Exercise: Working with Large Zip Files

Summary

This exercise focuses on techniques that are useful for working with large datasets stored in CSV format within zipped archives.

Input Data

The input data is a file called **id114_2014.zip** that will need to be downloaded from the course Google Drive link on the main course web page. Please resist the temptation to unzip the file: the contents are large (72 MB) and part of the point of the exercise is to work with the data without unzipping the file first.

The zip file contains a single large CSV file called **id114_2014.csv**. It contains data from the Pecan Street Project, a research project in Austin, Texas, that collects high resolution data on household electricity consumption. This file particular file is the data for household 114 in 2014.

The file has one record for every minute (around 525,600 in all). Each record contains electricity use for up to 67 circuits in the house, though not all are used for this particular household. In this exercise we'll only use two of the fields: the timestamp for the record, `localminute`, which shows the date and time of the data in Austin's local time zone, and the household's total usage, `use`, in kilowatts (kW). The timestamp is the second field in each record and the usage is the third field. If you'd like to see what the records in the CSV file look like without unzipping the file, see **firstlines.csv** in the Google Drive folder: it has the file's first 10 lines.

If you want to run the `demo.py` script for this exercise, you'll also need to download **demo.zip**. It has data on ISO country codes but it isn't necessary for the actual assignment.

Deliverables

A script called **pecan.py** that goes through the steps below to produce a sorted file of average hourly usage data called `usage.csv`, and an updated version of the Markdown file **results.md**. Note that `usage.csv` itself is not a deliverable and will not be uploaded to GitHub when you push your repository: the .gitignore file in the directory tells GitHub not to upload zip or csv files. I'll build a copy by running your script.

Instructions

A. Initial setup

1. Import the following modules: `csv`, `io`, and `zipfile`.
2. Import the `numpy` module as `np`.
3. Import `defaultdict` from module `collections`.
4. Open the CSV file within the zip archive following the process shown in `demo.py`. There are three steps: (1) using `zipfile.ZipFile()` to set up a zip object for working with the archive as a whole; (2) using that object's `open()` call to open file `id114_2014.csv` within the archive in binary (bytes) mode; and (3) using a call to `io.TextIOWrapper()` to handle character encoding by converting the bytes into strings.
5. Set `inp_reader` to the result of calling `csv.DictReader()` on `inp_handle`.
6. Set `out_handle` to the result of calling `open()` with arguments `"usage.csv"`, `"w"`, and `newline=""`. As before, the `newline` argument is needed to keep the file from having extra returns when it is used with the CSV module. See the tips section for an explanation of the benefits of opening output files early in a script.
7. Set `out_writer` to the result of calling `csv.writer()` on argument `out_handle`. Note that for this exercise we're using `csv.writer()` instead of `csv.DictWriter()` because the data we'll be writing isn't a list of dictionaries.

B. Reading and grouping the data

1. Set `hourly` to the result of calling `defaultdict()` on `list` to create a dictionary for grouping the data by hour.
2. Use `rec` to loop over `inp_reader`. Within the loop do the following.
    1. Skip any records where the value of `"use"` in `rec` is missing by using an `if` statement to check whether `rec["use"]` is `""` (two quotes with no space between them). If that's the case, use a `continue` statement to go on to the next record.
    2. Set variable `ts` (short for timestamp) to the value of `"localminute"` in `rec`. Then set `use` to the result of calling `float()` on the value of `"use"` in `rec`.
    3. Now we'll take apart the timestamp in order to build a key that will allow the data to be grouped by hour and later sorted correctly. Split `ts` on spaces and call the pieces `date` and `time`. A convenient way to do that is to set the tuple `(date,time)` to the result of the `split()` function. Then split `date` using `"/"` and call the pieces `mo`,`dy`, and `yr`. Finally, split `time` into `hr` and `mi` using `":"`.
    4. Create a tuple called `hour` that consists of `int(mo)`, `int(dy)` and `int(hr)`. It uniquely identifies the hour within the year.
    5. Append `use` to the value of `hourly` for key `hour`.

C. Computing and saving the hourly averages

1. Following the loop above, call `out_writer.writerow()` on the list `['month','day','hour','usage']`. That will write a list of column names to the output file.
2. Create an empty list called `averages`.
3. Use variable `hour` to loop over `sorted(hourly)`. Within the loop do this:
    1. Set `values` to the value of `hourly` for `hour`.
    2. Check whether the length of `values` is greater than 60. If it is, print out a message indicating the `hour` that is being dropped and then use a `continue` statement to go on to the next entry. The extra minutes happen due to daylight savings time: an hour in November is repeated when the clock shifts back. We'll drop it just to illustrate how to remove inconsistent data. There's also a missing hour in the spring when the clock shifts forward, and a few hours where some minutes are missing. We won't do anything special about the missing data. See the tips section for more information about local time and daylight savings.
    3. Set variable `avg` to the result of calling `np.mean()` on `values` and then rounding the result to 3 decimal places.
    4. Set the tuple `(mo,dy,hr)` to `hour` to pull out the components of the `hour` tuple and store them in `mo`, `dy`, and `hr`.
    5. Call `out_writer.writerow()` on the list `[mo,dy,hr,avg]` to write the results to the output file in CSV format. The month, day, and hour organization would be very convenient for joining on temperature and weather data later for statistical analysis of the determinants of electricity demand.
    6. Append `avg` to the `averages` list.

D. Summarizing usage volatility

1. An important feature of electricity consumption is that it has sharp spikes: that is, it has a small number of very high values. To see this, we'll look at some percentiles for this data. Set `pctiles` to a list of the following percentiles: 1, 5, 10, 25, 50, 75, 90, 95, 99.
2. Now set `pctvals` to the result of calling `np.percentile()` on the arguments `averages` and `pctiles`. That will calculate the average usage at each percentile cutoff.
3. Use the tuple `(pct,kw)` to loop over the result of calling the `zip()` function on arguments `pctiles` and `pctvals`. The `zip()` function creates a list of tuples by pairing the corresponding elements of the input

lists. Within the loop call `print()` on the string `f"{pct:2d} %: {val:4.3f} kW"`. The `:2d` says that `pct` should be printed as an integer (`d`) using 2 spaces; it will right-align the percentages even though the first one is only one digit. The `4.3f` says that `val` should be printed using 4 digits total, of which 3 should be to the right of the decimal point; it has the effect of lining up all the usages. If all goes well, the result should be a nicely formatted table.

E. Updating results.md

1. Edit the Markdown file `results.md` and replace the `TBD` placeholders with answers based on your percentile results.

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

- It's handy to open the output file *before* reading the whole input file so the script will crash immediately if opening the output file fails. A common cause of open failures is having the last output file open in Excel and then trying to rerun the script. Excel locks the file to prevent other applications from using it at the same time, and that will cause the script to crash with a permission denied error. If your script opens the output file *after* processing the input file you may end up waiting a long time for the input processing to finish only to have the script crash at the end.
- There are many other ways to translate timestamp strings into numerical values beside using `split()` repeatedly. In this case, however, `split()` is substantially faster than most of the alternatives. That matters because this file is only one small part of a large dataset with multiple years of data and hundreds of households.
- Using the `sorted()` call with tuples is very handy. It's especially useful when the data will be viewed by a person: you can order the elements of the tuples so the data will be grouped in the most appropriate way.
- The daylight savings issue comes about because the instrument used to record the data is set to local time (i.e., what a clock on the wall would show). An alternative would be to use Coordinated Universal Time (UTC), a world-wide standard that has no adjustments for daylight savings or time zones. If you're ever working on datasets that need to be synchronized, and have a say in how the timestamps are set up, UTC is often better than local time.