When you reach a row that doesn't have a year as its first datum, you have finished assimilating all of the raw data. The table data fields in each row other than the first may or may not contain useful data. Check this using `is_num`. If the table data field is `'-----'`, place the flag in the list of lists in its place. As an example:

`load_lists(`

2000	1.75	-----	0.89
2001	1.03	0.79	1.00
2002	0.98	1.01	0.43
2003	2.01	0.56	0.57
2004	-----	0.99	1.11

`, -999)`

```
-->    [[2000, 2001, 2002, 2003, 2004],
        [1.75, 1.03, 0.98, 2.01, -999],
        [-999, 0.79, 1.01, 0.56, 0.99],
        [0.89, 1.00, 0.43, 0.57, 1.11]]
```

`replace_na`: 'na' is an abbreviation for not available. This is standard jargon for missing data. In our case, we have replaced all missing data with the flag `-999` as we loaded our list of lists. We want to clean our data by putting in reasonable values where data was missing. We are particularly interested in trends with time, so replacing missing data with averages of data from nearby years is a natural approach. The data for a given month through the years is represented by a row (because we transposed it from the website format). Therefore, when confronted with the flag in a position in a row, we will take the data from the 5 previous positions in that month's row and 5 following positions and use this to calculate an average. In `clean_data`, we will replace the flag with the average. In this function, we will calculate and return that average for `clean_data` to use. We must delete any occurrences of the flag in the 5 following years, as that would really mess up the average.

This function returns a replacement value for `data[r][c]` with the average of the surrounding 10 years. Its first parameter is the list, the second and third are the row and column, respectively, the next is the flag, and the last is a precision with a default value of 5. If one of the surrounding years also contains the flag, leave that position out of the average. Round the replacement value to the precision specified by the last argument.

`clean_data`: this function traverses the list of lists and every time it finds the flag, it calls `replace_na` to replace the flag. Its parameters are the list, the flag, and a precision with a default value of 5 to be passed on to `replace_na`.

`recalculate_annual_data`: on the website, the last column is the total rainfall for the year or the average annual temperature. We have transposed this data, so this information is now in the last row, i.e. the last inner list. Because we have replace missing data with reasonable approximations, the data in our annual column no longer matches the value calculated from the monthly data. Therefore, we need to recalculate our annual data.

This function has three parameters. The first is the list of lists (we are recalculating the last row); the second a Boolean with a default value of `False`. The Boolean argument is `True` if the annual data should be averages (temperature data); `False` if they should be totals (precipitation data). The third argument is a precision with a default value of 5. Round the recalculated annual data to this precision. In order to minimize round-off errors messing with the test, when recalculating averages, round the sum before dividing by *N*, then round again after dividing.

`clean_and_jsonify`: this function takes three arguments. The first is a list of filenames to be cleaned and saved to files as json objects. The second is the flag. The third is a precision to be passed on to functions that `clean_and_jasonify` calls. It has a default value of 5. For each file in the first argument, get soup, transform it into a list of lists, clean the list, recalculate the annual data, and store it in a file as a json object (as described in class). Name your JSON files the same as your html files but with the extension `.json`. So your `wrcc_pcpn.html` will result in a file called `wrcc_pcpn.json`.

`main`: add these two lines:

```
    fnames = ['wrcc_pcpn.html','wrcc_mint.html', 'wrcc_maxt.html']
    clean_and_jsonify(fnames, -999, 2)
```