Homework 9: SQL and APIs

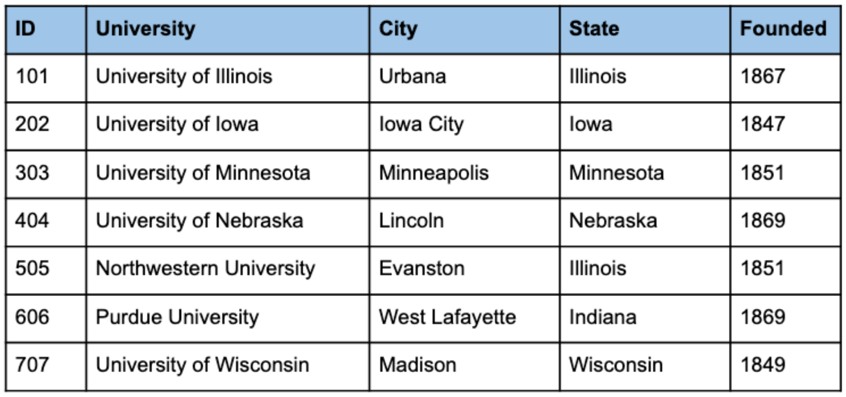
Due April 19, 11:59 pm

Worth 25 points

Instructions on writing and submitting your homework can be found on the course webpage at `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2024/STAT606/hw_instructions.html`. *Failure to follow these instructions will result in lost points.* Please direct any questions the instructor.

1 Warmup: `sqlite3` (3 points)

Here is a table similar to the ones that we saw in the lecture slides, describing some information about the colleges in the West Division of the Big 10 conference. [1]



ID	University	City	State	Founded
101	University of Illinois	Urbana	Illinois	1867
202	University of Iowa	Iowa City	Iowa	1847
303	University of Minnesota	Minneapolis	Minnesota	1851
404	University of Nebraska	Lincoln	Nebraska	1869
505	Northwestern University	Evanston	Illinois	1851
606	Purdue University	West Lafayette	Indiana	1869
707	University of Wisconsin	Madison	Wisconsin	1849

1. Use `sqlite3` to create a database with a single table (in addition to the standard metainformation tables) called `t_big10west` that recreates the table in the figure above. That is, `t_big10west` should have five columns (`ID`, `University`, `City`, `State` and `Founded`), and seven rows corresponding to the seven universities in the table. Save the database in a file called `big10.db`, and include this file in your submission.

2. Oops! There's a typo in that table. The University of Wisconsin was founded in 1848, not 1849. Write a SQL command to correct the corresponding entry of the table, and save it in a string-valued variable called `big10_correction`. You **do not** need to run this command (but you probably should, to check that it's correct, and if you do, and you change the entry in the table in `big10.db`, that's okay).

[1] `https://en.wikipedia.org/wiki/Big_Ten_Conference`

2 Relational Databases and SQL (7 points)

In this problem, you'll interact with a toy SQL database using Python's built-in `sqlite3` package. Documentation can be found at `https://docs.python.org/3/library/sqlite3.html`. For this problem, we'll use a popular toy SQLite database, called Chinook, which represents a digital music collection. See the documentation at

`https://github.com/lerocha/chinook-database/blob/master/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite`

for a more detailed explanation. We'll use the `.sqlite` file `Chinook_Sqlite.sqlite`, which you should download from the GitHub page above. **Note:** Don't forget to save the file in the directory that you're going to compress and hand in, and make sure that you use a relative path when referring to the file, so that when the grading script runs your code on one of our machines the file path will still work!

1. Load the database using the Python `sqlite3` package. How many tables are in the database? Save the answer in the variable `n_tables`.
2. What are the names of the tables in the database? Save the answer as a list of strings, `table_names`. **Note:** you should write Python `sqlite3` code to answer this; don't just look up the answer in the documentation!
3. Write a function `list_album_ids_by_letter` that takes as an argument a single character (i.e., a string of length one) and returns a list of the primary keys of all the albums whose titles start with that character. Your function should ignore case, so that the inputs "a" and "A" yield the same results. Include error checking that raises an appropriate error in the event that the input is of the wrong type or if it is not a single character.
4. Write a function `list_song_ids_by_album_letter` that takes as an argument a single character and returns a list of the primary keys of all the songs whose album names begin with that letter (again ignoring case). As in `list_album_ids_by_letter`, your function should ignore case and perform error checking as appropriate. **Hint:** you'll need a JOIN statement here. You can use the `cursor.description` attribute to find out about tables and the names of their columns.
5. Write a function `total_cost_by_album_letter` that takes as an argument a single character and returns the total cost of buying all the songs whose album begins with that letter. This cost should be based on the tracks' unit prices, so that the cost of buying a set of tracks is simply the sum of the unit prices of all the tracks in the set. Again your function should ignore case and perform appropriate error checking.

3 Warmup: interacting with the Yelp API (5 points)

In this problem, you'll get some practice working with the Yelp API, which we already saw in lecture.

1. First, you need to obtain an API key in order to authenticate to the Yelp API. Follow the instructions at

   `https://docs.developer.yelp.com/docs/fusion-authentication` under the section titled "Create an app on Yelp's Developers site". You may fill in whatever information you like in the app information. Note that you will need a Yelp account to create an app, which you need in order to obtain an API key. If you do not feel comfortable doing this, please let me know *promptly* by email.

Once you have filled out your information, you will be given a ClientID and an API Key. This ID and key come with an associated 300 free calls to the Yelp API to use in the month following the day you create your app. You should not need anywhere near these 300 API calls to test your code, but if you do run out of API calls, please let me know promptly.

2. Write a function called `near_msc` that takes three arguments: a string, a non-negative integer and another string, in that order, representing a search string, a distance in meters and a Yelp API key, respectively. `near_msc( s, d, key )` should return a list of strings, representing the Yelp aliases of all of the establishments matching the given search string `s` that are within `d` meters meters of the statistics department (1300 University Ave, Madison WI), using the given API key. The distance argument should default to 1000. You may have the `key` argument default however you want. Note that we are including this optional `key` argument so that when it comes time to test your code, we can swap out your API key for that of the instructor or the grader. It will be most convenient for you to have this argument default to your API key, but be sure to change this behavior before submitting the assignment if you do not wish to share your API key with the instructor and grader (we will use our own keys to test your code, anyway, of course). Your function should perform error checking to ensure that the arguments are of the right type, and you should raise an appropriate error in the event that the distance argument is negative. **Hint:** you have my permission to modify the code from the slides, which already essentially carries out this operation. **Second hint:** see the documentation at

   `https://docs.developer.yelp.com/reference/v3_business_search` .

3. Write a function called `best_near_msc` that has the same signature as `near_msc` (i.e., takes the same arguments and has the same default behavior) and returns a string representing the alias of the highest-rated establishment matching the given search string and within the given distance of the statistics department. If no businesses exist inside the given distance, your function should return `None`. Note that the ratings of the businesses are rounded to the nearest half star, so you will likely have ties, which you may break arbitrarily. **Hint:** it will be easiest to retrieve some search results and look at the attributes of the resulting JSON objects. You're looking for an attribute that corresponds to a rating.

4 Tracking Asteroids with NASA's NeoWs API (10 points)

In this problem, you'll get more practice working with APIs, this time using one maintained by NASA for retrieving information about near earth objects (NEOs), asteroids that pass close to Earth. The documentation is available at `https://api.nasa.gov/` (scroll down to the API titled *Asteroids NeoWs*).

1. First and foremost, you'll need an API key for accessing the service. You can get one at `https://api.nasa.gov/`. You'll need to supply an email address, which can be either your Wisconsin email or a personal email address.

2. We'll use the Asteroids NeoWs Feed to retrieve Near Earth Objects based on the date of their closest approach to Earth. This can be done using the Feed service. If you read the documentation, you'll see that the Feed API is accessible at

   `https://api.nasa.gov/neo/rest/v1/feed`

   and takes three URL parameters: `start_date`, `end_date` and `api_key`. This last parameter just specifies the API key that you requested previously. `start_date` and `end_date` specify the start and end of a date range, both formatted as `YYYY-MM-DD`.

   Retrieve a JSON object from the NASA NeoWs Feed API for January 1st, 2015 (i.e., set `start_date` and `end_date` to be `'2015-01-01'`). You'll notice that the JSON object has three attributes:

   - `element_count`: the number of near earth objects that had their nearest approach during the time spanned by `start_date` and `end_date`.
   - `near_earth_objects`: a JSON object whose attributes are the dates (represented by strings of the form `YYYY-MM-DD`) in the time spanned by `start_date` and `end_date`. Each such date attribute has as its value an array of JSON objects, each of which represents a near Earth object.
   - `links`: URLs pointing to the "current" day, and the days before and after

   The JSON object for January 1st, 2015 should have `element_count` attribute equal to 14. That is, if your JSON object is stored in `neo_json`, evaluating

   `neo_json['element_count']`

   should return 14. JSON objects representing the NEOs are stored in the array

   `neo_json['near_earth_objects']['2015-01-01']` .

   If you pick out one of the JSON objects in this array, it should have attributes that include strings like

   `'estimated_diameter'` and `'is_potentially_hazardous_asteroid'`.

   Extract the names of all of these attributes and store them in a Python list called `neo_attrs`.

3. We're going to write a function to retrieve all the near Earth objects from a particular day. To start, write a function called `get_neos_response` that takes three positive integers, `yyyy`, `mm` and `dd`, and a string `key`, in that order. `yyyy`, `mm` and `dd` will encode a year, a month and a date, respectively, and the string `key` will be an API key. `get_neos_response` should return the JSON object that is returned by the NASA NeoWs Feed for the specified day. `yyyy` should be a required argument, but `mm` and `dd` should be optional, with both defaulting to 1. The `key` argument should also be optional, and you may have it default however you like (see our discussion in Problem 1 regarding leaving your API key in the code). You may assume that

the first three arguments are all of the appropriate type (i.e., integers), that they are all positive and that `yyyy-mm-dd` specifies a valid date. That is, the grader script will not try to do something funny like get the objects from October 32 by calling `get_neos_response(2023,10,32)` or get a day in the year 0 or -70, or get the objects from February 29 in a non-leap year. You also do not need to worry about the fact that this data set only goes goes back to about 1900. **Hint:** Break the problem down into simple steps:

(a) Use the supplied year, month and day to construct the `start_date` and `end_date` string arguments (which should be the same!). **Hint:** you may want to write a helper function to do this.

(b) Use `start_date` and `end_date` along with the API key to create a dictionary of URL parameters, and use the Python `requests` module to submit an HTTP GET request to the NASA NeoWs Feed API.

(c) Extract the JSON object stored in the resulting `requests` object and return it.

Note: your function should return a *JSON object* (i.e., a Python dictionary), not its string representation. The JSON object your return should have attributes `'links'`, `'near_earth_objects'` and `'element_count'`.

4. So we're in the process of writing a function to retrieve the near Earth objects from a given day. Before moving on, though, it will be useful to have a function for checking that the year, month and date supplied by the user actually encode a real date. Write a function `is_valid_date` that takes three integer arguments, `yyyy`, `mm` and `dd`, in that order, and returns a Boolean that is true if and only if the arguments specify a valid date (that is, a date that actually happens in the Gregorian calendar). For example, `yyyy=1900,mm=13,dd=2` is invalid, because there is no thirteenth month. `yyyy=2020,mm=10,dd=32` is invalid because there is no October 32nd. `yyyy=2020,mm=0,dd=1` is invalid because there is no month zero.[2] Your function should return `False` if any of the arguments are not positive. You need not perform any error checking in this function. That is, there is no need to check that the arguments are integers– we will do that "upstream" in our code, before passing arguments into `is_valid_date`. **Hint:** you may find it useful to create a dictionary that maps the numbers 1 through 12 to the number of days in the corresponding months (i.e., 1 maps to 31, the number of days in January, 2 maps to 28, the number of days in February during a non-leap year, 3 maps to 31, the number of days in March, etc.).

5. Our function `get_neos_response` gets us a JSON object, but we're really just interested in the asteroids, not the extra information included in the JSON object. Write a function `get_neos` that has the same signature as `get_neos_response` (i.e., takes the same arguments and has the same default behavior), in which the three integer arguments specify a date and the key argument specifies an API key, and returns a list of JSON objects that represent the near Earth objects that made their closest approach on the specified date. Your function should check that the arguments are of the appropriate type and that they are all positive, and raise an appropriate error if they are not. Your function should use `is_valid_date` to check that the arguments

[2] Leap years in the Gregorian calendar are those that are divisible by 4 but not by 100, except if they are divisible by 400. So 2016 and 2020 were leap years, 1900 and 1990 were not, but 2000 was. For more information, see `https://en.wikipedia.org/wiki/Leap_year`

jointly describe a valid date and raise an appropriate error if they do not. You do not need to worry about the fact that this data set only goes back to about 1900. You should raise an error if any of the three integer arguments are not positive, but otherwise, so long as the arguments specify a valid date, there is no need to raise an error, even though the specified date might not have any data, or might even be a day in the future! Indeed, if you write this code in a reasonable way, it will happen automatically that if the user specifies a valid date that has no data associated to it, your code will just return an empty list. **Hint:** once again, you may find it helpful to break the problem down into simpler steps:

(a) Perform error checking.

(b) Use `get_neos_response` to get the JSON object for the given date from the NASA API.

(c) Extract the array of near Earth objects.

(d) Return that array as a Python list.

6. Each near Earth object that we get from the API has a number of attributes describing the asteroid. Among these is the `'estimated_diameter'` attribute, whose value is another JSON object that gives the maximum and minimum (estimated) diameter of the asteroid in several different units (e.g., miles, kilometers, etc.). Write a function `get_neos_avg_maxdiam_km` that has the same signature as `get_neos_response` and `get_neos` and the same default values, and returns the average maximum diameter *in kilometers* of all the near Earth objects that made their closest approach on the given day. If no near earth objects made their nearest approach on the given day, your function should return `None`. Your function should perform error checking as described in `get_neos`, but if you're careful, you won't need to write any error checking in this function— `get_neos` already does error checking for us!

7. One thing we might like to explore is how the number of near Earth objects per day changes from day to day. Is this number correlated from one day to the next? Does it vary seasonally (i.e., does it have a time-varying component)? Write a function `count_neos` that has the same signature as `get_neos_response` and `get_neos` and returns a nonnegative integer corresponding to the number of near Earth objects that made their closest approach on the given day. Your function should perform error checking as described in `get_neos`, but once again, you should be able to rely on `get_neos` to do that for you.

8. **Bonus (not worth any points, just bragging rights):** Use your new-found knowledge of this API to find the object discussed in this news story from last year (and which was the inspiration for this homework problem), noting that the API does indeed support retrieving closest approach data for dates in the future: `https://www.npr.org/2021/03/27/981917655`