Northeastern University
DS2001 – Programming with Data — Social Science Practicum
Fall 2019, Mattsson & McCabe

Practicum 8: Dictionaries

In this week's practicum we will be using dictionaries to simplify interacting with and summarizing datasets, using the same data we studied last week.

1 Fatal Encounters with Police (US)

Following the fatal shooting of Michael Brown in Ferguson, Mo. on Aug. 9, 2014, there has been increased attention on collecting and analyzing data related to fatal police encounters in the US. You can read more about the dialogue around this issue and specific cases which have raised questions about officer conduct here: Fatal Police Shootings: Accounts Since Ferguson, New York Times
Since Jan. 1, 2015, The Washington Post has been compiling a database of every fatal shooting in the US by a police officer in the line of duty. It is difficult to find reliable data from before this period. The Washington Post is tracking details about each killing - including the race, age and gender of the deceased. They have gathered this information from law enforcement websites, local news reports, social media, independent databases, and their own reporting.
This description of the data, and the version we use, was compiled by Karolina Wullum. It is available here: Fatal Police Shootings in the US.

1.1 Download

Please download `pr08.zip` from the practicum website (or Blackboard), unzip it, and move the directory/folder to where you want it in your file system. Use Atom's `File > "Add Project Folder..."` to open this folder and view `dictionaries.py`, `data.py`, and `fatal_encounters.csv`.

1.2 Moving from a list-of-lists to a list-of-dicts

For the past few weeks, we have been working with datasets in list-of-list form. This has some advantages—we can do almost anything we want with numeric keys and lists are pretty familiar at this point—but also some major disadvantages. In particular, we have to store a separate header variable containing the names of the columns, and then use that to figure out which numeric index we want. If we store the dataset as a list of dictionaries instead, we no longer need that header variable.

1.3 The `csv` module

Last week we focused on writing our own functions to load data from a file using low-level operations on files and string methods. The particular form of file we examined—comma-separated value files—are very common, and Python's standard library provides a module for interacting with them. The `DictReader` class from the `csv` module takes an open file pointer (like we made last week with the `open` function), and gives us a dictionary for each row. On the state population data (`state_population.csv`), an entry will look like the following:

```
{'state': 'AL',
 'white': '72.51',
 'black': '23.52',
```

```
 'native_american': '0.66',
 'asian': '0.48',
 'hispanic': '2.98'}
```

1.3.1 A note on `dict`s and `OrderedDict`s

Actually, your output will probably look slightly different from what was written above if you are running Python 3.7. Python `dict`s were historically unordered, so the ordering of the data values coming out of the `DictReader` did not necessarily match the ordering in the columns in the csv. To address this `DictReader` now returns an `OrderedDict`, which behaves like a regular `dict` but has a fixed order. All of this is largely moot now because, as of Python 3.7, regular `dict`s are also ordered! If you want to cast the `OrderedDict`s back to `dict`s, just use the `dict()` function with an `OrderedDict` as input, but it shouldn't matter either way for this exercise.

1.4 Loading the data

Similar to last week, we are going to write a function for our `data.py` file. Instead of `data_header` and `data_as_list_of_lists` functions, we will write a single `load_data` function that takes a filename (`str`) as input and:

1. opens the file
2. creates a `DictReader` from the file
3. creates a list of dictionaries from the `DictReader`
4. closes the file
5. returns the list of dictionaries

After implementing this function in `data.py`, import the `data` module in `dictionaries.py` and use the function to load the Fatal Encounters data.
Hint: the `DictReader` gives us a generator that we can iterate through like a list, but it isn't a list as such. We can turn a generator into a list in two ways: first, by casting it to a list with the `list()` function, similar to how we cast `int`s to `float`s using the `float()` function; second, by using a list comprehension: `[x for x in generator]`.

1.5 View and summarize the dataset

The `head` function we used in the past won't serve us well any more, because we don't have a separate header and data files. Instead, to get an idea of what our data looks like, use list slicing to print the first five rows of the dataset. Ensure that each element is a dictionary.

2 Revisiting data functions

2.1 `get_column`, revisited

In previous weeks we used a `get_column` function that took our list-of-lists dataset and an integer-valued index, then returned a list of values corresponding to the ith "column" (the ith index in each list) in the dataset. This served us well for some cases, but was often pretty tedious, because we had to count out the index of the column we wanted. Now that we have dictionaries, we can rewrite this to use the name of the column directly. So, on the Fatal Encounters dataset, instead of writing `get_column(data, 2)` to get a list of ages, we can now write `get_column(data, 'age')`.

One weakness of using the approaches to reading files we've used this week and last week is that all of our variables are strings. (Look at the example output from Section 1.3 above.). We're going to add another parameter to our `get_column` function, indicating whether or not we want to get numeric or string output back. In total, it will take three inputs: a list-of-dicts dataset, a column name corresponding to a key in the dicts, and a boolean indicator for whether or not numeric or string output should be returned.

2.2 Answering a question with get_column

With this in hand, let's answer a question. Use your `get_column` function in combination with our friends `max`, `min`, and `mean` to find the average age of those killed in police-involved shootings, as well as the oldest and youngest individuals.

2.3 Using `Counters` to... count things

Python's standard library has another module that is useful when analyzing data. The `collections` module has a `Counter` data structure that is a special kind of dictionary. Give it a list of items, and it will give you back a dictionary where the keys are the unique items and the values are the number of times each appeared in the list. For example:

```
>>> from collections import Counter
>>> data = ['a', 'a', 'b', 'b', 'b', 'c', 'ab']
>>> counts = Counter(data)
>>> counts
Counter({'a': 2, 'b': 3, 'c': 1, 'ab': 1})
>>> counts['a']
2
```

The `Counter` is useful in its own right for storing counts of things, but it also has a useful method, `most_common`, that returns the n items with the highest counts.

```
>>> counts.most_common(2)
[('b', 3), ('a', 2)]
```

Last week, we wanted to focus our analysis on states where there were enough shootings for the proportions to not be too noisy, but we didn't have a convenient way to test which states had a large sample without guesswork. This week, use a `Counter` to establish which states had over 100 police-involved shootings in the time period we're studying.

3 Revisiting disparities in fatalities

3.1 Working with state population data

It makes sense to work with the fatal encounters data as a list of dicts, because each row is a specific case and we want to aggregate over these cases. We could have created a dictionary of dictionaries, maybe using the name as a key. So we could have accessed the age of a particular victim with `data['John Doe']['age']`. This isn't very useful to use because (i) we have no guarantees the names will be unique, and (ii) names are pretty unwieldy to use as keys anyway. However, the state population data is a case where a dict-of-dicts structure would be very convenient, because then we could access California's demographics directly, without having to figure out which integer index corresponds to California. That way, if we want to find the proportion of whites in California, we can just write `data['CA']['white']`.

In `data.py`, we have provided a `list_to_dict` function that takes a list-of-dicts and makes a dict-of-dicts, replacing the integer keys of the list with a key you specify (that should correspond to one of the columns in the CSV). Use your `load_data` function to open `state_population`, then use `list_to_dict` to turn it from a list-of-dicts to a dict-of-dicts. To confirm that it works, use a query similar to what we used in the above paragraph to find the proportion of whites in California.

3.2 Putting it all together: another look at disparities

With this in place, we have all the components we need to re-consider our analysis from last week. So, again, pick a demographic group and a state with > 100 police-involved shootings, and report: the proportion of shootings involving members of that group, and their proportion in the state's population. To subset the Fatal Encounters data by state and by race, you can either use list comprehensions or the `filter_by_categorical` function provided in `data.py`.
The results should be comparable to the analysis last week, and require some trickier concepts, but once we have these concepts in place things get a lot easier. Recall that to do this analysis last week, we had to do some fairly awkward list traversals and maybe even look at the raw data directly to get a sense of what states to consider.

4 Extra stuff!

4.1 Functions!

We have all the logic implemented to study these datasets, and we have done so for one case. Let's automate this (again) by putting it all in a function (with a lot of parameters). Write a function with the following signature:

```
def disparity_analysis(fe_data, sp_data, state, group)
```

that returns a 2-tuple with the prevalence of shootings and of population for a given group and state. Try it out and see what happens. Assume that `fe_data` is a list-of-dicts and that `sp_data` is a list-of-dicts.

4.2 Time series!

We are looking at aggregate data for a two and a half year window. Are there systematic differences over the course of this dataset?
To look at this, let's break the data out by year and month Write a function with the following signature:

```
def deaths_in_month(fe_data, year, month)
```

and returns a subset of the dataset in list-of-dicts form (this is just another variant of the `filter` functions we've written before). Use this function with some loops to print out the number of deaths in the each month of the dataset.
Some hints:

- The dates in the data are DD/MM/YY, not MM/DD/YY
- The dates just strings, so we can use `split()` here.
- The months are padded to two digits (e.g. '01')
- It's probably easiest to assume that `year` and `month` in your `deaths_in_months` will be strings, instead of dealing with the type conversion and padding the strings to two digits as necessary.

This handout was originally created by Carolina Mattsson and Stefan McCabe, Fall 2019. This exercise was originally created by Sarah Shugars, Spring 2019.