APIs and SQL

This notebook inclues adapted content from [Melanie Walsh's chapter on Data Collection (https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/00-Data-Collection.html)](https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/00-Data-Collection.html).

In this lab, we'll introduce a useful way to extract data from online, as well as a canonical tool used to explore large datasets when you don't have access to a Python environment. We'll go over the following topics:

- Accessing an API
- API Wrappers
- SQL and SQLite
- pandasql

APIs

It seems only natural that we should be able to extract any data from the internet by programmatically logging information after "going" to each website you're interested in. (In fact, it is [perfectly legal (https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/01-User-Ethics-Legal-Concerns.html)](https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/01-User-Ethics-Legal-Concerns.html).) One way to do this is using [web scraping (https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/02-Web-Scraping-Part1.html)](https://melaniewalsh.github.io/Intro-Cultural-Analytics/04-Data-Collection/02-Web-Scraping-Part1.html), where you write an algorithm which parses website content, logs data, and loops through several HTML web pages. But, this method is becoming less effective over the years, as websites are becoming far more complex (harder to scrape), and most companies are transitioning to a platform where their data is more easily accessible (and controlled) in an Application Programming Interface (API).

What is an API?

An API allows you to programmatically extract and interact with company data which drives their websites. In this way, social networks, museums, foundations, research labs, applications, and projects can make their data publicly available, allowing for developers to use the data to build applications and tools (e.g., for your phone, computer, or refrigerator) that can be used by the general populous. For example, the reason you can access Google Maps on your phone is because developers used the Google Maps API to build that functionality.

Of course, there are plenty of companies or foundations which will likely never use APIs to store/access their data. In these cases though, you can usually find an API that is *related* to that website, or someone may have built (or, they are building) a third-party API for that purpose. Web scraping should typically be a last resort, so we do not teach it in this class.

Caveat: People typically design their APIs such that they decide exactly which kinds of data they want to share. So, they often choose not to share their most lucrative and desirable data. In those cases, you are usually asked to pay some fee.

Using Environment Variables

We will discuss environment variables more in a future lesson, but to use APIs properly, we need to have at least a basic understanding of what environment variables are.

When working on any data science project (e.g., like the web app you'll build later in this course), you will likely track your progress using Git/GitHub. But, keys and secret strings (like the ones we will use to access an API) should never be pushed to GitHub. Instead, it's a best practice to use environment variables when dealing with this kind of data. In short, environment variables are values stored in a special file on your local computer, or on the cloud where your project may be hosted. In this way, those variables are only accessible to agents with access to that file (e.g., your Python interpreter, or the one on the cloud).

In this class, we will use [dotenv (https://github.com/theskumar/python-dotenv#getting-started)](https://github.com/theskumar/python-dotenv#getting-started) to manage environment variables for API. You'll need to `pip` install it, as directed in the instructions, then create a file with the name *.env* (notice the period) in your project directory to hold any keys or secrets. Since we're going to use this package in this notebook, we'll import the library here. *Note: If you're using Git/GitHub, make sure ".env" is added to your [.gitignore file (https://www.atlassian.com/git/tutorials/saving-changes/gitignore)](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)*.

In [1]:
```
from dotenv import load_dotenv
```

Accessing an API

The steps to access *any* API are about the same, no matter the API. So, in this lesson, we're going to use the [Genius (https://genius.com/)](https://genius.com/) API to access data about songs.

Step 1: Client Access Token

Typically, to use an API, you need a special API key usually called a "Client Access Token", which is kind of like a password. Many APIs require authentication keys to gain access to them. To get your necessary Genius API keys, follow these steps:

1. Navigate to the [api-clients page (https://genius.com/api-clients)](https://genius.com/api-clients) (which will prompt you to [sign up for an account (https://genius.com/signup_or_login)](https://genius.com/signup_or_login) if you haven't already). Then, click the button that says **"New API Client"**.
2. Remember, APIs are expecting *developers* to use their APIs to build applications (e.g., for your phone, computers, etc.). But, since we're only doing data analysis for a college course in informatics, we only need to fill in the fields for "App Name" (e.g., "*Song Lyrics Project*"), and "App Website URL" (e.g., "*[https://github.com/leontoddjohnson/i501" (https://github.com/leontoddjohnson/i501")](https://github.com/leontoddjohnson/i501")*). Then, click **Save**.
3. When you click "Save," you'll be given a series of API Keys: a "Client ID" and a "Client Secret." **Copy/Paste these values into your *.env* file** without quotations, as instructed in the dotenv documentation. For example, my *.env* file looks something like this: CLIENT_ID=asdfghjkl;123456789 CLIENT_SECRET=qwertyuiop098765432
4. To generate your "Client Access Token," which is the API key that we'll be using in this notebook, you need to click "Generate Access Token". Place that in your *.env* file as you did

the other variables, maybe under the variable name ACCESS_TOKEN.

We can access our `ACCESS_TOKEN` by using *dotenv* to load our environment variables into the

In [2]:
```
load_dotenv()
```
Out[2]:
```
True
```

In [3]:
```
import os
# do not print this variable anywhere if the notebook is going on GitHub
ACCESS_TOKEN = os.environ['ACCESS_TOKEN']
```

Step 2: Making an API Request

Making an API request is very similar to accessing a URL in your browser. But, instead of getting a rendered HTML web page in return, you get some data in return.

There are a few different ways that we can [query the Genius API (https://docs.genius.com/#songs-h2)](https://docs.genius.com/#songs-h2), but here we'll use [the basic search (https://docs.genius.com/#search-h2)](https://docs.genius.com/#search-h2), which allows you to get a bunch of Genius data about any artist or songs that you search for:

```
http://api.genius.com/search?q={search_term}&access_token=
{client_access_token}
```

First we're going to assign the string "Missy Elliott" to the variable `search_term`. Then we're going to make an f-string URL that contains the variables we'd like to include in our query.

In [4]:
```
search_term = "Missy Elliott"
```

In [5]:
```
genius_search_url = f"http://api.genius.com/search?q={search_term}&access_t
```

You can see the data we'll be requesting from this API by printing the `genius_search_url`, pasting it into your browser.

In [6]:
```
# print(genius_search_url)
```

The data you might see when you navigate to your URL is in [JSON (https://www.w3schools.com/whatis/whatis_json.asp)](https://www.w3schools.com/whatis/whatis_json.asp) format. JSON is an acronym for JavaScript Object Notation, and it is a data format commonly used by APIs. JSON data can be nested, and contains key-value pairs, much like a Python dictionary.

We can access this JSON directly in Python using the [`requests` library (https://requests.readthedocs.io/en/latest/)](https://requests.readthedocs.io/en/latest/) to send HTTP requests to a remote client. If you like, you can read more about what a "request" is [here (https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview), but it suffices to say that it represents an online communication between your computer and the server storing the data you want.

In [7]:
```
import requests
```

In [8]:
```
# here, we make a "GET" request to the Genius server
response = requests.get(genius_search_url)
json_data = response.json()
```

In [9]:
```
json_data
```


```
      'header_image_url': 'https://images.genius.com/d91c82fa4ae2f1016fad
c1c24fbbc59e.1000x333x1.jpg',
      'id': 1529,
      'image_url': 'https://images.genius.com/085828b7d79bf8cf068b1557ca7
a5e4c.1000x1000x1.jpg',
      'is_meme_verified': False,
      'is_verified': False,
      'name': 'Missy Elliott',
      'url': 'https://genius.com/artists/Missy-elliott'}}},
   {'highlights': [],
    'index': 'song',
    'type': 'song',
    'result': {'annotation_count': 14,
     'api_path': '/songs/4640',
     'artist_names': 'Missy Elliott',
     'full_title': 'Get Ur Freak On by\xa0Missy\xa0Elliott',
     'header_image_thumbnail_url': 'https://images.genius.com/8c561e799c6
8d7b4fef60e5d3ef347a9.300x300x1.jpg',
     'header_image_url': 'https://images.genius.com/8c561e799c68d7b4fef60
e5d3ef347a9.1000x1000x1.jpg',
```


In [10]:
```
for i in json_data['response'].keys():
    print(i)
```

```
hits
```

Genius places all of its search results into the "hits" element. By default, it looks like it returns at most 10 search results for any request.

In [11]:
```
for i in json_data['response']['hits']:
    for key in i.keys():
        print(key)
```

```
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
highlights
index
type
result
```

According to the documentation, we can use [referents (https://docs.genius.com/#referents-h2)](https://docs.genius.com/#referents-h2) to increase that number to a maximum of 20 results per request using `per_page`. With this slight adjustment added, let's consolidate our request into a single function to use again later. We'll also add this to our *api_util.py* file for the lab.

In [12]:
```
def genius(search_term, per_page=15):
    '''
    Collect data from the Genius API by searching for `search_term`.

    **Assumes ACCESS_TOKEN is loaded in environment.**
    '''
    genius_search_url = f"http://api.genius.com/search?q={search_term}&" + \
                        f"access_token={ACCESS_TOKEN}&per_page={per_page}"

    response = requests.get(genius_search_url)
    json_data = response.json()

    return json_data['response']['hits']
```

In [13]:
```
json_data = genius("The Beatles")
len(json_data)
```

Out[13]: 15

Loading JSON Data Into a DataFrame

For us to efficiently work with the JSON data, we need to load them into a DataFrame. Using panda's [read_json function (https://pandas.pydata.org/docs/reference/api/pandas.read_json.html)](https://pandas.pydata.org/docs/reference/api/pandas.read_json.html), we can do just that in a pretty efficient way. The only detail is we need the JSON to be in one of the acceptable orientations (see the `orient` argument in the documentation).

In [14]:
```
import pandas as pd
import json
```

In [15]:
```
json_data[0]
```


```
  'header_image_url': 'https://images.genius.com/67d46a92276344c6a8684f9c
7d27ef80.1000x563x1.jpg',
  'id': 2236,
  'lyrics_owner_id': 7,
  'lyrics_state': 'complete',
  'path': '/The-beatles-yesterday-lyrics',
  'pyongs_count': 94,
  'relationships_index_url': 'https://genius.com/The-beatles-yesterday-sa
mple',
  'release_date_components': {'year': 1965, 'month': 9, 'day': 13},
  'release_date_for_display': 'September 13, 1965',
  'release_date_with_abbreviated_month_for_display': 'Sep. 13, 1965',
  'song_art_image_thumbnail_url': 'https://images.genius.com/f9bfd62a8c65
1caab16f631039a9a0b6.300x300x1.jpg',
  'song_art_image_url': 'https://images.genius.com/f9bfd62a8c651caab16f63
1039a9a0b6.600x600x1.jpg',
  'stats': {'unreviewed_annotations': 2,
   'concurrents': 7,
   'hot': False,
```

When we look at any of the hits, we see the data we're interested in is contained in the `"result"` element. We can consolidate all of the "result" elements for each "hit" using a list comprehension. We then use the `json` library to convert this list of JSONs into a single JSON.

Looking ahead: Notice that the `"stats"` and the `"primary_artist"` elements contain *dictionaries* of interesting data that we'll need to unpack once we have our data into a DataFrame.

In [16]:
```
hits = [hit['result'] for hit in json_data]
hits_json = json.dumps(hits)

# load JSON into DataFrame
df = pd.read_json(hits_json)
```

```
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/21141716
18.py:5: FutureWarning: Passing literal json to 'read_json' is deprecated
and will be removed in a future version. To read from a literal string, w
rap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
```

In [17]:
```
df['stats'][0]
```

Out[17]:
```
{'unreviewed_annotations': 2,
 'concurrents': 7,
 'hot': False,
 'pageviews': 2252739}
```

Recall that `"stats"` and `"primary_artist"` contain dictionaries which we want to unpack. After a bit of StackOverflow searching (say), we find that we can [use (https://stackoverflow.com/a/38231651)](https://stackoverflow.com/a/38231651) `pd.apply(pd.Series)` and `pd.concat` to explode these into columns. We'll need to make a slight adjustment to the column names to avoid repeats.

In [18]:
```
df_stats = df['stats'].apply(pd.Series)
df_stats.rename(columns={c:'stat_' + c for c in df_stats.columns},
                inplace=True)

df_primary = df['primary_artist'].apply(pd.Series)
df_primary.rename(columns={c:'primary_artist_' + c for c in df_primary.colu
                  inplace=True)

df = pd.concat((df, df_stats, df_primary), axis=1)
```

```
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/22698097
98.py:1: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/22698097
98.py:5: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
```

In [19]:
```
df[['stat_unreviewed_annotations', 'stat_hot', 'stat_pageviews', 'stat_conc
```

Out[19]:

	stat_unreviewed_annotations	stat_hot	stat_pageviews	stat_concurrents
0	2	False	2252739	7.0
1	1	False	1695402	2.0
2	3	False	1266540	2.0
3	4	False	1251257	NaN
4	5	False	1139975	NaN
5	3	False	1049868	NaN
6	1	False	945703	2.0
7	2	False	848390	5.0
8	1	False	811654	NaN
9	2	False	786860	2.0
10	2	False	703832	NaN
11	0	False	621365	NaN
12	1	False	549690	NaN
13	2	False	545280	NaN
14	0	False	528124	2.0

In [20]:
```
df
```

Out[20]:

	annotation_count	api_path	artist_names	full_title	header_image_
0	6	/songs/2236	The Beatles	Yesterday by The Beatles	https://images.genius.com/67d46a922
1	9	/songs/1575	The Beatles	Let It Be by The Beatles	https://images.genius.com/92f06c735
2	23	/songs/82381	The Beatles	Hey Jude by The Beatles	https://images.genius.com/d3ed7c6e7
3	17	/songs/56218	The Beatles	Come Together by The Beatles	https://images.genius.com/5a6f82f01
4	8	/songs/87577	The Beatles	Here Comes the Sun by The Beatles	https://images.genius.com/003c2b3d4
5	7	/songs/87564	The Beatles	Something by The Beatles	https://images.genius.com/584344512a
6	12	/songs/1577	The Beatles	Eleanor Rigby by The Beatles	https://images.genius.com/d09a9e0db
7	7	/songs/71861	The Beatles	In My Life by The Beatles	https://images.genius.com/1a5e91831
8	18	/songs/1436	The Beatles	A Day in the Life by The Beatles	https://images.genius.com/0123ecd81
9	6	/songs/1556	The Beatles	Blackbird by The Beatles	https://images.genius.com/85f5a0ea
10	25	/songs/56245	The Beatles	I Am the Walrus by The Beatles	https://images.genius.com/8254e742e
11	13	/songs/75670	The Beatles	While My Guitar Gently Weeps by The Beatles	https://images.genius.com/85f5a0ea

	annotation_count	api_path	artist_names	full_title	header_image
12	16	/songs/68179	The Beatles	Strawberry Fields Forever by The Beatles	https://images.genius.com/f2d185ee
13	13	/songs/123444	The Beatles	Across the Universe by The Beatles	https://images.genius.com/3303a7899
14	10	/songs/71029	The Beatles	With a Little Help from My Friends by The Beatles	https://images.genius.com/0123ecd81

15 rows × 35 columns

Collecting Multiple API Calls

We are going to want to perform analysis on more than one artist, so let's use what we've written above to collect data from multiple API calls by looping through multiple search terms. When we loop through each search term, we use the [tqdm package (https://pypi.org/project/tqdm/)](https://pypi.org/project/tqdm/) to help us visualize our progress (you may need to use `pip` to install it). This kind of thing is helpful when we're running multiple API calls, and we don't know how long it will take.

In [21]:
```
from tqdm import tqdm
```

In [22]:
```
search_terms = ['The Beatles', 'Missy Elliot', 'Andy Shauf', 'Slowdive', 'M
n_results_per_term = 10

dfs = []

# loop through search_terms in question
for search_term in tqdm(search_terms):
    json_data = genius(search_term, per_page=n_results_per_term)
    hits = [hit['result'] for hit in json_data]
    hits_json = json.dumps(hits)

    # load JSON into DataFrame
    df = pd.read_json(hits_json)

    # expand dictionary elements
    df_stats = df['stats'].apply(pd.Series)
    df_stats.rename(columns={c:'stat_' + c for c in df_stats.columns},
                    inplace=True)

    df_primary = df['primary_artist'].apply(pd.Series)
    df_primary.rename(columns={c:'primary_artist_' + c for c in df_primary.
                      inplace=True)

    df = pd.concat((df, df_stats, df_primary), axis=1)

    # add to list of DataFrames
    dfs.append(df)
```

```
  0%|                                                     | 0/5 [00:00<?,
?it/s]/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37
02969109.py:13: FutureWarning: Passing literal json to 'read_json' is dep
recated and will be removed in a future version. To read from a literal s
tring, wrap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:16: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:20: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
 20%|█████████                                    | 1/5 [00:00<00:03,  1.
02it/s]/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/3
702969109.py:13: FutureWarning: Passing literal json to 'read_json' is de
precated and will be removed in a future version. To read from a literal
string, wrap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:16: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:20: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
 40%|██████████████████                           | 2/5 [00:01<00:02,  1.
13it/s]/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/3
702969109.py:13: FutureWarning: Passing literal json to 'read_json' is de
precated and will be removed in a future version. To read from a literal
string, wrap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:16: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:20: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
 60%|███████████████████████████                  | 3/5 [00:02<00:01,  1.
17it/s]/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/3
702969109.py:13: FutureWarning: Passing literal json to 'read_json' is de
precated and will be removed in a future version. To read from a literal
string, wrap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:16: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
```

```
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:20: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
 80%|████████████████████████████████████          | 4/5 [00:03<00:00,  1.
22it/s]/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/3
702969109.py:13: FutureWarning: Passing literal json to 'read_json' is de
precated and will be removed in a future version. To read from a literal
string, wrap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:16: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/37029691
09.py:20: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
100%|██████████████████████████████████████████████| 5/5 [00:04<00:00,  1.
20it/s]
```

In [23]:
```
df_genius = pd.concat(dfs)
```

In [24]:
```
df_genius.shape
```

Out[24]:
```
(50, 35)
```

In [25]:
```
df_genius.sample(3)
```

Out[25]:

	annotation_count	api_path	artist_names	full_title	header_image_
4	7	/songs/2885794	Andy Shauf	Martha Sways by Andy Shauf	https://images.genius.com/0d031f033
9	19	/songs/33191	Missy Elliott	Bomb Intro/Pass That Dutch by Missy Elliott	https://images.genius.com/8c2a5015c
7	3	/songs/1136681	Slowdive	Sleep by Slowdive	https://images.genius.com/42a2ba80c

3 rows × 35 columns

Of course, we'll want to copy this function into (or build it *within* our IDE in) our .py file.

Using an API Wrapper

More often than not, someone has built an "API Wrapper" for the API you are working with. An API wrapper makes an API easier to use, and it often extends the API itself. It will typically consist of classes and functions similar to the ones we've built above, but spanning a wide range of functionality and access to the API. For example, John Miller's [LyricsGenius (https://github.com/johnwmillr/LyricsGenius)](https://github.com/johnwmillr/LyricsGenius) gives us an almost universal access to the Genius website, and it even uses web scraping to collect song lyrics themselves.

If ever you're working with an API, do some Googling to make sure there isn't a wrapper you can use to make things easier on you!

First, we'll [install LyricsGenius (https://github.com/johnwmillr/LyricsGenius#installation)](https://github.com/johnwmillr/LyricsGenius#installation) *(in 2023, there is no conda option, so we would need to use `pip`)*, then import it.

In [26]:
```
import lyricsgenius
```

In [27]:
```
# creating an "API Class" is typical for API wrappers
LyricsGenius = lyricsgenius.Genius(ACCESS_TOKEN)
```

To get the top songs and song lyrics from a specific artist you can use the method `.search_artist()`:

In [28]:
```
artist = LyricsGenius.search_artist("Missy Elliott", max_songs=2)
```

```
Searching for songs by Missy Elliott...

Song 1: "Work It"
Song 2: "WTF (Where They From)"

Reached user-specified song limit (2).
Done. Found 2 songs.
```

In [29]:
```
print(artist.songs[0].lyrics[:300])
```

```
[Intro]
DJ, please pick up your phone, I'm on the request line
This is a Missy Elliott one-time exclusive (Come on)

[Chorus]
Is it worth it? Let me work it
I put my thing down, flip it and reverse it
Ti esrever dna ti pilf, nwod gniht ym tup
Ti esrever dna ti pilf, nwod gniht ym tup
If you got a bi
```

You'll notice this function took *much* longer than our function above. If you take a quick glance at the documentation for the function (use Shift+Tab in the parentheses next to the function), you'll see that the `get_full_info=True` argument slows down the search (likely because it includes scraping lyrics). If we were using the lyrics in our investigation, we might be okay with this delay, but since we're only interested in numerical data for the time being (and because setting

SQL

Section Prerequisites: [SQLBolt (https://sqlbolt.com/)](https://sqlbolt.com/) lessons 1-6 (and, 7-12 if possible).

[Structured Query Language (SQL) (https://www.sqltutorial.org/what-is-sql/)](https://www.sqltutorial.org/what-is-sql/) is a programming language designed to allow users to **query** databases containing multiple tables (rows and columns) of data, each related to one another using column-to-column relationships. It is easily the most popular language for accessing large tabular databases, so naturally, many companies and users have used the language to create their own variants of the language called [dialects (https://arctype.com/blog/sql-dialects/)](https://arctype.com/blog/sql-dialects/). In this class, we will be using [SQLite (https://www.sqlite.org/index.html)](https://www.sqlite.org/index.html) and its corresponding dialect.

In the job setting, you will typically make SQL queries using a combination of the following:

- Some data browser, with data stored on the cloud or on in-house servers
- Python, likely using a package called SQLAlchemy to connect to a database

In this class, we will use [DB Browser (https://sqlitebrowser.org/about/)](https://sqlitebrowser.org/about/) to simulate the kind of browser you'd use on the job, and SQLAlchemy to gain practice with the tool. We'll also see how you can use the [pandasql package (https://pypi.org/project/pandasql/)](https://pypi.org/project/pandasql/) to assimilate SQL queries into the pandas framework.

Set Up

SQLite

Before we can continue, you need to make sure that SQLite is installed on your machine.

- For MacOS users, SQLite should already be installed on your computer. You can test this by running `sqlite3 --version` in your terminal.
- For everyone else, you'll need to [follow these steps to download and install SQLite (https://www.sqlitetutorial.net/download-install-sqlite/)](https://www.sqlitetutorial.net/download-install-sqlite/).

Lastly, for this lab, we're going to use the [SQLite Sample Database (https://www.sqlitetutorial.net/download-install-sqlite/)](https://www.sqlitetutorial.net/download-install-sqlite/). Scroll down to the "Download SQLite sample database" section of the page for the link, or download it directly [here (as of Sep 2023) (https://www.sqlitetutorial.net/wp-content/uploads/2018/03/chinook.zip)](https://www.sqlitetutorial.net/wp-content/uploads/2018/03/chinook.zip). Unzip the file, and move the .db file to a convenient location (e.g., the same place where this lab is saved). *Note: If you are using GitHub, ".db" should typically be added to your [.gitignore file (https://www.atlassian.com/git/tutorials/saving-changes/gitignore)](https://www.atlassian.com/git/tutorials/saving-changes/gitignore).*

SQLAlchemy and pandasql

Lastly, we'll be using [SQLAlchemy (https://www.sqlalchemy.org/)](https://www.sqlalchemy.org/) to connect our Python environment to our database, and [pandasql (https://pypi.org/project/pandasql/)](https://pypi.org/project/pandasql/) to use SQL "in" pandas. You can install them both using anaconda:

```
pip install SQLAlchemy
pip install -U pandasql
```

DB Browser

Next, you'll need to [install DB Browser (https://sqlitebrowser.org/dl/)](https://sqlitebrowser.org/dl/) by following the installation instructions provided for your particular operating system. *Note: For Mac "M1/M2" chips, you'll use the "Apple Silicon" option.*

Exploring in DB Browser

Let's explore the `chinook` database (i.e., the SQLite Sample Database) using DB Browser to "test" queries, and SQLAlchemy (below) to run queries here in the notebook.

1. First, in DB Browser, click the "Open Database" button, then find the *chinook.db* file on your computer. (This is how you would open any SQL ".db" file.)
2. Close the side panels on the right until all you see is the Main window and the handy "DB Schema" viewer on the right. The [SQL Schema (https://www.sqlite.org/schematab.html)](https://www.sqlite.org/schematab.html) provides information on the tables and columns within a database. *Note: you can also view a SQL database schema directly using [PRAGMA commands (https://www.sqlite.org/pragma.html)](https://www.sqlite.org/pragma.html).*
3. Select the "Execute SQL" tab to start writing SQL Queries.

SQLAlchemy

Again, you can use DB Browser to explore your data, but you can also *bring that data into your notebook* using a combination of SQLAlchemy and pandas.

Connect

To connect to a database using SQLAlchemy, we need to define the database location. Using `create_engine` , we create a connection between the SQL database represented in the *.db* file, and our Python instance.

Note: In our case, we have a database immediately accessible on our computer. But in practice, you'll more likely need to [access a remote database (https://docs.sqlalchemy.org/en/20/core/engines.html#custom-dbapi-connect-arguments-on-connect-routines)](https://docs.sqlalchemy.org/en/20/core/engines.html#custom-dbapi-connect-arguments-on-connect-routines) requiring credentials.

In [30]:
```
import os  # if you haven't already, above

from sqlalchemy import inspect, create_engine
import pandas as pd
```

For this lab, the database is stored in a *data* folder inside the same directory as this notebook. This notebook has a "working directory" or file path associated with it, which can be used by Python to "navigate" to the same location. Using the same `os` library from above, we can use `os.getcwd()` to get the **current working directory**, and use it to navigate to the *chinook* database.

In [31]:
```
cwd = os.getcwd()
db_path = cwd + "/data/chinook.db"  # complete path to the database file
```

In [32]:
```
# the "engine" is a connection between Python and the database
engine = create_engine(f"sqlite:////{db_path}")
```

In [33]:
```
# this is one way to access an aspect of the schema
insp = inspect(engine)
```

In [34]:
```
insp.get_table_names()
```

Out[34]:
```
['albums',
 'artists',
 'customers',
 'employees',
 'genres',
 'invoice_items',
 'invoices',
 'media_types',
 'playlist_track',
 'playlists',
 'tracks']
```

Load into pandas

Once you have a connection between Python and the database, we can use `pd.read_sql()` to load the result of SQL queries into pandas.

In [35]:
```
query = \
'''
SELECT
    DISTINCT(city)
FROM employees;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[35]:

	City
0	Edmonton
1	Calgary
2	Lethbridge

SQL Statements

This section contains code examples from the [SQLite Tutorial Website (https://www.sqlitetutorial.net/)](https://www.sqlitetutorial.net/). Refer to this website for more in-depth explanations.

We interact with SQL using a "query", or the code/interface between the user and the database. It contains keywords, column names, tables names, and even function operations. In this notebook, we will introduce a couple of examples of some common query statements, and then follow up with a few more methods SQL provides.

*Note: **SQL code is case-insensitive**, but I find it to be a good practice to capitalize keywords (e.g., "SELECT"), and lower the case of column names (e.g., "employees", above) when using SQL. I also tend to use different lines and indenting wherever possible to keep the code clean.*

[SELECT (https://www.sqlitetutorial.net/sqlite-select/)](https://www.sqlitetutorial.net/sqlite-select/)

The foundation of virtually all SQL queries is the `SELECT` statement. Typically, this is followed (at some point) by a `FROM`, denoting (naturally) where we are selecting our data from. `DISTINCT` removes duplicate rows of a column.

In [36]:

```
query = \
'''
SELECT
    DISTINCT city
FROM employees;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[36]:

	City
0	Edmonton
1	Calgary
2	Lethbridge

[LIMIT (https://www.sqlitetutorial.net/sqlite-limit/)](https://www.sqlitetutorial.net/sqlite-limit/)

`LIMIT` only returns the first set of rows for a result, very much like `head()` .

In [37]:
```
query = \
'''
SELECT
    name,
    composer,
    milliseconds,
    unitprice
FROM tracks
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[37]:

	Name	Composer	Milliseconds	UnitPrice
0	For Those About To Rock (We Salute You)	Angus Young, Malcolm Young, Brian Johnson	343719	0.99
1	Balls to the Wall	None	342562	0.99
2	Fast As a Shark	F. Baltes, S. Kaufman, U. Dirkscneider & W. Ho...	230619	0.99
3	Restless and Wild	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...	252051	0.99
4	Princess of the Dawn	Deaffy & R.A. Smith-Diesel	375418	0.99
5	Put The Finger On You	Angus Young, Malcolm Young, Brian Johnson	205662	0.99
6	Let's Get It Up	Angus Young, Malcolm Young, Brian Johnson	233926	0.99
7	Inject The Venom	Angus Young, Malcolm Young, Brian Johnson	210834	0.99
8	Snowballed	Angus Young, Malcolm Young, Brian Johnson	203102	0.99
9	Evil Walks	Angus Young, Malcolm Young, Brian Johnson	263497	0.99

In [38]:
```
query = \
'''
SELECT
    name,
    composer,
    unitprice
FROM tracks
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[38]:

	Name	Composer	UnitPrice
0	For Those About To Rock (We Salute You)	Angus Young, Malcolm Young, Brian Johnson	0.99
1	Balls to the Wall	None	0.99
2	Fast As a Shark	F. Baltes, S. Kaufman, U. Dirkscneider & W. Ho...	0.99
3	Restless and Wild	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...	0.99
4	Princess of the Dawn	Deaffy & R.A. Smith-Diesel	0.99
5	Put The Finger On You	Angus Young, Malcolm Young, Brian Johnson	0.99
6	Let's Get It Up	Angus Young, Malcolm Young, Brian Johnson	0.99
7	Inject The Venom	Angus Young, Malcolm Young, Brian Johnson	0.99
8	Snowballed	Angus Young, Malcolm Young, Brian Johnson	0.99
9	Evil Walks	Angus Young, Malcolm Young, Brian Johnson	0.99

ORDER BY (https://www.sqlitetutorial.net/sqlite-order-by/)

We can also order our data based on some column (akin to "sort", in pandas). Sort will default to ascending order, but it's a good practice to include `ASC` or `DESC` as needed.

In [39]:
```
query = \
'''
SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks
ORDER BY
    albumid ASC,
    milliseconds DESC;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[39]:

	Name	Milliseconds	AlbumId
0	For Those About To Rock (We Salute You)	343719	1
1	Spellbound	270863	1
2	Evil Walks	263497	1
3	Breaking The Rules	263288	1
4	Let's Get It Up	233926	1
...	...	...	...
3498	Pini Di Roma (Pinien Von Rom) \ I Pini Della V...	286741	343
3499	String Quartet No. 12 in C Minor, D. 703 "Quar...	139200	344
3500	L'orfeo, Act 3, Sinfonia (Orchestra)	66639	345
3501	Quintet for Horn, Violin, 2 Violas, and Cello ...	221331	346
3502	Koyaanisqatsi	206005	347

3503 rows × 3 columns

Note: the column you use to sort your data does not need to be included in the SELECT statement.

In [40]:
```
query = \
'''
SELECT
    name,
    composer,
    albumid
FROM
    tracks
ORDER BY
    milliseconds DESC
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[40]:

	Name	Composer	AlbumId
0	Occupation / Precipice	None	227
1	Through a Looking Glass	None	229
2	Greetings from Earth, Pt. 1	None	253
3	The Man With Nine Lives	None	253
4	Battlestar Galactica, Pt. 2	None	253
5	Battlestar Galactica, Pt. 1	None	253
6	Murder On the Rising Star	None	253
7	Battlestar Galactica, Pt. 3	None	253
8	Take the Celestra	None	253
9	Fire In Space	None	253

Using `LIMIT` in tandem with `ORDER BY` helps us extract the `n` th item (highest or lowest), ordered by some column.

In [41]:
```
# second longest track
query = \
'''
SELECT
    trackid,
    name,
    milliseconds
FROM
    tracks
ORDER BY
    milliseconds DESC
LIMIT 1 OFFSET 2;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[41]:

	TrackId	Name	Milliseconds
0	3244	Greetings from Earth, Pt. 1	2960293

WHERE (https://www.sqlitetutorial.net/sqlite-where/)

We can *filter* our data using the `WHERE` clause. In the same way that pandas provides logical operations, there are also several available in SQLite (see the section link above for more on these, and this article on "glob" operators (https://www.sqlitetutorial.net/sqlite-glob/)).

In [42]:
```
query = \
'''
SELECT
    name,
    albumid,
    Milliseconds,
    mediatypeid
FROM
    tracks
WHERE
    mediatypeid IN (2, 3)
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[42]:

	Name	AlbumId	Milliseconds	MediaTypeId
0	Balls to the Wall	2	342562	2
1	Fast As a Shark	3	230619	2
2	Restless and Wild	3	252051	2
3	Princess of the Dawn	3	375418	2
4	Welcome to the Jungle	90	273552	2
5	It's So Easy	90	202824	2
6	Nightrain	90	268537	2
7	Out Ta Get Me	90	263893	2
8	Mr. Brownstone	90	228924	2
9	Paradise City	90	406347	2

The `%` wildcard can be handy.

In [43]:
```
query = \
'''
SELECT
    name,
    albumid,
    composer
FROM
    tracks
WHERE
    composer LIKE '%Smith%'
ORDER BY
    albumid;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[43]:

	Name	AlbumId	Composer
0	Restless and Wild	3	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...
1	Princess of the Dawn	3	Deaffy & R.A. Smith-Diesel
2	Killing Floor	19	Adrian Smith
3	Machine Men	19	Adrian Smith
4	2 Minutes To Midnight	95	Adrian Smith/Bruce Dickinson
...	...	...	...
92	Savior	195	Anthony Kiedis/Chad Smith/Flea/John Frusciante
93	Dancing Barefoot	234	Ivan Kral/Patti Smith
94	Take the Box	322	Luke Smith
95	What Is It About Men	322	Delroy "Chris" Cooper, Donovan Jackson, Earl C...
96	Amy Amy Amy (Outro)	322	Astor Campbell, Delroy "Chris" Cooper, Donovan...

97 rows × 3 columns

In [44]:
```
query = \
'''
SELECT
    name,
    milliseconds,
    bytes,
    albumid
FROM
    tracks
WHERE
    albumid = 1
    AND milliseconds > 250000;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[44]:

	Name	Milliseconds	Bytes	AlbumId
0	For Those About To Rock (We Salute You)	343719	11170334	1
1	Evil Walks	263497	8611245	1
2	Breaking The Rules	263288	8596840	1
3	Spellbound	270863	8817038	1

[IS (NOT) NULL (https://www.sqlitetutorial.net/sqlite-is-null/)](https://www.sqlitetutorial.net/sqlite-is-null/)

Of course, we may want to include or exclude missing values in the data. In SQL, "missing" values are encoded as `NULL` .

In [45]:
```
query = \
'''
SELECT
    Name,
    Composer
FROM
    tracks
WHERE
    Composer IS NOT NULL
ORDER BY
    Name;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[45]:

	Name	Composer
0	"40"	U2
1	"Eine Kleine Nachtmusik" Serenade In G, K. 525...	Wolfgang Amadeus Mozart
2	#1 Zero	Cornell, Commerford, Morello, Wilk
3	'Round Midnight	Miles Davis
4	(Anesthesia) Pulling Teeth	Cliff Burton
...	...	...
2520	É Fogo	Mônica Marianno
2521	É Preciso Saber Viver	Erasmo Carlos/Roberto Carlos
2522	É Uma Partida De Futebol	Samuel Rosa
2523	É que Nessa Encarnação Eu Nasci Manga	Lucina/Luli
2524	Último Pau-De-Arara	Corumbá/José Gumarães/Venancio

2525 rows × 2 columns

In [46]:
```
query = \
'''
SELECT
    InvoiceId,
    BillingCity,
    BillingState,
    BillingPostalCode,
    Total
FROM invoices
WHERE
    BillingState IS NULL
    AND BillingPostalCode IS NULL
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[46]:

	InvoiceId	BillingCity	BillingState	BillingPostalCode	Total
0	22	Santiago	None	None	1.98
1	28	Lisbon	None	None	1.98
2	33	Santiago	None	None	13.86
3	51	Lisbon	None	None	3.96
4	73	Lisbon	None	None	5.94
5	88	Santiago	None	None	17.91
6	125	Lisbon	None	None	0.99
7	126	Porto	None	None	1.98
8	149	Porto	None	None	3.96
9	171	Porto	None	None	5.94

JOIN (https://www.sqlitetutorial.net/sqlite-join/)

The SQL JOIN allows us to merge data from multiple tables. It's essentially the same thing as the `pandas.merge` , but the code is a bit more accessible than pandas when it comes to merging many tables together.

One common practice is for databases to have an "entity" table, which contains the ID along with many other attributes of the entity. Then, in a table, one only needs to reference the ID of the entity rather than store redundant data that exists already in another table.

INNER JOIN (https://www.sqlitetutorial.net/sqlite-inner-join/)

The `INNER JOIN` only matches rows where the column value in question exists in **both** tables.

In [47]:
```
query = \
'''
SELECT
    ar.Name artist_name,
    al.Title AS album_title
FROM
    albums al
INNER JOIN artists ar
    ON al.ArtistId = ar.ArtistId
ORDER BY ar.Name
LIMIT 15;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[47]:

	artist_name	album_title
0	AC/DC	For Those About To Rock We Salute You
1	AC/DC	Let There Be Rock
2	Aaron Copland & London Symphony Orchestra	A Copland Celebration, Vol. I
3	Aaron Goldberg	Worlds
4	Academy of St. Martin in the Fields & Sir Nevi...	The World of Classical Favourites
5	Academy of St. Martin in the Fields Chamber En...	Sir Neville Marriner: A Celebration
6	Academy of St. Martin in the Fields, John Birc...	Fauré: Requiem, Ravel: Pavane & Others
7	Academy of St. Martin in the Fields, Sir Nevil...	Bach: Orchestral Suites Nos. 1 - 4
8	Accept	Balls to the Wall
9	Accept	Restless and Wild
10	Adrian Leaper & Doreen de Feis	Górecki: Symphony No. 3
11	Aerosmith	Big Ones
12	Aisha Duo	Quiet Songs
13	Alanis Morissette	Jagged Little Pill
14	Alberto Turco & Nova Schola Gregoriana	Adorate Deum: Gregorian Chant from the Proper ...

A few things to note here:

- the `AS` keyword (or a space) followed by some string or keyword allows us to change the way data is presented in the result of a query, such as for renaming columns or tables.
- Whenever we join multiple tables, it's a good practice to "name" those tables (using the `AS` /space syntax), then reference columns with the `table.column` notation.

[LEFT JOIN (https://www.sqlitetutorial.net/sqlite-left-join/)](https://www.sqlitetutorial.net/sqlite-left-join/)

Similarly, the `LEFT JOIN` collects all rows where the value in question exists in the "left" table,

In [48]:
```
query = \
'''
SELECT
    ar.Name artist_name,
    al.Title album_title,
    ar.ArtistId
FROM
    artists ar
LEFT JOIN albums al ON
    ar.ArtistId = al.ArtistId
WHERE al.Title IS NULL
ORDER BY Name
LIMIT 5;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[48]:

	artist_name	album_title	ArtistId
0	A Cor Do Som	None	43
1	Academy of St. Martin in the Fields, Sir Nevil...	None	239
2	Aerosmith & Sierra Leone's Refugee Allstars	None	161
3	Avril Lavigne	None	166
4	Azymuth	None	26

[CROSS JOIN (https://www.sqlitetutorial.net/sqlite-cross-join/)](https://www.sqlitetutorial.net/sqlite-cross-join/)

The `CROSS JOIN` collects all combinations of values between two columns in a table. This kind of function is handy when you want to calculate something for multiple groups based on all the values that exist.

In [49]:
```
query = \
'''
SELECT *
FROM media_types
CROSS JOIN genres
LIMIT 50;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[49]:

	MediaTypeId	Name	GenreId	Name
0	1	MPEG audio file	1	Rock
1	1	MPEG audio file	2	Jazz
2	1	MPEG audio file	3	Metal
3	1	MPEG audio file	4	Alternative & Punk
4	1	MPEG audio file	5	Rock And Roll
5	1	MPEG audio file	6	Blues
6	1	MPEG audio file	7	Latin
7	1	MPEG audio file	8	Reggae
8	1	MPEG audio file	9	Pop
9	1	MPEG audio file	10	Soundtrack
10	1	MPEG audio file	11	Bossa Nova

[FULL OUTER JOIN (https://www.sqlitetutorial.net/sqlite-full-outer-join/)](https://www.sqlitetutorial.net/sqlite-full-outer-join/)

The `FULL OUTER JOIN` collects the *union* of all rows which have matching columns values between tables.

In [50]:
```
query = \
'''
SELECT
    ar.Name artist_name,
    al.Title album_title
FROM
    artists ar
FULL OUTER JOIN albums al ON
    ar.ArtistId = al.ArtistId
ORDER BY Name
'''

df_result = pd.read_sql(query, engine)
print("Number of rows: ", df_result.shape[0])
df_result
```

```
Number of rows:  418
```

Out[50]:

	artist_name	album_title
0	A Cor Do Som	None
1	AC/DC	For Those About To Rock We Salute You
2	AC/DC	Let There Be Rock
3	Aaron Copland & London Symphony Orchestra	A Copland Celebration, Vol. I
4	Aaron Goldberg	Worlds
...	...	...
413	Xis	None
414	Yehudi Menuhin	Bartok: Violin & Viola Concertos
415	Yo-Yo Ma	Bach: The Cello Suites
416	Youssou N'Dour	None
417	Zeca Pagodinho	Ao Vivo [IMPORT]

418 rows × 2 columns

Note: there are only 275 rows in the `artists` table, and 347 in the `albums` table. We'll see how you can calculate these values shortly!

[SELF JOIN (https://www.sqlitetutorial.net/sqlite-self-join/)](https://www.sqlitetutorial.net/sqlite-self-join/)

The `SELF JOIN` is just a join between a table and itself.

In [51]:
```
query = \
'''
SELECT m.firstname || ' ' || m.lastname AS 'manager',
       e.firstname || ' ' || e.lastname AS 'direct_report'
FROM employees e
INNER JOIN employees m
    ON m.employeeid = e.reportsto
ORDER BY manager;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[51]:

	manager	direct_report
0	Andrew Adams	Nancy Edwards
1	Andrew Adams	Michael Mitchell
2	Michael Mitchell	Robert King
3	Michael Mitchell	Laura Callahan
4	Nancy Edwards	Jane Peacock
5	Nancy Edwards	Margaret Park
6	Nancy Edwards	Steve Johnson

GROUP BY (https://www.sqlitetutorial.net/sqlite-group-by/)

The `GROUP BY` function exists across many data manipulation frameworks (e..g, R, pandas, etc.), and it is meant to break up the data into groups. Typically, once data is broken into groups, continuous values are aggregated to a single value within each group. SQL provides many aggregation functions (https://www.sqlitetutorial.net/sqlite-aggregate-functions/) which can be used with `GROUP BY` .

In [52]:
```
query = \
'''
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid
ORDER BY COUNT(trackid) DESC;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[52]:

	AlbumId	COUNT(trackid)
0	141	57
1	23	34
2	73	30
3	229	26
4	230	25
...	...	...
342	343	1
343	344	1
344	345	1
345	346	1
346	347	1

347 rows × 2 columns

In [53]:
```
query = \
'''
SELECT
    t.albumid AS album_ID,
    a.title AS album_name,
    COUNT(t.trackid) AS num_track_ids
FROM
    tracks t
INNER JOIN albums a
    ON a.albumid = t.albumid
GROUP BY
    t.albumid
ORDER BY
    num_track_ids DESC
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[53]:

	album_ID	album_name	num_track_ids
0	141	Greatest Hits	57
1	23	Minha Historia	34
2	73	Unplugged	30
3	229	Lost, Season 3	26
4	230	Lost, Season 1	25
5	251	The Office, Season 3	25
6	83	My Way: The Best Of Frank Sinatra [Disc 1]	24
7	231	Lost, Season 2	24
8	253	Battlestar Galactica (Classic), Season 1	24
9	24	Afrociberdelia	23

HAVING (https://www.sqlitetutorial.net/sqlite-having/)

The `HAVING` operator is the `WHERE` operator which we can apply *after* the `GROUP BY`. That is, the "Group By Section" of a query has keywords in this order: `WHERE` → `GROUP BY` → `HAVING`.

A good way to remember this is that the word "having" makes more sense if you think about it as applied to *collections* (or groups) of things rather than the things themselves (e.g., "I *have* a handful of marbles", not "I *where* a handful of marbles").

In [54]:

```
query = \
'''
SELECT
   albumid,
   COUNT(trackid)
FROM
   tracks
GROUP BY
   albumid
HAVING
   COUNT(albumid) BETWEEN 18 AND 20
ORDER BY albumid;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[54]:

	AlbumId	COUNT(trackid)
0	21	18
1	37	20
2	54	20
3	55	20
4	72	18
5	102	18
6	115	20
7	145	18
8	146	18
9	202	18
10	211	18
11	213	18
12	221	20
13	227	19
14	248	19
15	258	19

In [55]:
```
query = \
'''
SELECT
    ar.name AS artist_name,
    a.title AS album_name,
    COUNT(trackid) AS num_tracks
FROM
    tracks t
INNER JOIN albums a
    ON t.albumid = a.albumid
LEFT JOIN artists ar
    ON a.ArtistId = ar.ArtistId
WHERE
    artist_name LIKE "%Jam%"
GROUP BY
    a.ArtistId,
    t.albumid
HAVING
    num_tracks > 10
ORDER BY
    artist_name ASC,
    num_tracks DESC
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[55]:

	artist_name	album_name	num_tracks
0	James Brown	Sex Machine	20
1	Jamiroquai	The Return Of The Space Cowboy	11
2	Jamiroquai	Synkronized	11
3	Pearl Jam	Live On Two Legs [Live]	16
4	Pearl Jam	Riot Act	15
5	Pearl Jam	Pearl Jam	13
6	Pearl Jam	Vs.	12
7	Pearl Jam	Ten	11

CASE (https://www.sqlitetutorial.net/sqlite-case/)

The SQL `CASE` statement is the analog for if-then-else operations in Python.

In [56]:
```
query = \
'''
SELECT customerid,
    firstname,
    lastname,
    country,
    CASE country
        WHEN 'USA'
            THEN 'Domestic'
            ELSE 'Foreign'
    END CustomerGroup
FROM
    customers
ORDER BY
    LastName,
    FirstName
LIMIT 20;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[56]:

	CustomerId	FirstName	LastName	Country	CustomerGroup
0	12	Roberto	Almeida	Brazil	Foreign
1	28	Julia	Barnett	USA	Domestic
2	39	Camille	Bernard	France	Foreign
3	18	Michelle	Brooks	USA	Domestic
4	29	Robert	Brown	Canada	Foreign
5	21	Kathy	Chase	USA	Domestic
6	26	Richard	Cunningham	USA	Domestic
7	41	Marc	Dubois	France	Foreign
8	34	João	Fernandes	Portugal	Foreign
9	30	Edward	Francis	Canada	Foreign
10	42	Wyatt	Girard	France	Foreign

In [57]:
```
query = \
'''
SELECT
    trackid,
    name,
    CASE
        WHEN milliseconds < 60000
            THEN 'short'
        WHEN milliseconds > 60000
        AND milliseconds < 300000
            THEN 'medium'
        ELSE
            'long'
        END category
FROM
    tracks
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[57]:

	TrackId	Name	category
0	1	For Those About To Rock (We Salute You)	long
1	2	Balls to the Wall	long
2	3	Fast As a Shark	medium
3	4	Restless and Wild	medium
4	5	Princess of the Dawn	long
5	6	Put The Finger On You	medium
6	7	Let's Get It Up	medium
7	8	Inject The Venom	medium
8	9	Snowballed	medium
9	10	Evil Walks	medium

Subqueries and Views

Sometimes, it's helpful to *use* the result of one query *within* another query. This is typically called a **Subquery** (https://www.sqlitetutorial.net/sqlite-subquery/).

- The (NOT) EXISTS (https://www.sqlitetutorial.net/sqlite-exists/) operator checks whether a subquery returns a result at all.
- If a subquery is overly complex, or if you plan to use it in the future, you can save it as a **view** (https://www.sqlitetutorial.net/sqlite-create-view/) (or, you can delete (https://www.sqlitetutorial.net/sqlite-drop-view/) one you no longer need).

In [58]:
```
query = \
'''
SELECT trackid,
       name,
       albumid
FROM tracks
WHERE albumid = (
   SELECT albumid
   FROM albums
   WHERE title = 'Let There Be Rock'
);
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[58]:

	TrackId	Name	AlbumId
0	15	Go Down	4
1	16	Dog Eat Dog	4
2	17	Let There Be Rock	4
3	18	Bad Boy Boogie	4
4	19	Problem Child	4
5	20	Overdose	4
6	21	Hell Ain't A Bad Place To Be	4
7	22	Whole Lotta Rosie	4

Set Operations

In SQL, there are also set operations [UNION (https://www.sqlitetutorial.net/sqlite-union/)](https://www.sqlitetutorial.net/sqlite-union/), [EXCEPT (https://www.sqlitetutorial.net/sqlite-except/)](https://www.sqlitetutorial.net/sqlite-except/) (i.e., set difference), and [INTERSECT (https://www.sqlitetutorial.net/sqlite-intersect/)](https://www.sqlitetutorial.net/sqlite-intersect/). For each of these, you'd use subqueries to build the query.

EXERCISE

Take a look at these different set operations. Can you build a query which returns a **single column** of the unique album names *and* artist names which contain the word "black"?

In [59]:
```
# your code here
query = \
'''
SELECT al.title || ' by ' || ar.name AS 'Albums Title + Artist Name'
FROM
    albums al
INNER JOIN artists ar
    ON al.ArtistId = ar.ArtistId
WHERE al.title LIKE '%black%'
   OR ar.name LIKE '%black%';
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[59]:

	Albums Title + Artist Name
0	Alcohol Fueled Brewtality Live! [Disc 1] by Bl...
1	Alcohol Fueled Brewtality Live! [Disc 2] by Bl...
2	Black Sabbath by Black Sabbath
3	Black Sabbath Vol. 4 (Remaster) by Black Sabbath
4	Black Album by Metallica
5	[1997] Black Light Syndrome by Terry Bozzio, T...
6	Live [Disc 1] by The Black Crowes
7	Live [Disc 2] by The Black Crowes
8	Back to Black by Amy Winehouse

SQL Functions

It's rare that we are satisfied with the data as it exists within the data table. Typically, we want to transform the data, and present it in a certain way. This is where SQL Functions come in.

Mathematical Operations

SQLite has several different [data types (https://www.sqlitetutorial.net/sqlite-data-types/)](https://www.sqlitetutorial.net/sqlite-data-types/), and sometimes, we'd like to leverage one type over another. Suppose we'd rather show the number of minutes rather than milliseconds. We can use the `CAST` operator to convert our value to a `FLOAT`, or we can divide *by* a float (e.g., `60000.0`) to coerce our data into the more complex `FLOAT` data type.

In [60]:
```
query = \
'''
SELECT
    name,
    albumid,
    CAST(Milliseconds AS FLOAT) / 60000 minutes,
    mediatypeid
FROM
    tracks
WHERE
    mediatypeid IN (2, 3)
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[60]:

	Name	AlbumId	minutes	MediaTypeId
0	Balls to the Wall	2	5.709367	2
1	Fast As a Shark	3	3.843650	2
2	Restless and Wild	3	4.200850	2
3	Princess of the Dawn	3	6.256967	2
4	Welcome to the Jungle	90	4.559200	2
5	It's So Easy	90	3.380400	2
6	Nightrain	90	4.475617	2
7	Out Ta Get Me	90	4.398217	2
8	Mr. Brownstone	90	3.815400	2
9	Paradise City	90	6.772450	2

In [61]:
```
query = \
'''
SELECT
    name,
    albumid,
    ROUND(Milliseconds / 60000.0, 3) AS minutes,
    mediatypeid
FROM
    tracks
WHERE
    mediatypeid IN (2, 3)
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[61]:

	Name	AlbumId	minutes	MediaTypeId
0	Balls to the Wall	2	5.709	2
1	Fast As a Shark	3	3.844	2
2	Restless and Wild	3	4.201	2
3	Princess of the Dawn	3	6.257	2
4	Welcome to the Jungle	90	4.559	2
5	It's So Easy	90	3.380	2
6	Nightrain	90	4.476	2
7	Out Ta Get Me	90	4.398	2
8	Mr. Brownstone	90	3.815	2
9	Paradise City	90	6.772	2

Date Functions (https://www.sqlitetutorial.net/sqlite-date-functions/)

Dates come with their own "numerical" representation which can be operated on. In SQL, we can calculate different date-based values using datetime functions.

In [62]:
```
query = \
'''
SELECT
    LastName,
    FirstName,
    title,
    BirthDate,
    strftime('%m', BirthDate) BirthMonth,
    HireDate
FROM employees
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[62]:

	LastName	FirstName	Title	BirthDate	BirthMonth	HireDate
0	Adams	Andrew	General Manager	1962-02-18 00:00:00	02	2002-08-14 00:00:00
1	Edwards	Nancy	Sales Manager	1958-12-08 00:00:00	12	2002-05-01 00:00:00
2	Peacock	Jane	Sales Support Agent	1973-08-29 00:00:00	08	2002-04-01 00:00:00
3	Park	Margaret	Sales Support Agent	1947-09-19 00:00:00	09	2003-05-03 00:00:00
4	Johnson	Steve	Sales Support Agent	1965-03-03 00:00:00	03	2003-10-17 00:00:00
5	Mitchell	Michael	IT Manager	1973-07-01 00:00:00	07	2003-10-17 00:00:00
6	King	Robert	IT Staff	1970-05-29 00:00:00	05	2004-01-02 00:00:00
7	Callahan	Laura	IT Staff	1968-01-09 00:00:00	01	2004-03-04 00:00:00

In [63]:
```
query = \
'''
SELECT
    LastName,
    FirstName,
    title,
    HireDate,
    DATE(HireDate,
        'start of month',
        '+1 month',
        '-1 day') last_day_of_hire_month
FROM employees;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[63]:

	LastName	FirstName	Title	HireDate	last_day_of_hire_month
0	Adams	Andrew	General Manager	2002-08-14 00:00:00	2002-08-31
1	Edwards	Nancy	Sales Manager	2002-05-01 00:00:00	2002-05-31
2	Peacock	Jane	Sales Support Agent	2002-04-01 00:00:00	2002-04-30
3	Park	Margaret	Sales Support Agent	2003-05-03 00:00:00	2003-05-31
4	Johnson	Steve	Sales Support Agent	2003-10-17 00:00:00	2003-10-31
5	Mitchell	Michael	IT Manager	2003-10-17 00:00:00	2003-10-31
6	King	Robert	IT Staff	2004-01-02 00:00:00	2004-01-31
7	Callahan	Laura	IT Staff	2004-03-04 00:00:00	2004-03-31

String Functions (https://www.sqlitetutorial.net/sqlite-string-functions/)

Strings are very versitile, and SQL has plenty of operations for handling them. For example, we can use `LENGTH` to determine the lengths of the names for some of these playlists.

In [64]:
```
query = \
'''
SELECT
    Name name,
    LENGTH(name) name_length
FROM playlists
ORDER BY name_length DESC
LIMIT 10;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[64]:

	name	name_length
0	Classical 101 - Next Steps	26
1	Classical 101 - The Basics	26
2	Classical 101 - Deep Cuts	25
3	Heavy Metal Classic	19
4	Brazilian Music	15
5	Music Videos	12
6	On-The-Go 1	11
7	Audiobooks	10
8	90’s Music	10
9	Audiobooks	10

In [65]:
```
query = \
'''
SELECT
    FirstName,
    LastName,
    REPLACE(
        REPLACE(Title, "Manager", "Boss"),
        "IT", "Computer") slang_title
FROM employees;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[65]:

	FirstName	LastName	slang_title
0	Andrew	Adams	General Boss
1	Nancy	Edwards	Sales Boss
2	Jane	Peacock	Sales Support Agent
3	Margaret	Park	Sales Support Agent
4	Steve	Johnson	Sales Support Agent
5	Michael	Mitchell	Computer Boss
6	Robert	King	Computer Staff
7	Laura	Callahan	Computer Staff

Window Functions (https://www.sqlitetutorial.net/sqlite-window-functions/)

Window functions perform calculations on rows of data **based on their row-index**. For instance, we might want to know how a row of data compares to others with values lower than it, or maybe a just the index of the row itself, or even a cumulative sum. The syntax for the query looks a bit like this:

```
SELECT
    ...,
    [SOME_EXPRESSION](columns_of_stuff)   --<-- Here lies the "wind
ow function"
        OVER (                            --<-- Apply this function
*OVER* some window
            PARTION BY ... ) AS ...       --<-- Define the "window"
and the final column name
FROM ...
```

We perform operations on the records that are inside the window. The `PARTITION` tells you what is included in the window.

For instance, we can use it to perform a similar task as `.transform`. This query tells us how far a `Total` Invoice amount is from the average total for its city.

In [74]:
```
query = \
'''
SELECT
    CustomerId,
    InvoiceDate,
    BillingCity,
    Total,
    Total - AVG(Total) OVER (
        PARTITION BY BillingCity
      ) AS diff_from_city_avg
FROM invoices
ORDER BY CustomerId, InvoiceDate;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[74]:

	CustomerId	InvoiceDate	BillingCity	Total	diff_from_city_avg
0	1	2010-03-11 00:00:00	São José dos Campos	3.98	-1.680000
1	1	2010-06-13 00:00:00	São José dos Campos	3.96	-1.700000
2	1	2010-09-15 00:00:00	São José dos Campos	5.94	0.280000
3	1	2011-05-06 00:00:00	São José dos Campos	0.99	-4.670000
4	1	2012-10-27 00:00:00	São José dos Campos	1.98	-3.680000
...	...	...	...	...	...
407	59	2009-07-08 00:00:00	Bangalore	5.94	-0.166667
408	59	2010-02-26 00:00:00	Bangalore	1.99	-4.116667
409	59	2011-08-20 00:00:00	Bangalore	1.98	-4.126667
410	59	2011-09-30 00:00:00	Bangalore	13.86	7.753333
411	59	2012-05-30 00:00:00	Bangalore	8.91	2.803333

412 rows × 5 columns

Or, we can use it to calculate a **cumulative** sum.

In [67]:
```
query = \
'''
SELECT
    CustomerId,
    InvoiceDate,
    BillingCity,
    TOTAL,
    SUM(Total) OVER (
        PARTITION BY CustomerId
        ORDER BY InvoiceDate
    ) AS customer_running_total
FROM invoices
ORDER BY CustomerId, InvoiceDate
LIMIT 20;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[67]:

	CustomerId	InvoiceDate	BillingCity	Total	customer_running_total
0	1	2010-03-11 00:00:00	São José dos Campos	3.98	3.98
1	1	2010-06-13 00:00:00	São José dos Campos	3.96	7.94
2	1	2010-09-15 00:00:00	São José dos Campos	5.94	13.88
3	1	2011-05-06 00:00:00	São José dos Campos	0.99	14.87
4	1	2012-10-27 00:00:00	São José dos Campos	1.98	16.85
5	1	2012-12-07 00:00:00	São José dos Campos	13.86	30.71
6	1	2013-08-07 00:00:00	São José dos Campos	8.91	39.62
7	2	2009-01-01 00:00:00	Stuttgart	1.98	1.98
8	2	2009-02-11 00:00:00	Stuttgart	13.86	15.84
9	2	2009-10-12 00:00:00	Stuttgart	8.91	24.75
10	2	2011-05-19 00:00:00	Stuttgart	1.98	26.73
11	2	2011-08-21 00:00:00	Stuttgart	3.96	30.69
12	2	2011-11-23 00:00:00	Stuttgart	5.94	36.63
13	2	2012-07-13 00:00:00	Stuttgart	0.99	37.62
14	3	2010-03-11 00:00:00	Montréal	3.98	3.98
15	3	2010-04-21 00:00:00	Montréal	13.86	17.84
16	3	2010-12-20 00:00:00	Montréal	8.91	26.75
17	3	2012-07-26 00:00:00	Montréal	1.98	28.73
18	3	2012-10-28 00:00:00	Montréal	3.96	32.69
19	3	2013-01-30 00:00:00	Montréal	5.94	38.63

And (among other things), we could assign a rank to each track of each album based on the length

In [68]:
```
query = \
'''
SELECT
    Name,
    Milliseconds,
    AlbumId,
    RANK () OVER (
        PARTITION BY AlbumId
        ORDER BY Milliseconds DESC
    ) LengthRank
FROM tracks
LIMIT 50;
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[68]:

	Name	Milliseconds	AlbumId	LengthRank
0	For Those About To Rock (We Salute You)	343719	1	1
1	Spellbound	270863	1	2
2	Evil Walks	263497	1	3
3	Breaking The Rules	263288	1	4
4	Let's Get It Up	233926	1	5
5	Inject The Venom	210834	1	6
6	Night Of The Long Knives	205688	1	7
7	Put The Finger On You	205662	1	8
8	Snowballed	203102	1	9
9	C.O.D.	199836	1	10
10	Balls to the Wall	342562	2	1

EXERCISES

Problem 1

In fact, it looks like we can use the `page` referent to capture up to 20 results for *multiple pages* of results (think of scrolling through search results), and append each page to a collection of final results. Is this possible? If so, adjust the above function to include the `page` referent in Genius to return more than 20 results for a search term. If not, explain why.

In [69]:
```
# your code here
json_data = genius("The Beatles",20)
hits = [hit['result'] for hit in json_data]
hits_json = json.dumps(hits)

# load JSON into DataFrame
df = pd.read_json(hits_json)
df
```

```
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/42126680
17.py:7: FutureWarning: Passing literal json to 'read_json' is deprecated
and will be removed in a future version. To read from a literal string, w
rap it in a 'StringIO' object.
  df = pd.read_json(hits_json)
```

Out[69]:

ate_for_display	release_date_with_abbreviated_month_for_display	song_art_image_thu
tember 13, 1965	Sep. 13, 1965	https://images.genius.com/f9bfd62a8c65
May 8, 1970	May. 8, 1970	https://images.genius.com/38df3b59f231
August 26, 1968	Aug. 26, 1968	https://images.genius.com/537342a11e24
tember 26, 1969	Sep. 26, 1969	https://images.genius.com/04df90137154
tember 26, 1969	Sep. 26, 1969	https://images.genius.com/003c2b3d4b48
tember 26, 1969	Sep. 26, 1969	https://images.genius.com/14ca82e1dbb4
August 5, 1966	Aug. 5, 1966	https://images.rapgenius.com/b669c9e35
ecember 3, 1965	Dec. 3, 1965	https://images.genius.com/1a5e9183169b
May 26, 1967	May. 26, 1967	https://images.genius.com/0123ecd81f4c
vember 22, 1968	Nov. 22, 1968	https://images.genius.com/85f5a0ea644
vember 24, 1967	Nov. 24, 1967	https://images.genius.com/81faf3566d8a
vember 22, 1968	Nov. 22, 1968	https://images.genius.com/85f5a0ea644

ate_for_display	release_date_with_abbreviated_month_for_display	song_art_image_thu
›bruary 13, 1967	Feb. 13, 1967	https://images.genius.com/2b88add61eeb‹
›ember 12, 1969	Dec. 12, 1969	https://images.genius.com/31222935882c‹
May 26, 1967	May. 26, 1967	https://images.genius.com/0123ecd81f4c‹
May 26, 1967	May. 26, 1967	https://images.genius.com/0123ecd81f4c‹
›ember 22, 1968	Nov. 22, 1968	https://images.genius.com/73d9b2583c1e‹
›cember 3, 1965	Dec. 3, 1965	https://images.genius.com/db1f79f43a91‹
July 23, 1965	Jul. 23, 1965	https://images.genius.com/c9366977715‹
August 5, 1966	Aug. 5, 1966	https://images.genius.com/48beb113d4f5‹

In [70]:
```
df_stats = df['stats'].apply(pd.Series)
df_stats.rename(columns={c:'stat_' + c for c in df_stats.columns},
                inplace=True)

df_primary = df['primary_artist'].apply(pd.Series)
df_primary.rename(columns={c:'primary_artist_' + c for c in df_primary.colu
                  inplace=True)

df = pd.concat((df, df_stats, df_primary), axis=1)
```

```
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/22698097
98.py:1: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_stats = df['stats'].apply(pd.Series)
/var/folders/f5/7rjytg0x7ml45vv503fg5g6r0000gn/T/ipykernel_80449/22698097
98.py:5: FutureWarning: Returning a DataFrame from Series.apply when the
supplied function returns a Series is deprecated and will be removed in a
future version.
  df_primary = df['primary_artist'].apply(pd.Series)
```

In [71]:
```
df[['stat_unreviewed_annotations', 'stat_hot', 'stat_pageviews', 'stat_conc
```

Out[71]:

	stat_unreviewed_annotations	stat_hot	stat_pageviews	stat_concurrents
0	2	False	2252739	7.0
1	1	False	1695402	2.0
2	3	False	1266540	2.0
3	4	False	1251257	NaN
4	5	False	1139975	NaN
5	3	False	1049868	NaN
6	1	False	945703	2.0
7	2	False	848392	5.0
8	1	False	811654	NaN
9	2	False	786860	NaN
10	2	False	703832	NaN
11	0	False	621365	NaN
12	1	False	549690	NaN
13	2	False	545281	NaN
14	0	False	528124	2.0
15	0	False	523954	NaN
16	2	False	494037	NaN
17	0	False	452253	NaN
18	1	False	407709	3.0
19	3	False	380898	NaN

Problem 2

Write a SQL query which provides the minimum, maximum, and average track count of albums for each genre. So, each row should be a genre, and the columns would reflect the minimum track count, maximum track count, and average track count. *Feel free to use DB Brower as your "scratchpad" to test out your code.*

In [72]:
```
# your code here
query = \
'''
SELECT
    AlbumId, genreid, TrackId, AVG(count) OVER(
        PARTITION BY GenreId
    ) AS 'avg Count',
    MIN(count) OVER(
        PARTITION BY GenreId
    ) AS 'min Count',
    MAX(count) OVER(
        PARTITION BY GenreId
    ) AS 'max Count'
    FROM (
        SELECT  tr.AlbumId, tr.GenreId,  tr.TrackId, COUNT(tr.trackID) OVER
            PARTITION By tr.AlbumId
        ) AS count
        FROM tracks tr
    )
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[72]:

	AlbumId	GenreId	TrackId	avg Count	min Count	max Count
0	1	1	1	13.674634	1	57
1	1	1	6	13.674634	1	57
2	1	1	7	13.674634	1	57
3	1	1	8	13.674634	1	57
4	1	1	9	13.674634	1	57
...	...	...	...	...	...	...
3498	343	24	3499	1.054054	1	2
3499	344	24	3500	1.054054	1	2
3500	345	24	3501	1.054054	1	2
3501	346	24	3502	1.054054	1	2
3502	317	25	3451	1.000000	1	1

3503 rows × 6 columns

Problem 3

Using window functions and the `chinook` database, write a query which tells us the time between each invoice for each customer in the `invoices` table. E.g., you might have a column that says "time_since_last_invoice".

In [79]:
```
# your code here
query = \
'''
SELECT InvoiceId, CustomerId, InvoiceDate,
    JULIANDAY(InvoiceDate) - JULIANDAY(lag(InvoiceDate)
        OVER (ORDER BY InvoiceDate) ) time_since_last_invoice
FROM invoices
'''

df_result = pd.read_sql(query, engine)
df_result
```

Out[79]:

	InvoiceId	CustomerId	InvoiceDate	time_since_last_invoice
0	1	2	2009-01-01 00:00:00	NaN
1	2	4	2009-01-02 00:00:00	1.0
2	3	8	2009-01-03 00:00:00	1.0
3	4	14	2009-01-06 00:00:00	3.0
4	5	23	2009-01-11 00:00:00	5.0
...	...	...	...	...
407	408	25	2013-12-05 00:00:00	1.0
408	409	29	2013-12-06 00:00:00	1.0
409	410	35	2013-12-09 00:00:00	3.0
410	411	44	2013-12-14 00:00:00	5.0
411	412	58	2013-12-22 00:00:00	8.0

412 rows × 4 columns

Problem 4

Take a look at the documentation for [pandasql (https://pypi.org/project/pandasql/)](https://pypi.org/project/pandasql/). Load in any data frame of your choosing, and select a column that best represents a unique identifier for each row. E.g., if my data frame contains a list of customers, I might use the customer name or customer ID. Then, use *pandasql* to run a SQL query which performs a self join on your data based on that unique identifier column.

In [81]:
```
movies = {'movie_id':[1,2,3,4,5,6,7,8,9,10],
          'movie_name':['Percy Jackson and The Lightning Theif', 'Harry Pott
                         'Lilo and Stitch','Harry Potter and the Chamber of S
                        'Lilo & Stitch 2: Stitch Has a Glitch', 'Mulan', 'Won
                        'Spider-Man: Across the Spider-Verse', 'Blue Beetle']
          'sequel_id': [0,4,5,0,0,0,0,9,0,0]}
```

In [83]:
```
movies_df = pd.DataFrame(movies)
movies_df
```

Out[83]:

	movie_id	movie_name	sequel_id
0	1	Percy Jackson and The Lightning Theif	0
1	2	Harry Potter and the Sorcerers Stone	4
2	3	Lilo and Stitch	5
3	4	Harry Potter and the Chamber of Secrets	0
4	5	Lilo & Stitch 2: Stitch Has a Glitch	0
5	6	Mulan	0
6	7	Wonka	0
7	8	Spider-Man: Into the Spider-Verse	9
8	9	Spider-Man: Across the Spider-Verse	0
9	10	Blue Beetle	0

In [85]:
```
engine = create_engine('sqlite:///:memory:')

# Write the DataFrame to the engine
movies_df.to_sql('movies', engine, if_exists='replace')
```

Out[85]: 10

In [86]:
```
# your code here
query = \
'''
SELECT m.movie_name AS 'Original',
       s.movie_name AS 'Sequel'
FROM movies m
INNER JOIN movies s
    ON m.movie_id = s.sequel_id
'''


df_result = pd.read_sql(query, engine)
df_result
```

Out[86]:

	Original	Sequel
0	Harry Potter and the Chamber of Secrets	Harry Potter and the Sorcerers Stone
1	Lilo & Stitch 2: Stitch Has a Glitch	Lilo and Stitch
2	Spider-Man: Across the Spider-Verse	Spider-Man: Into the Spider-Verse