Exercise: Working with Large Zip Files

Summary

This exercise focuses on Matplotlib and Pandas features that are useful for working with large datasets.

Input Data

The input data is a file called **id114_2014.zip** that will need to be downloaded from the course Google Drive. Please **do not** 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. If you'd like to see what the data looks like, see **firstlines.csv** in the Google Drive folder: it has the file's first 10 lines.

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 during the year (around 525,600 in all). Each record contains electricity use for up to 67 circuits, though not all of the circuits 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.

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`, a figure called **load_duration.png** that shows the load-duration curve for the household, a figure called **usage.png** that compares electricity use in January and July, 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

The first steps will load the data and filter out records that are missing either `"localminute"` or `"use"`.

1. Import `pandas` as `pd` and `matplotlib.pyplot` as `plt`.

2. Set the default DPI for plots to 300 by setting the Matplotlib parameter dictionary entry `plt.rcParams["figure.dpi"]` to `300`. The change will only affect plots made by this script: it's not permanent.

3. Create a variable called `raw` by calling `pd.read_csv()` with argument `"id114_2014.zip"`. A nice feature of `.read_csv()` is that it can read CSV files inside zip files without unzipping them first as long as the zip file contains just a single CSV file.

4. Create variable `start` by calling the `.head()` method of `raw` to select the first few rows. Open `start` in Spyder's Variable Explorer to see what it looks like.

5. Drop rows from `raw` that are missing key data by creating a variable called `usable` that is equal to the result of calling the `.dropna()` method of `raw` with the argument `subset=["localminute","use"]`. That will drop any record for which either `"localminute"` or `"use"` has missing data ("not available" in Pandas terminology).

6. Print a message giving the number of dropped records. The number is the difference between the lengths of the dataframes: `len(raw)-len(usable)`.

B. Building a trimmed dataframe

Next we'll build a trimmed dataframe with just the data we'll need later. In the process we'll split up the `"localminute"` field into its components.

1. Create a variable called `trim` that is equal to the result of calling the `.str.split()` method of `usable["localminute"]` with the arguments `r"/| |:"` and `expand=True`. The `r"` indicates that the string is a "regular expression" (RE) that will be used for matching patterns rather than literal characters. For clarity, the string inside the quotes is a forward slash (/), a vertical bar (|), a space, another vertical bar (|), and a colon (:). The RE will match and split on either a slash, a space, or a colon. Since `"localminute"` is a string like "1/6/2014 18:18" the result will be a dataframe that has rows like `1,6,2014,18,18`: that is, five columns corresponding to the month, day, year, hour and minute.
2. Create a dictionary called `colnames` that has the following key-value pairs: `0:"mo"`, `1:"dy"`, `2:"yr"`, `3:"hr"`, `4:"mi"`. Note that the keys are numbers, not strings.
3. Set `trim` equal to the result of calling the `.rename()` method of `trim` using the argument `columns=colnames`. Check `trim` in Variable Explorer to make sure the columns have been renamed correctly.
4. Set `trim` to the result of calling the `.astype(int)` method of `trim`. The effect will be to convert all the data from strings to integers.
5. Copy the `"use"` data into `trim` by setting column `"use"` in `trim` equal to the `"use"` column in `usable`.
6. Print the result of applying the `.head()` method to `trim`. You should see 5 rows with the index and columns for `"mo"`, `"dy"`, `"yr"`, `"hr"`, `"mi"` and `"use"`.
7. Now set `trim` to the result of calling the `.set_index()` method of `trim` with the following list as its argument: `["mo","dy","yr","hr","mi"]`. The result will be a dataframe with the date and time as its index.
8. Finally, create a variable called `use_by_min` that is equal to the `"use"` column of `trim`.

C. Grouping and aggregation

Next we'll aggregate the data to average use by hour.

1. Create a variable called `group_by_hour` that is equal to the result of calling the `.groupby()` method of `use_by_min` with the following list as the argument: `["mo","dy","hr"]`. Be sure to note that the last element is the hour, not the year. The resulting object groups all of the 1-minute records by the day and hour in which they occur.
2. Create a variable called `minutes` that is equal to the result of calling the `.size()` method of `group_by_hour`. The result will be a series where the index indicates the day and hour and the value indicates the number of 1-minute records found for that hour.
3. Create a variable called `mean_use` that is equal to the result of calling the `.mean()` method of `group_by_hour` and then applying the `.round()` method with argument `3` to round the averages to 3 places.
4. To illustrate how to detect and remove inconsistent data, we'll remove an hour that is recorded twice due to the end of daylight savings time. Create a variable called `fall_back` that is equal to `minutes > 60`. The result will be a series with `True` in each row that has more than 60 minutes and `False` in all other rows. There should be exactly one row with `True`: the hour when daylight savings ended and clocks were changed back by an hour. It has 120 minutes since it ends up being recorded twice.

5. Print information about that row by printing `mean_use[fall_back]`, which will print any rows where `fall_back` is `True`. This is an example of a Pandas feature known as "boolean selection": that is, selecting rows based on a sequence of `True` and `False` values. Only rows where the value is not `False` or `0` are kept.
6. Remove the daylight savings day by setting `mean_use` equal to the value of `mean_use[ fall_back == False ]`. Be sure to note that a double equals sign is needed for the test inside the square brackets.
7. Sort the data by date and time by setting `mean_use` to the result of calling the `.sort_index()` method of `mean_use`. The records will be sorted by month, then day, and then hour.
8. Write out the data by calling the `.to_csv()` method of `mean_use` with the argument `"usage.csv"`.

D. Analyzing the hourly data

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. Create a variable called `cutoffs` that is equal to a list of the following values: `0.01`, `0.05`, `0.10`, `0.25`, `0.50`, `0.75`, `0.90`, `0.95`, `0.99`. That is, the list is the decimal versions of 1%, 5%, and so on.
2. Now create a variable called `pct` that is equal to the result of calling the `.quantile()` method of `mean_use` with the argument `cutoffs`. It will be a series giving the hourly mean usage at each of the percentiles in `cutoffs`.
3. Print an informative heading and then print `pct`. The results will be used to answer questions in the "results.md" file.
4. Next we'll plot the "load-duration" curve for the data. A load-duration curve shows the number of hours per year that electricity use is at or above a given level. Start by setting variable `load_duration` equal to the result of sorting the hourly means from largest to smallest by calling the `.sort_values()` method of `mean_use` with the argument `ascending=False`.
5. Now set `load_duration` equal to the result of calling the `.reset_index()` method of `load_duration` with the argument `drop=True`. This drops the existing index and replaces it with a simple sequence of integers.
6. The index now starts with `0` following the Python standard but for a true load-duration curve it should start with `1` to indicate that the first value is the load for one hour a year. To add 1 to each index entry, set `load_duration.index` equal to `load_duration.index + 1`. The series will now start with 1 and will show the number of hours during the year that the load is at or above the corresponding `"use"` value.
7. Create a figure by setting `fig0,ax0` equal to the result of calling `plt.subplots()`.
8. Add a graph to `ax0` by calling the `.plot.line()` method of `load_duration` with the arguments `ax=ax0` and `title="Load Duration Curve for House 114"`.
9. Add a Y axis label by calling the `.set_ylabel()` method of `ax0` with the argument `"kW (red=5th and 95th percentile`
10. Add an X axis label by calling the `.set_xlabel()` method of `ax0` with the argument `"Hours"`.
11. Add a red horizontal line at the 5th percentile by calling the `.axhline()` method of `ax0` with four arguments: `pct[0.05]`, `c="r"`, `ls="--"`, and `lw=1`. The "ax" in the method name is a reminder that it applies to Axes objects, and the "hline" indicates that it draws a horizontal line. The `pct[0.05]` argument says where to draw the line, the `c="r"` argument sets the line color to red, the `ls="--"` argument sets the line style to dashed, and the `lw=1` argument sets the line weight to 1 (relatively thin).

Note that the `0.05` is a number, not a string, since it refers to the numeric value used in building the quantiles in `pct`.

12. Using a similar approach, add a second red line at the 95th percentile.
13. Call the `.tight_layout()` method of `fig0` to make sure the spacing of the figure is correct.
14. Call the `.savefig()` method of `fig0` with the argument `"load_duration.png"` to save the figure.

E. Examining the minute data in more detail

1. Set variable `mo01` equal to the result of calling the `.xs()` method of `use_by_min` with two arguments: `1` and `level="mo"`. That will pick out data for January.
2. Use a similar statement to set variable `mo07` to the data for month `7`, July.
3. Create a variable called `nbins` set to `100`. It will be the number of bins in a pair of histograms.
4. Create a vertical two-panel figure by setting `fig1, (ax1,ax2)` to the result of calling `plt.subplots()` with three arguments: `2`, `1`, and `sharex=True`.
5. Add a figure title by calling the `.suptitle()` method of `fig1` with the argument `"Electrical Load for House 114 (1-`
6. Add a histogram for January to the top panel of the figure (`ax1`) by calling the `.plot.hist()` method of `mo01` with arguments `bins=nbins`, `ax=ax1`, and `title="January"`.
7. Add a red vertical line at the mean by calling the `.axvline()` method of `ax1` with two arguments: `x=mo01.mean()` and `color="r"`.
8. Add a green vertical line at the median by calling the `.axvline()` method of `ax1` with two arguments: `x=mo01.median()` and `color="g"`.
9. Add a histogram for July to the bottom panel of the figure (`ax2`) by calling the `.plot.hist()` method of `mo07` with arguments `bins=nbins`, `ax=ax2`, and `title="July"`.
10. Follow the process used above add vertical lines for July's mean and median.
11. Add an X axis label to the lower panel by calling the `.set_xlabel()` method of `ax2` with the argument `"kW (red=mean, green=median)"`.
12. Call the `.tight_layout()` method of `fig1` to make sure the spacing of the figure is correct.
13. Call the `.savefig()` method of `fig1` with the argument `"usage.png"` to save the figure.

F. Updating results.md

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

- There are many other ways to translate timestamp strings into numerical values beside using `split()`. 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.
- 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). In addition to the hour in the fall with 120 minutes, there's an hour in the spring when clocks are moved forward that has 0 minutes. 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.