Project 2: Mapping tweets

In this project you will be analyzing data taken from Twitter. The goal is to produce a map that shows how popular tweets about a given topic are in each state. Some of the work – specifically the parts that involve reading from files and drawing graphics – has been done for you. What you need to do is the data analysis. You'll need to make a data abstraction, as well as work with existing data abstractions, lists, dictionaries, and strings. In the end, you'll be able to create maps like the one below, which shows how frequently people in each state are posting tweets containing the word "beach", with darker blues representing more tweets (per capita).

Begin by downloading project2.zip from the course website. This is a zip file, which you should unzip to create a directory. That directory should include the following files.

trends.py	A framework for implementing the data analysis. This is the file where you will do all your work.
geo.py	A data abstraction for geographic positions, as well as functions for map projections and other useful geometric calculations.
maps.py	Functions for drawing maps.
data.py	Functions for loading Twitter data from files.
graphics.py	A simple Python graphics library.

There should also be a "data" subdirectory, which contains the necessary data. (This is mostly a raw database of tweets, but also data representing the location of state borders.)

Exercise 1: The tweet data abstraction

This project uses a "tweet" data abstraction. A tweet consists not just of the actual text of the tweet, but also data about the time at which it was sent and the location from which it was sent. The trends.py file includes the constructor for this data abstraction, but the selectors are not yet complete. You must complete the constructors, `tweet_text`, `tweet_time`, and `tweet_location`. Note that `tweet_location` returns a position. The position data abstraction has already been implemented for you. Look in geo.py to see how it works. Remember to respect abstraction barriers when working with positions (and later in the assignment, when working with tweets).

Exercise 2: `check_intersect`

In order to color the states in our map, we need to know how many tweets of a given type lie inside that state. To do that, we need a function that will take a tweet and a state and tell us whether the tweet is inside that state. The algorithm we will be using is the ray casting algorithm, which you should read about at http://en.wikipedia.org/wiki/Point_in_polygon.

The algorithm works by taking a ray that shoots out in one direction from the point in question. In our case we'll be using a ray that points due east. (That makes the math simpler.) The algorithm asks how many times the ray intersects the edge of the shape/state. The end of the ray must be outside the shape, so the point is inside the shape if and only if the number of intersections is odd.

We have very high-precision locations, so we are going to simplify the problem some by assuming that the ray never passes through the point where two edges of the shape meet, and that the point is never on the edge of the polygon. We are also going to assume that we can treat the surface of the earth as flat, with longitude and latitude representing coordinates in a coordinate plane.

In order to find the number of intersections, we will check each edge of the polygon(s) that represent a given state and find out if that edge intersects with the ray. In order to do this, you must first implement `check_intersect`. This function takes as input three locations. The first is the point from which the ray is shooting east. The next two are two endpoints of a line segment. The function should return `True` if the ray intersects the line segment and `False` otherwise.

Hints for writing `check_intersect`:

The ray in question only points east, but you can think of it as part of an infinite line along that line of latitude. Checking whether the line segment in question crosses that line at all is a much easier problem. If it does cross the relevant line of latitude, then you can calculate the exact point where it crosses. (This is just the intersection of a line given by two points and a horizontal line – you might need to refresh yourself on some high school algebra you haven't used in a while.) You can then check whether the point of intersection is east or west of the start of the ray and return `True` or `False` accordingly.

Exercise 3: `is_in_state`

Now use `check_intersect` to write `is_in_state`, which takes a point and a state and returns `True` or `False` depending on whether the point is in the state in question. A state is represented as a list of polygons. Each polygon is represented as a list of locations representing the corners (in order) of the state. The first location in a polygon is repeated at the end, so that any two consecutive points represent a line segment in the border of the state. To check whether a point is in a polygon, step through each line segment in the border of the polygon and use `check_intersect` to see if the ray coming out of the point intersects that line segment. Count how many it intersects, and if the number is odd, the point is in that polygon. A point is in a state if it is in one of the polygons that make up that state.

Exercise 4: `count_tweets_by_state`

We now need to figure out what color to make each state. You can use the `us_states` variable, which contains a dictionary. The keys of the dictionary are two-letter postal codes for each state. The values are lists of polygons representing the shape of the corresponding state. (Some states, like Hawaii or Michigan, are not single connected pieces of land, so we need multiple polygons for these states.) This function should take as input a list of tweets. Its output should be a new dictionary, again with state abbreviations as the keys.

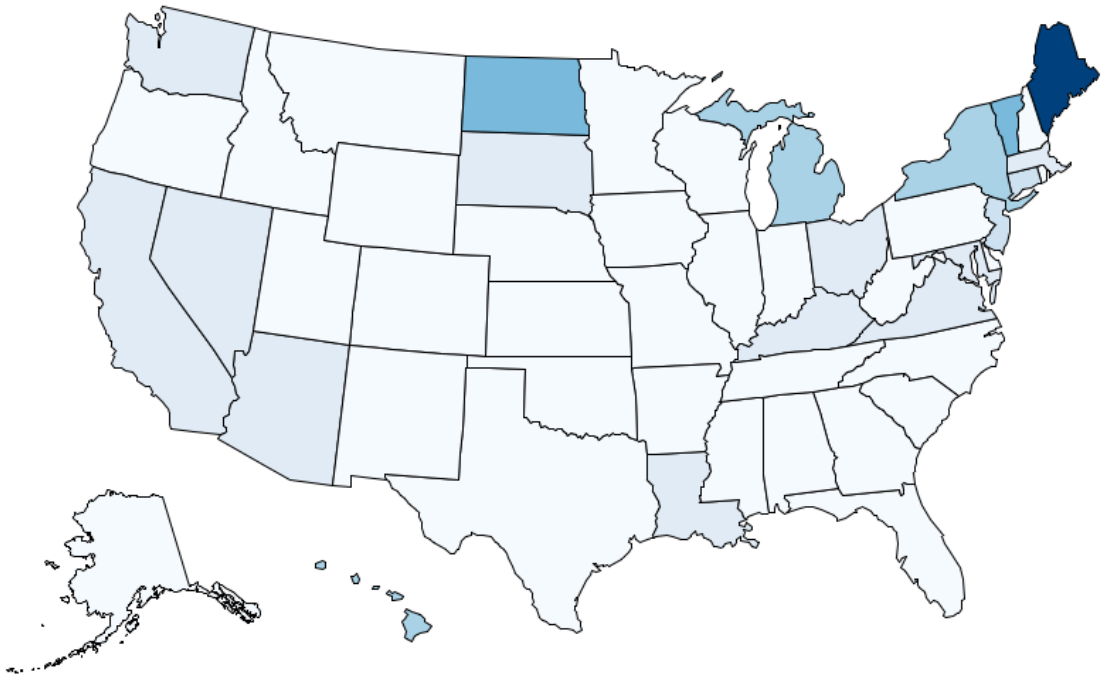
You should start by getting the output dictionary to include as values the number of tweets in the list that were in the given state. To do this, use the `is_in_state` function. You should be able to take each tweet, search through the dictionary seeing if it is in each state, and then add one to the count of tweets in whatever state it finds. (Some tweets are not in any state, and you should ignore those.)

Now if we left counts that way, the darkest states would always be California and Texas. Instead we want to color states by how many tweets in the list were there, compared to the population of the state. You should the `us_state_pop` dictionary, which has the same keys (two-letter abbreviations) and has as its values the population of each state. Update your output dictionary by dividing each number of tweets by the population of that state. This gives you a per-capita number of tweets.

Finally, we need to normalize these numbers. We want to always color the state with the most per capita tweets dark blue, with all other states colored accordingly. That means we need to "normalize" the values, scaling them so that they are on some predictable scale. You should find the maximum value that any state has and divide all states by that value. That will mean that the state with the most per capita tweets will have value 1, and all other states will have values between 0 and 1, with a value of .5 meaning that it has half as many tweets as the maximal state, etc. The dictionary with these values is the final output of the function, and those values are used to color the map.

Exercise 5: `Tweets about Canada`

Before counting the queries for each state, the program must filter only the tweets we are interested in (say, with a given word in them). It does this using a "query" function, which takes as input a string, the text of a tweet, and returns `True` or `False`. The program filters out tweets that cause the query to output `False`, and `count_tweets_by_state` will be run on a list of only those tweets that cause the query to return `True`. You should implement `canada_query`, which returns `True` if the text contains the word "canada". This function should ignore the case of both the tweet text and the word "canada". (For example, a tweet that contains "CANada" should be accepted.) In order to do this, you might need to look up some simple operations on strings in the Python documentation. If this is done correctly, you should now be able to run the line `draw_map_for_query(canada_query)` at the end of the program and create the map below, which unsurprisingly shows that in general people who live near the Canadian border tweet about Canada more often.

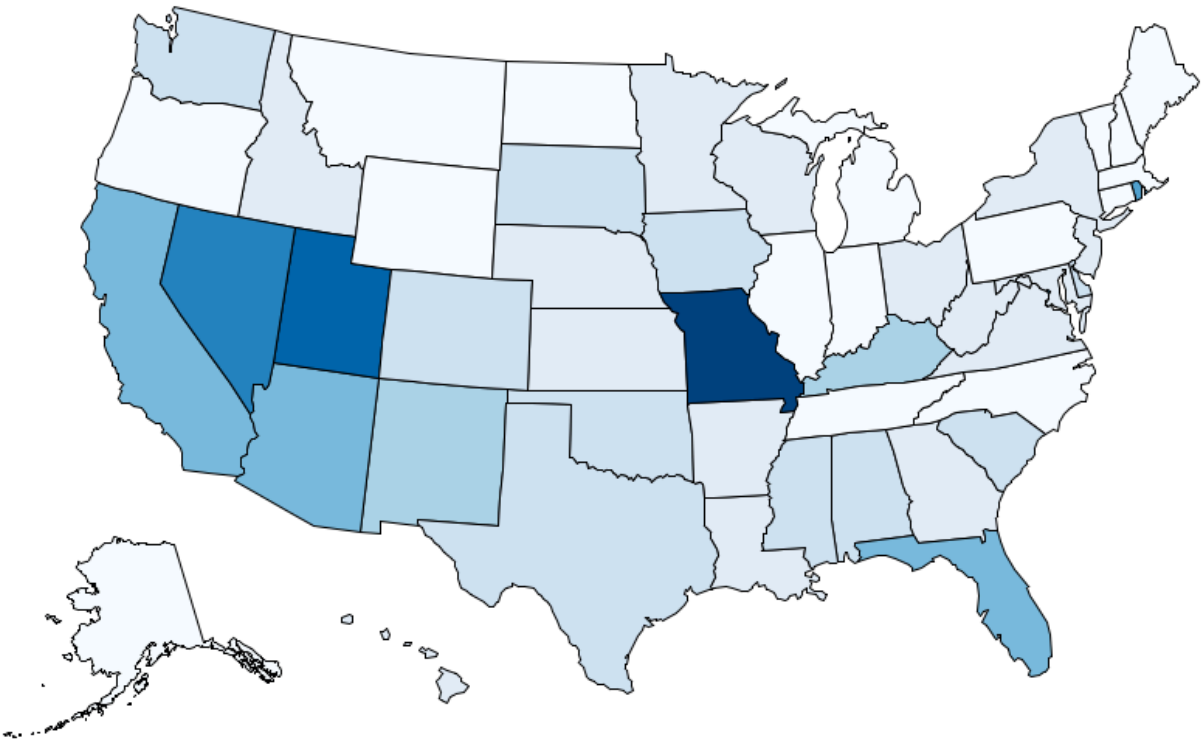


Exercise 6: `make_searcher`

It's not just "canada" that we care about. Searching for tweets with a particular string of text in them is a very reasonable type of query. To facilitate this, implement `make_searcher`, which takes as input a string `term` and outputs a function that will return `True` on strings that have `term` as a substring. For example, `draw_map_for_query(make_searcher("canada"))` should return the same map you got for `canada_query`, and `draw_map_for_query(make_searcher("beach"))` should return the map shown at the beginning of this assignment.

Exercise 7: `mexico_query`

Sometimes we need more complicated queries. We could try to do exactly the same thing as before, but about Mexico rather than Canada. You should try this using `make_searcher`. What you will find is that there are so many tweets about New Mexico that tweets about Mexico are drowned out. What we need is a query function that accepts tweets that include "mexico" but *not* tweets that also include "new". You should implement this as `mexico_query`. Using this query should give you the following map.



Exercise 8: Explore

Play around with other queries. You should find two queries not mentioned here that give interesting results. Print out those maps, along with a short explanation of what query created them (and maybe why you find them interesting). In order to print the maps, you will need to use the print-screen functionality of your operating system and copy them to a file. The program does not support printing maps directly.

Submit

The printed maps from exercise 8 should be turned in (stapled!) in class. You should also upload your edited trends.py file to the relevant problem on the homework submission website. (You don't need to upload the other files.) *You can only submit this file once.*

Congrats! You're done.

Acknowledgements

This project is a modified version of a project originally developed by Aditi Muralidharan, John DeNero, and others at UC Berkeley.