Spotify Charts

Midterm Project
DSC 10, Fall 2022
due Tuesday, November 1st at 11:59PM

Welcome to the Midterm Project! Projects in DSC 10 are similar in format to homeworks, but are different in a few key ways. First, a project is comprehensive, meaning that it draws upon everything we've learned this quarter so far. Second, since problems can vary quite a bit in difficulty, some problems will be worth more points than others. Finally, in a project, the problems are more open-ended; they will usually ask for some result, but won't tell you what method should be used to get it. There might be several equally-valid approaches, and several steps might be necessary. This is closer to how data science is done in "real life."

It is important that you **start early** on the project! It will take the place of a homework in the week that it is due, but you should also expect it to take longer than a homework. You are especially encouraged to **find a partner** to work through the project with. If you work with a partner, you must follow the Pair Programming Guidelines (https://dsc10.com/pair-programming/) on the course website. In particular, you must work together at the same time, and you are not allowed to split up the problems and each work on certain problems. If you work with a partner, only one of you needs to upload your notebook to Gradescope; after uploading, you'll see an option to add the other partner to the submission.

Important: The `otter` tests don't usually tell you that your answer is correct. More often, they help catch basic mistakes. It's up to you to ensure that your answer is correct. If you're not sure, ask someone (not for the answer, but for some guidance about your approach). Directly sharing answers between groups is not okay, but discussing problems with the course staff or with other students is encouraged.

Avoid looping through DataFrames. Do not import any packages. Loops in Python are slow, and looping through DataFrames should usually be avoided in favor of the DataFrame methods we've learned in class, which are much faster. Please do not import any additional packages – you don't need them, and our autograder may not be able to run your code if you do.

As you work through this project, there are a few resources you may want to have open:

- DSC 10 Reference Sheet (https://drive.google.com/file/d/1mQApk9Ovdi-QVqMgnNcq5dZcWucUKoG-/view)
- `babypandas` notes (https://notes.dsc10.com/front.html)
- `babypandas` documentation (https://babypandas.readthedocs.io/en/latest/)
- Other links in the Resources (https://dsc10.com/resources/) and Debugging (https://dsc10.com/debugging/) tabs of the course website

Start early, good luck, and let's begin! 🏃

In [24]:
```
# Please don't change this cell, but do make sure to run it.
import babypandas as bpd
import numpy as np
from IPython.display import HTML, display, IFrame, YouTubeVideo

import matplotlib.pyplot as plt
plt.style.use('ggplot')

import otter
import numbers # Not sure if needed
grader = otter.Notebook()

import warnings
warnings.simplefilter('ignore')

def play_spotify(uri):
    code = uri[uri.rfind(':')+1:]
    src = f"https://open.spotify.com/embed/track/{code}"
    width = 400
    height = 75
    display(IFrame(src, width, height))
```

Outline

The project is divided into seven sections, each of which contains several questions. Use the outline below to help you quickly navigate to the part of the project you're working on. Questions are worth one point each, unless they contain a ⭐️⭐️ next to them, in which case they are worth two points (e.g. **Question 1.3.** ⭐️⭐️). You can expect questions worth two points to be longer and more challenging than questions worth one point.

- The Data 🎧
- Section 1: What's a Song? 🤔
- Section 2: The Sound of Music 🎶
- Section 3: Slow and Steady 🐢🐇
- Section 4: Crazy in Love 💕
- Section 5: The Test of Time ⌛️
- Section 6: Party in the USA 💃
- Section 7: Encore 🔂

There's also an Emoji Quiz 💯 at the end of the project, just for fun. Try to identify songs and artists based on emoji descriptions, and see how many you can get!

The Data 🎧

[Spotify (https://spotify.com)](https://spotify.com), the world's popular music streaming service ([source (https://www.businessofapps.com/data/music-streaming-market/)](https://www.businessofapps.com/data/music-streaming-market/)), is known for keeping close tabs on what its subscribers listen to. They maintain an analytics site, called [Spotify Charts (https://charts.spotify.com)](https://charts.spotify.com), where they post the daily and weekly top 200 songs on Spotify in various countries and cities. This should not be a surprise – in [Lecture 7 (https://dsc10.com/resources/lectures/lec07/lec07.html)](https://dsc10.com/resources/lectures/lec07/lec07.html), we downloaded a dataset containing the top 200 songs globally on October 4th.

In this project, we will work with a dataset containing **the top 200 songs on Spotify each week, from the week of February 4th, 2021 through the week of July 14th, 2022, in each of the United States, Canada, and Mexico**. A song is in the top 200 for a given week and country if it is one of the 200 most streamed songs during that week in that country.

Run the cell below to load in the dataset and save it to a DataFrame named `charts` .

In [25]:
```
charts = bpd.read_csv('data/weekly_charts.csv')
charts
```

Out[25]:

	week	rank	track_name	uri	release_date	stream
0	2021-02-04	1	drivers license	spotify:track:7lPN2DXiMsVn7XUKtOW1CS	2021-01-08	205431
1	2021-02-04	2	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	91651
2	2021-02-04	3	Save Your Tears	spotify:track:5QO79kh1waicV47BqGRL3g	2020-03-20	86606
3	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
4	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
...	...	...	...	...	...	..
70178	2022-07-14	196	Get Into It (Yuh)	spotify:track:0W6I02J9xcqK8MtSeosEXb	2021-06-25	159210
70179	2022-07-14	197	Fancy Like	spotify:track:58UKC45GPNTflCN6nwCUeF	2022-01-21	159012
70180	2022-07-14	198	Stick Season	spotify:track:0GNVXNz7Jkicfk2mp5OyG5	2022-07-08	158330
70181	2022-07-14	199	Call Out My Name	spotify:track:09mEdoA6zrmBPgTEN5qXmN	2018-03-30	158323
70182	2022-07-14	200	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	157921

70183 rows × 24 columns

`charts` has 24 columns.

In [26]:
```
charts.columns
```

Out[26]:
```
Index(['week', 'rank', 'track_name', 'uri', 'release_date', 'streams
',
       'artist_names', 'artist_individual', 'artist_id', 'artist_gen
re',
       'artist_img', 'danceability', 'energy', 'key', 'mode', 'loudn
ess',
       'speechiness', 'acousticness', 'instrumentalness', 'liveness'
,
       'valence', 'tempo', 'duration', 'country'],
      dtype='object')
```

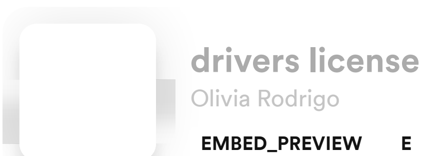
Below, we describe some of the columns of `charts` .

Column	Description
`'week'`	Week during which the song was in the top 200.
`'rank'`	The position of the song in the top 200, in the specified country.
`'track_name'`	The name of the song.
`'uri'`	The song's uniform resource indicator. This is an identfier that can be used to play the song on Spotify.
`'release_date'`	The date on which the song was released.
`'streams'`	The number of streams that the song received during the specified week in the specified country.
`'artist_names'`	All artists on the song.
`'artist_individual'`	One of the artists on the song. (If there are n artists on the song, the song appears in n rows of `charts` for each week and country it was in the top 200, once for each artist.)
`'artist_id'`	The individual artist's uniform resource indicator.
`'artist_genre'`	The individual artist's primary genre.
`'artist_img'`	A URL to the image of the individual artist.
`'duration'`	The length of the song, in milliseconds.
`'country'`	The country in which the song was in the top 200 in the specified week.

There are several columns – namely, `'danceability'` , `'energy'` , `'key'` , `'mode'` , `'loudness'` , `'speechiness'` , `'acousticness'` , `'instrumentalness'` , `'liveness'` , `'valence'` , and `'tempo'` – that we didn't describe above. These are all *audio features*, meaning they describe the musical content of songs, as opposed to the other columns, which describe metadata. Spotify provides [documentation (https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features)](https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features) that describes what each audio feature means. We'll provide you a link to this documentation again right before Section 2, when you'll actually start using these columns.

As the table above mentions, we can use a song's `'uri'` to play it on Spotify. We've provided you with a function named `play_spotify` that takes in a `'uri'` and plays the song in your notebook. Run the cell below to see it in action!

In [27]:
```
# URI for Olivia Rodrigo's "drivers license"
play_spotify('spotify:track:7lPN2DXiMsVn7XUKtOW1CS')
```



Section 1: What's a Song? 🤔

(return to the outline)

Let's look at the first and last few rows of `charts` once again.

In [28]:
```
charts
```

Out[28]:

	week	rank	track_name	uri	release_date	stream
0	2021-02-04	1	drivers license	spotify:track:7lPN2DXiMsVn7XUKtOW1CS	2021-01-08	205431
1	2021-02-04	2	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	91651
2	2021-02-04	3	Save Your Tears	spotify:track:5QO79kh1waicV47BqGRL3g	2020-03-20	86606
3	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
4	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
...	...	...	...	...	...	..
70178	2022-07-14	196	Get Into It (Yuh)	spotify:track:0W6I02J9xcqK8MtSeosEXb	2021-06-25	159210
70179	2022-07-14	197	Fancy Like	spotify:track:58UKC45GPNTflCN6nwCUeF	2022-01-21	159012
70180	2022-07-14	198	Stick Season	spotify:track:0GNVXNz7Jkicfk2mp5OyG5	2022-07-08	158330
70181	2022-07-14	199	Call Out My Name	spotify:track:09mEdoA6zrmBPgTEN5qXmN	2018-03-30	158323
70182	2022-07-14	200	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	157921

70183 rows × 24 columns

You may notice that some songs, like `'Mood (feat. iaan dior)'` , appear multiple times. This happens for a few reasons. For one, songs that appear on the top 200 for multiple weeks will have separate rows for each week. Furthermore, for each week that a song appears on the top 200, there will be a separate row for each artist included on that song. Notice that the `'artist_names'` column has **all** artists that collaborated on a song, and the `'artist_individual'` has just one. In addition, `charts` contains the top 200 for each week for each of the United States, Canada, and Mexico. There could be other reasons why a song might appear in multiple rows of `charts` , as well.

In this first section of the project, we'll work towards understanding which rows of `charts` actually correspond to the same song.

Question 1.1. For now, we'll think of a song as being defined by its `'uri'` . How many distinct `'uri'` s actually appear in this dataset? Store your answer in a variable called `unique_uris` .

In [29]:
```
unique_uris = len(charts.get('uri').unique())
unique_uris
```

Out[29]: 2850

In [30]:
```
grader.check("q1_1")
```

Out[30]: **q1_1** passed!

Although the dataset has over 70,000 rows, it contains far fewer songs.

It turns out that `'uri'` is not actually a unique indicator for each song. One song may appear on Spotify under various `'uri'` s if there are different versions of the song, such as an explicit version and a "clean" version, or a remix. Similarly, sometimes a song is released as a single, then as part of an album, and maybe years later as part of a "best-of" compilation album.

Question 1.2. To illustrate this, let's look at the track named `'Astronaut In The Ocean'` by `'Masked Wolf'` . (You may be familiar with this song from TikTok – it starts with "What you know about rollin' down in the deep?")

Set `astronaut_ocean_uris` to an array of all the unique `'uri'` s associated with the `'track_name'` `'Astronaut in the Ocean'` .

In [31]:
```
v = charts[charts.get('track_name') == 'Astronaut In The Ocean'].get('
uri').unique()
astronaut_ocean_uris = np.array(v)
astronaut_ocean_uris
```

Out[31]:
```
array(['spotify:track:3Ofmpyhv5UAQ70mENzB277',
       'spotify:track:3VT8hOC5vuDXBsHrR53WFh',
       'spotify:track:0BGwAKW4u8kWOhWFflZxfl',
       'spotify:track:6E90gq0KO6FYZVOXx8kCcC'], dtype=object)
```

In [32]:
```
grader.check("q1_2")
```

Out[32]: **q1_2** passed!

As we saw in the data description section, to play a song in our notebook, we call the function `play_spotify` on the song's `'uri'`. For example, the next cell plays a random song.

In [33]:
```
random_uri = np.random.choice(charts.get('uri')) # This line randomly
selects a uri.
play_spotify(random_uri) # This line plays the song with that uri.
```

Beautiful Mistakes (feat. Megan Thee

Maroon 5, Megan Thee Stallion

EMBED_PREVIEW

Question 1.3. Loop through all the `'uri'` s in `astronaut_ocean_uris` and play each song. Since you're using a loop, you should only have to call the function `play_spotify` one time!

In [34]:
```
for x in astronaut_ocean_uris:
    b = play_spotify(x)
print(b)
```

Astronaut In The Ocean
Masked Wolf
EMBED_PREVIEW E

Astronaut In The Ocean
Masked Wolf
EMBED_PREVIEW E

Astronaut In The Ocean
Masked Wolf
EMBED_PREVIEW E

Astronaut In The Ocean
Masked Wolf
EMBED_PREVIEW E

```
None
```

`'Astronaut In The Ocean'` is not the only song with multiple `'uri'` s. Let's take a look at how common it is to have multiple `'uri'` s for one `'track_name'` .

Question 1.4. ⭐⭐ Create a DataFrame, indexed by `'track_name'` , with just one column called `'uri_count'` containing the number of different `'uri'` s associated with each `'track_name'` . Sort the rows in descending order of `'uri_count'` and assign the resulting DataFrame to the variable `uris_per_track` .

In [35]:
```
clean_chart = charts.get(['track_name', 'uri']).groupby(['track_name',
'uri']).count().reset_index().groupby('track_name').count().sort_value
s(by='uri', ascending=False)
uris_per_track = clean_chart.assign(uri_count = clean_chart.get('uri')
).drop(columns='uri')
uris_per_track
```

Out[35]:

track_name	uri_count
Toxic	5
Astronaut In The Ocean	4
Memories	4
As the World Caves In	4
Bonita	3
...	...
Hello	1
Hello (feat. A Boogie Wit da Hoodie)	1
Here Comes Santa Claus (Right Down Santa Claus Lane)	1
Here Comes Santa Claus (Right Down Santa Claus Lane) - 1947 Version	1
Índigo	1

2521 rows × 1 columns

In [36]:
```
grader.check("q1_4")
```

Out[36]: **q1_4** passed!

Question 1.5. What's the average number of `'uri'` s per track? Store your answer in a variable called `avg_uri_count` .

In [37]:
```
avg_uri_count = uris_per_track.get('uri_count').mean()
avg_uri_count
```

Out[37]: `1.1305037683458945`

In [38]:
```
grader.check("q1_5")
```

Out[38]: **q1_5** passed!

Let's look more closely at the song `'Toxic'`, which has more `'uri'` s than any other `'track_name'` in the dataset. Part of the reason it has so many `'uri'` s is that there are actually several different songs named `'Toxic'`, by different artists.

Question 1.6. Create an array called `toxic_artists` containing all unique `'artist_names'` that have a song named `'Toxic'`.

In [39]:
```
toxic_artists = charts[charts.get('track_name') == 'Toxic'].groupby(['
uri', 'artist_names']).count().reset_index().get('artist_names').uniqu
e()
toxic_artists
```

Out[39]:
```
array(['BoyWithUke', 'Polo G', 'Britney Spears'], dtype=object)
```

In [40]:
```
grader.check("q1_6")
```

Out[40]: **q1_6** passed!

If you did Question 1.6 correctly, you'll see that there are 3 different `'artist_names'` who have songs named `'Toxic'`. Let's try and redo our calculation for *all* `'track_names'` in our dataset, not just `'Toxic'`.

Question 1.7. ⭐⭐ Create a DataFrame of all `'track_names'` that are associated with **multiple** `'artist_names'`. Your DataFrame should have two columns:

1. `'track_name'`, the name of a song.
2. `'num_artists'`, the number of different artists (or groups of artists) that have songs by this name.

Save your DataFrame as `repeat_titles`.

In [41]:
```
cleaned = charts.get(['track_name', 'artist_names']).groupby(['track_name','artist_names']).count().reset_index().groupby('track_name').count().sort_values(by='artist_names', ascending= False)
repeat_titles = cleaned.assign(num_artists = cleaned.get('artist_names')).drop(columns='artist_names').reset_index()
repeat_titles = repeat_titles[repeat_titles.get('num_artists')>1]
repeat_titles
```

Out[41]:

	track_name	num_artists
0	Memories	3
1	Christmas (Baby Please Come Home)	3
2	Toxic	3
3	Lost	2
4	Body	2
...	...	...
25	Ella	2
26	Rudolph The Red-Nosed Reindeer	2
27	As the World Caves In	2
28	Have Yourself A Merry Little Christmas	2
29	Y Si Se Quiere Ir	2

30 rows × 2 columns

In [42]:
```
grader.check("q1_7")
```

Out[42]: **q1_7** passed!

Question 1.8. ⭐⭐ Add a column to `repeat_titles` called `'all_artists'`. Each entry of this column should be a string of all the `'artist_names'` associated with a given `'track_name'`, in any order. Format each string so that `'; '` appears between each of the `'artist_names'`.

For example, the `'track_name'` `'Memories'` is associated with the `'artist_names'` `'Maroon 5'`, `'dvsn, Ty Dolla $ign'`, and `'Conan Grey'`, so the value in the `'all_artists'` column for `'Memories'` could be `'Maroon 5; dvsn, Ty Dolla $ign'; Conan Grey'`.

Hint: Start by defining a function, then `apply` this function to each `'track_name'`.

In [43]:
```
x = charts.get(['track_name', 'artist_names']).groupby(['track_name','
artist_names']).count().reset_index()
def all_artists(names):
    s = x[x.get('track_name') == names].get('artist_names')
    y = np.array(s)
    z = list(y)
    t = '; '
    g = t.join(z)
    return g
```

In [44]:
```
repeat_titles = repeat_titles.assign(all_artists = repeat_titles.get('
track_name').apply(all_artists))
repeat_titles
```

Out[44]:

	track_name	num_artists	all_artists
0	Memories	3	Conan Gray; Maroon 5; dvsn, Ty Dolla $ign
1	Christmas (Baby Please Come Home)	3	Darlene Love; Mariah Carey; Michael Bublé
2	Toxic	3	BoyWithUke; Britney Spears; Polo G
3	Lost	2	Frank Ocean; Maroon 5
4	Body	2	Megan Thee Stallion; Russ Millions, Tion Wayne
...	...	...	...
25	Ella	2	Boza; Junior H
26	Rudolph The Red-Nosed Reindeer	2	Burl Ives; Dean Martin
27	As the World Caves In	2	Matt Maltese; Sarah Cothran
28	Have Yourself A Merry Little Christmas	2	Judy Garland; Sam Smith
29	Y Si Se Quiere Ir	2	Hijos De Barron; Luis Angel "El Flaco"

30 rows × 3 columns

In [45]:
```
grader.check("q1_8")
```

Out[45]: **q1_8** passed!

So far, we've established that we can't use `'uri'` to identify a song, because some songs have multiple versions and hence multiple `'uri'` s. We also can't use `'track_name'` to identify a song, because different artists sometimes have songs with the same name.

However, it's a pretty safe assumption that no artist will have two different songs with the same name, so from here on, we will use both `'track_name'` and `'artist_names'` to identify a song.

Question 1.9. If we define a song as a combination of `'track_name'` and `'artist_names'` , how many songs are in `charts` ? Store your answer in a variable called `num_songs` .

In [46]:
```
num_songs = charts.groupby(['track_name','artist_names']).count().shap
e[0]
num_songs
```

Out[46]: 2554

In [47]:
```
grader.check("q1_9")
```

Out[47]: **q1_9** passed!

Defining a song in this way means that multiple rows in `charts` correspond to the same song, for a variety of reasons we have already explored. If we want to make a DataFrame of just songs, we will need a way to handle discrepancies between the rows of `charts` that correspond to the same song. In each column where it makes sense to do so, we'll just take the median of all values corresponding to the same song.

Question 1.10. Create a DataFrame called `songs_on_charts` containing one row for each song that appears in `charts`. The first two columns should be `'track_name'` and `'artist_names'`. The remaining columns should be those listed below, and each column should contain the **median** value among all instances of the song.

- `'danceability'`
- `'energy'`
- `'key'`
- `'mode'`
- `'loudness'`
- `'speechiness'`
- `'acousticness'`
- `'instrumentalness'`
- `'liveness'`
- `'valence'`
- `'tempo'`
- `'duration'`

In [48]:
```
songs_on_charts = charts.groupby(['track_name','artist_names']).median
().drop(columns=['rank','streams']).reset_index()
songs_on_charts
```

Out[48]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 14 columns

In [49]:

```
grader.check("q1_10")
```

Out[49]: **q1_10** passed!

For the next few sections of the project, we'll use data from the `songs_on_charts` DataFrame to explore some of the audio features of these songs. As a reminder, Spotify provides [documentation (https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features)](https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features) on what these features represent. Note that many of these features (such as `'valence'`) are defined and determined by Spotify. We have no way of knowing exactly how they determine the values of these audio features for each song, as their algorithms are proprietary.

Section 2: The Sound of Music 🎶

(return to the outline)

We'll start this section by providing you with `songs`, a correct copy of the `songs_on_charts` DataFrame you produced in the last question of Section 1. We're providing you with a fresh copy of the data to prevent any earlier mistakes from creating a snowball effect. It's a good idea to verify that your `songs_on_charts` DataFrame and the provided `songs` DataFrame have the same number of rows, otherwise you certainly made a mistake in Section 1.

And if you didn't complete Section 1, that's fine – you can start from Section 2 without using any results from Section 1.

In [50]:
```
songs = bpd.read_csv('data/songs.csv')
songs
```

Out[50]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 14 columns

As a reminder, `songs` has one row for every song that appeared on the top 200 weekly charts during the period of data collection. The columns contain information about the audio features of songs, as mentioned at the end of Section 1.

Let's try and make some sense of these audio features!

Question 2.1. First, let's make the `'duration'` column more readable by changing the units from milliseconds to minutes. Add a new column to `songs` called `'duration_min'` that contains the duration of each track in minutes, without rounding, and drop the `'duration'` column.

In [51]:
```
songs = songs.assign(duration_min = songs.get('duration') / 60000).dro
p(columns='duration')
songs
```

Out[51]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 14 columns

In [52]:
```
grader.check("q2_1")
```

Out[52]: **q2_1** passed!

Question 2.2. What's the longest song, in minutes, in `songs`? Save the `'track_name'` of this song to the variable `longest_song_name`, save the `'artist_names'` of this song to the variable `longest_song_artist`, and save the length of this song (in minutes) to the variable `longest_song_minutes`.

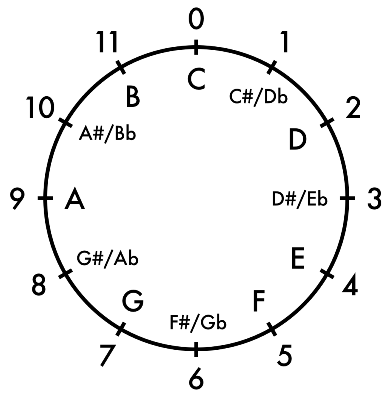
In [53]:
```
longest_song_name = songs.sort_values(by='duration_min', ascending=False).get('track_name').iloc[0]
longest_song_artist = songs.sort_values(by='duration_min', ascending=False).get('artist_names').iloc[0]
longest_song_minutes = songs.sort_values(by='duration_min', ascending=False).get('duration_min').iloc[0]

print('The longest song in the dataset is "' + longest_song_name + '" by ' + longest_song_artist + '.')
print('It lasts a whopping ' + str(round(longest_song_minutes, 1)) + ' minutes!')
```

```
The longest song in the dataset is "Jesus Lord pt 2" by Kanye West.
It lasts a whopping 11.5 minutes!
```

In [54]:
```
grader.check("q2_2")
```

Out[54]: **q2_2** passed!

A [musical key (https://www.studybass.com/lessons/harmony/keys-in-music/)](https://www.studybass.com/lessons/harmony/keys-in-music/) describes a certain set of pitches, and usually a song is played in a certain key. In music theory, the different keys are associated with integers from 0 to 11 using what's known as [pitch class notation (https://en.wikipedia.org/wiki/Pitch_class#Other_ways_to_label_pitch_classes)](https://en.wikipedia.org/wiki/Pitch_class#Other_ways_to_label_pitch_classes). Often, the keys are represented on a clock-like diagram like the one below, which shows the pitches associated with each of the 12 different musical keys.



([source (https://davidkulma.com/musictheory/integers)](https://davidkulma.com/musictheory/integers))

If you want to hear what each key sounds like, check out [this virtual piano 🎹 (https://www.musicca.com/piano)](https://www.musicca.com/piano).

Question 2.3. Create an array of all the unique keys in the `songs` DataFrame. Save it as `unique_keys` .

In [55]:
```
unique_keys = songs.get('key').unique()
unique_keys
```

Out[55]:
```
array([ 1. ,  9. ,  0. ,  5. ,  2. , 10. ,  6. ,  8. ,  7. ,  3. , 1
1. ,
         4. ,  2.5])
```

In [56]:
```
grader.check("q2_3")
```

Out[56]: **q2_3** passed!

In [57]:
```
charts.get('key').unique()
```

Out[57]:
```
array([10,  1,  0,  7,  3,  6, 11,  2,  4,  8,  5,  9])
```

In [58]:
```
x = songs_on_charts[songs_on_charts.get('key')== 2.5].get('track_name'
)
x
```

Out[58]:
```
41    34+35 Remix (feat. Doja Cat, Megan Thee Stalli...
Name: track_name, dtype: object
```

In [59]:
```
x = charts[charts.get('track_name').str.contains('feat. Doja Cat, Mega
n Thee Stalli')].get(['track_name','key'])
y = x[x.get('key') != 0].sort_values(by='key',ascending=True)
y
```

Out[59]:

	track_name	key
1033	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49995	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49661	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49660	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49659	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
...	...	...
1433	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
1035	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
1034	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49996	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5
49997	34+35 Remix (feat. Doja Cat, Megan Thee Stalli...	5

18 rows × 2 columns

Question 2.4. If you answered Question 2.3 correctly, you'll notice that not all of the keys are integers. This doesn't quite make sense, given the explanation we provided you before Question 2.3.

Below, in two sentences, explain why not all of the keys in `songs` are integers.

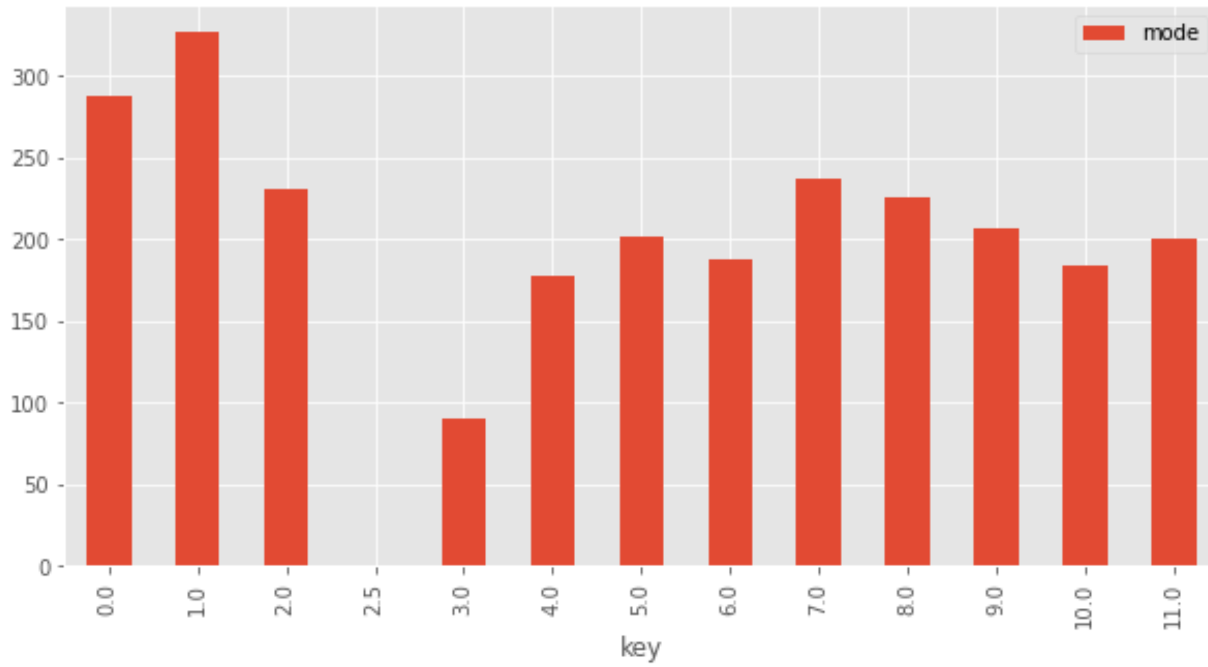
Hint: Find the unique keys in the `charts` DataFrame.

In songs dataframe, the values in the key column is the median of the all keys used in each song, so there is a song which uses 0 and 5 keys and has 36 entries so when we find the median of it we will find that the value is 2.5. thus there is a value that is not an integer in the key column of songs dataframe.

Question 2.5. Create a visualization that shows the distribution of `'key'` in the `songs` DataFrame.

In [75]:
```
songs.groupby('key').count().reset_index().get(['key','mode']).plot(kind='bar',x='key',figsize=(10,5))
```

Out[75]: <AxesSubplot:xlabel='key'>



Section 3: Slow and Steady 🐢🐇

([return to the outline](#))

In music, there are Italian words that describe the tempo, or pace, of a song. In this section, we will analyze the relationship between a song's tempo and its other audio features. But before we do that, we will convert the tempo of each song to its corresponding Italian description. Use the following definitions of Italian tempo markings:

Italian name	Corresponding tempo range, in beats per minute
Lento	[0, 60)
Adagio	[60, 90)
Adante	[90, 110)
Moderato	[110, 120)
Allegro	[120, 160)
Vivace	[160, 180)
Presto	180 or more

Question 3.1. Add a new column to `songs` called `'tempo_name'` that contains the Italian tempo name for each song.

In [38]:
```
def convert_to_italian(v):
    if v < 60:
        return 'Lento'
    elif v < 90:
        return 'Adagio'
    elif v < 110:
        return 'Adante'
    elif v < 120:
        return 'Moderato'
    elif v < 160:
        return 'Allegro'
    elif v < 180:
        return 'Vivace'
    elif v >= 180:
        return 'Presto'


songs = songs.assign(tempo_name = songs.get('tempo').apply(convert_to_
italian))
songs
```

Out[38]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 15 columns

In [39]:

```
grader.check("q3_1")
```

Out[39]: **q3_1** passed!

Question 3.2. Find the most common combination of `'tempo_name'` and `'key'` among all songs in `songs`. Save both answers in a list named `most_common_combo`. The `'tempo_name'` in the most common combination should come first. For example, your answer might look like `['Vivace', 3.0]`.

Similarly, find the least common combination of `'tempo_name'` and `'key'` among all songs in `songs` and save your answers in a `list` named `least_common_combo`, again with the `'tempo_name'` coming first.

In the case of a tie for most or least common, choose any of the combinations involved in the tie.

In [40]:
```
x = songs.groupby(['key','tempo_name']).count().reset_index().get(['key','tempo_name','track_name']).sort_values(by='track_name', ascending = False)
least_common = x.get(['tempo_name','key']).iloc[-1]
most_common = x.get(['tempo_name','key']).iloc[0]
```

In [41]:
```
m = (most_common.get('tempo_name'), most_common.get('key'))
most_common_combo = list(m)
l = (least_common.get('tempo_name'), least_common.get('key'))
least_common_combo = list(l)

print('The most common combination is a tempo of ' + most_common_combo[0] + ' and a key of ' + str(most_common_combo[1]) + '.')
print('The least common combination is a tempo of ' + least_common_combo[0] + ' and a key of ' + str(least_common_combo[1]) + '.')
```

```
The most common combination is a tempo of Allegro and a key of 1.0.
The least common combination is a tempo of Adante and a key of 2.5.
```

In [42]:
```
grader.check("q3_2")
```

Out[42]: **q3_2** passed!

Question 3.3. Let's identify which songs have the `most_common_combo` of `'tempo_name'` and `'key'`. Starting with `songs`, create a DataFrame of only the songs with this most common `'tempo_name'` and `'key'` combination. Save the result as `common_songs`.

In [43]:
```
common_songs = songs[(songs.get('tempo_name') == 'Allegro') & (songs.get('key') == 1.0)]
common_songs
```

Out[43]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	aco
43	3G (feat. Lil Uzi Vert)	Yeat, Lil Uzi Vert	0.758	0.572	1.0	1.0	-8.087	0.2050	
51	5X	Don Toliver	0.898	0.518	1.0	1.0	-6.991	0.2100	
53	7 rings	Ariana Grande	0.778	0.317	1.0	0.0	-10.732	0.3340	
63	999 (with Camilo)	Selena Gomez, Camilo	0.781	0.748	1.0	1.0	-4.604	0.2420	
66	A Keeper	Drake	0.600	0.482	1.0	1.0	-11.596	0.0701	
...	...	...	...	...	...	...	...	...	
2456	family ties (with Kendrick Lamar)	Baby Keem, Kendrick Lamar	0.711	0.611	1.0	1.0	-5.453	0.3300	
2491	mainstream sellout	Machine Gun Kelly	0.541	0.693	1.0	1.0	-4.252	0.0612	
2498	moved to miami (feat. Lil Baby)	Roddy Ricch, Lil Baby	0.717	0.444	1.0	1.0	-11.126	0.1800	
2529	thought i was playing	Gunna, 21 Savage	0.679	0.730	1.0	1.0	-7.483	0.0689	
2531	too easy	Gunna, Future	0.798	0.574	1.0	1.0	-6.548	0.1570	

126 rows × 15 columns

In [44]:
```
grader.check("q3_3")
```

Out[44]: **q3_3** passed!

We want to listen to some of these `common_songs` to see if they have a similar sound. But we have a problem. In order to play a song, we need its `'uri'`, and `common_songs` doesn't have a `'uri'` column. The `charts` DataFrame does have a `'uri'` column so we should be able to bring in that information by merging the two DataFrames. Though `charts` contains `'uri'`, it also has a ton of information that we don't need, since all of the relevant information per song is already in `songs`. As a result, before we merge, we should prepare a smaller, simpler DataFrame from `charts` with only the new information we need.

Question 3.4. Create a DataFrame called `to_merge` from `charts`. The DataFrame `to_merge` should have one row for each song (defined as a combination of `'track_name'` and `'artist_names'`) and three columns: `'track_name'`, `'artist_names'`, and `'uri'`. For each song, the associated `'uri'` should be the **first alphabetically**, among all `'uri'`s associated with that song.

In [45]:
```
to_merge = charts.groupby(['track_name','artist_names']).min().reset_index().get(['track_name','artist_names','uri'])
to_merge
```

Out[45]:

	track_name	artist_names	uri
0	'Til You Can't	Cody Johnson	spotify:track:4k3lPl8YTKuY8c1HelVnm3
1	'Till I Collapse	Eminem, Nate Dogg	spotify:track:4xkOaSrkexMciUUogZKVTS
2	(Don't Fear) The Reaper	Blue Öyster Cult	spotify:track:5QTxFnGygVM4jFQiBovmRo
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	spotify:track:2n1xrggQtAGEV1AgzvooGB
4	(There's No Place Like) Home for the Holidays ...	Perry Como	spotify:track:0hvN2v6fAcB6xWyW7UaooA
...	...	...	...
2549	you broke me first	Tate McRae	spotify:track:45bE4HXI0AwGZXfZtMp8JR
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	spotify:track:68pE8830rWrd5LSSfKcRqn
2551	¿Quién Te Crees?	MC Davo, Calibre 50	spotify:track:2LXOSAYiSrTfIf8smheLaz
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	spotify:track:3NqbKUOgaU2LgIFRbu4B12
2553	Índigo	Camilo, Evaluna Montaner	spotify:track:4knc1Fp3kbuq8bH2byOvLu

2554 rows × 3 columns

In [46]:
```
grader.check("q3_4")
```

Out[46]: **q3_4** passed!

Notice that `'track_name'` and `'artist_names'` are columns names in the `common_songs` DataFrame and in the `to_merge` DataFrame. Further, they are the *only* column names that these DataFrames have in common.

It turns out that when we merge two DataFrames without specifying which columns to merge on, `babypandas` will merge them on the set of shared column names, which means it will match up rows that have the same values in *all* shared columns.

Question 3.5. Merge `common_songs` and `to_merge` on both `'track_name'` and `'artist_names'`. Save the resulting DataFrame as `common_songs_uri`. Think about why we want to merge on both columns in this case (i.e. why we *can't* merge on just `'track_name'` or just `'artist_names'`).

In [47]:
```
common_songs_uri = to_merge.merge(common_songs)
common_songs_uri
```

Out[47]:

	track_name	artist_names	uri	danceability	energy
0	3G (feat. Lil Uzi Vert)	Yeat, Lil Uzi Vert	spotify:track:3O0XntET8Ee1nFI3rDTwOJ	0.758	0.572
1	5X	Don Toliver	spotify:track:2OcbewDrWFNTYRqpSzJBCY	0.898	0.518
2	7 rings	Ariana Grande	spotify:track:6ocbgoVGwYJhOv1GgI9NsF	0.778	0.317
3	999 (with Camilo)	Selena Gomez, Camilo	spotify:track:0EtuSDTRJYUwIPf4y6colz	0.781	0.748
4	A Keeper	Drake	spotify:track:0nAZGkBGKQCXyaoSJfRhC1	0.600	0.482
...	...	...	...	...	...
121	family ties (with Kendrick Lamar)	Baby Keem, Kendrick Lamar	spotify:track:3QFInJAm9eyaho5vBzxInN	0.711	0.611
122	mainstream sellout	Machine Gun Kelly	spotify:track:0XugRTkCzcwTJ0ZZJbeVHO	0.541	0.693
123	moved to miami (feat. Lil Baby)	Roddy Ricch, Lil Baby	spotify:track:3rjwafyisDpLdoJ4RecHp6	0.717	0.444
124	thought i was playing	Gunna, 21 Savage	spotify:track:3XLbDUB5BX2WqL2qoAsvtb	0.679	0.730
125	too easy	Gunna, Future	spotify:track:2Hph3X77ySNgBCIak5CMc6	0.798	0.574

126 rows × 16 columns

In [48]:

```
grader.check("q3_5")
```

Out[48]: **q3_5** passed!

Question 3.6. It would be great if we could listen to the songs in `common_songs_uri` to see if they sound alike, but there are too many songs to listen to them all. In an array called `certain_uris`, store the following `'uri'` s:

- the first alphabetical `'uri'` in `common_songs_uri`,
- then every 40th song thereafter, when the songs are ordered alphabetically by `'uri'`.

Then, play all the songs whose `'uri'` s are stored in `certain_uris`. As in Question 1.3, you should only call the function `play_spotify` one time!

In [49]:
```
b = common_songs_uri.sort_values(by='uri', ascending = True)
c = np.arange(0, 126, 40)
c
```

Out[49]:
```
array([  0,  40,  80, 120])
```

In [50]:
```
h = np.array([])

for i in c:
    x = b.get('uri').iloc[i]
    h = np.append(h,x)


certain_uris = h


# Play the songs here.
for x in certain_uris:
    p = play_spotify(x)
print(p)
```

Mr. Brightside
The Killers
EMBED_PREVIEW

too easy
Gunna, Future
EMBED_PREVIEW E

Dos Mil 16
Bad Bunny
EMBED_PREVIEW

Die For You (feat. Dominic Fike)
Justin Bieber, Dominic Fike
EMBED_PREVIEW

None

In [51]:
```
grader.check("q3_6")
```

Out[51]: **q3_6** passed!

Question 3.7. Now, let's categorize songs by their Italian tempo names. Specifically, find the mean of each numerical variable for each `'tempo_name'`. Store these means in a DataFrame indexed by `'tempo_name'` and sorted from slowest to fastest tempos. Save your DataFrame to the variable `song_means`.

In [52]:
```
song_means = songs.groupby('tempo_name').mean().sort_values(by='tempo'
)
song_means
```

Out[52]:

	danceability	energy	key	mode	loudness	speechiness	acousticnes
tempo_name							
Lento	0.382000	0.363000	8.333333	0.666667	-13.754667	0.065867	0.52933
Adagio	0.621284	0.559786	5.140704	0.625628	-7.147053	0.144588	0.31834
Adante	0.704633	0.634094	5.306588	0.576014	-6.452064	0.102016	0.26380
Moderato	0.698731	0.616624	5.129151	0.619926	-6.774517	0.081015	0.27654
Allegro	0.681836	0.613450	5.152151	0.631689	-6.842650	0.112475	0.24905
Vivace	0.622379	0.637733	5.192308	0.615385	-6.232313	0.166024	0.24186
Presto	0.513000	0.600714	5.116883	0.688312	-6.845909	0.139118	0.30643

In [53]:
```
grader.check("q3_7")
```

Out[53]: **q3_7** passed!

Question 3.8. One `'tempo_name'` category has far fewer songs than the others. Since there are too few songs of this `'tempo_name'` for us to draw any meaningful conclusions from, let's create a version of `song_means` without this row. Save the resulting DataFrame in `song_means_modified`.

In [54]:
```
songs.groupby('tempo_name').count().sort_values(by='track_name').index
[0]
```

Out[54]: 'Lento'

In [55]:
```
song_means_modified = song_means.iloc[1:7]
song_means_modified
```

Out[55]:

tempo_name	danceability	energy	key	mode	loudness	speechiness	acousticness
Adagio	0.621284	0.559786	5.140704	0.625628	-7.147053	0.144588	0.318340
Adante	0.704633	0.634094	5.306588	0.576014	-6.452064	0.102016	0.263806
Moderato	0.698731	0.616624	5.129151	0.619926	-6.774517	0.081015	0.276549
Allegro	0.681836	0.613450	5.152151	0.631689	-6.842650	0.112475	0.249058
Vivace	0.622379	0.637733	5.192308	0.615385	-6.232313	0.166024	0.241861
Presto	0.513000	0.600714	5.116883	0.688312	-6.845909	0.139118	0.306433

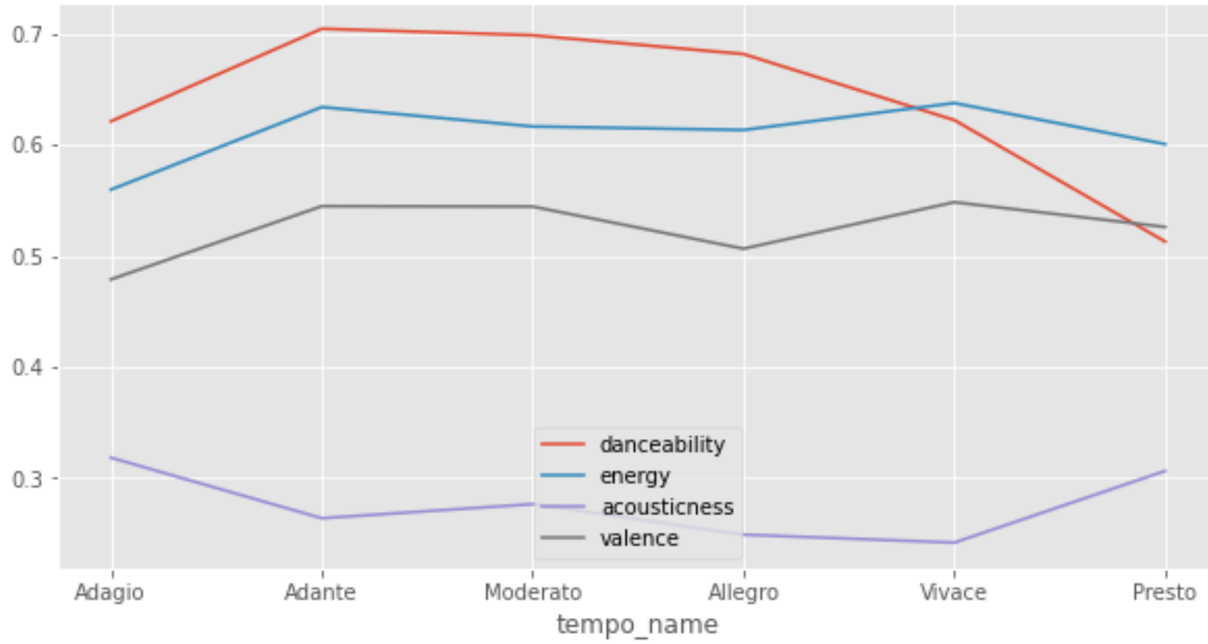
In [56]:
```
grader.check("q3_8")
```

Out[56]: **q3_8** passed!

Question 3.9. Using `song_means_modified`, create a line plot that portrays how `'danceability'`, `'energy'`, `'acousticness'`, and `'valence'` change according to `'tempo_name'`. Make sure your plot arranges songs from slowest to fastest tempos.

In [57]:
```
song_means_modified.get(['danceability', 'energy', 'acousticness','val
ence']).plot(kind='line', figsize=(10,5))
```

Out[57]: <AxesSubplot:xlabel='tempo_name'>



Question 3.10. You may have noticed from the plot in the previous question that `'energy'` and `'valence'` seem to move together. This means these variables are *associated*.

In the cell below, answer the following questions.

- Can we use the `'energy'` of a song to predict its `'valence'` ?
- If so, does this mean that high `'energy'` causes high `'valence'` ? Why or why not?

We can use the song's energy to predict the song's valence; however, we can't say that there is a causality between those two since there are other confounding factors that might have played a role in this behavior such as the tempo and the key of the song, but we can say there is an association or relation between them.

Section 4: Crazy in Love 💕

(return to the outline)

Now that we've developed an understanding of how a song's `'tempo_name'` relates to its audio features, let's turn our attention to the relationship between a song's `'track_name'` and its audio features. We'll start by looking at songs that contain `'love'` in the `'track_name'` and learning about what makes them special relative to other songs.

Question 4.1. Create a DataFrame called `love_and_not` that has all the same rows and columns as `songs`, plus one extra column, called `'has_love'`. This column should contain either the **string** `'True'` or `'False'`, corresponding to whether or not the string `'love'` is part of the song's `'track_name'`.

We consider `'love'` to be a part of a song's `'track_name'` even in the following scenarios:

- `'love'` is part of another word, e.g. `'track_name'` contains `'lovely'`.
- The capitalization is different, e.g. the `'track_name'` contains `'LoVE'`.

Note: It may seem strange that we're asking you to use the strings `'True'` and `'False'` rather than the Boolean values `True` and `False` directly; this will make more sense in the coming questions.

In [58]:
```
g = np.array(songs.get('track_name').str.lower().str.contains('love'))
def from_bool_to_str(x):
    if x == True:
        return str(True)
    elif x == False:
        return str(False)
o = np.vectorize(from_bool_to_str)
love_and_not = songs.assign(has_love = o(g))
love_and_not
```

Out[58]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 16 columns

In [59]:

```
grader.check("q4_1")
```

Out[59]: **q4_1** passed!

Question 4.2. Let's compare the `'loudness'` of songs whose `'track_name'` s include `'love'` with the songs whose `'track_name'` s don't include `'love'` . Calculate the mean `'loudness'` of all songs containing the word `'love'` and store that in `average_love_song_loudness` . Similarly, calculate the mean `'loudness'` of all songs not containing the word `'love'` and store that in `average_non_love_song_loudness` .

Note: `'loudness'` is represented as a negative number; smaller numbers correspond to quieter songs.

In [60]:
```
average_love_song_loudness = love_and_not[love_and_not.get('has_love')
== 'True'].get('loudness').mean()
average_non_love_song_loudness = love_and_not[love_and_not.get('has_lo
ve') == 'False'].get('loudness').mean()

print('The average loudness of songs whose titles include "love" is '
+ str(round(average_love_song_loudness, 2)) + '.')
print('The average loudness of songs whose titles don\'t include "love
" is ' + str(round(average_non_love_song_loudness, 2)) + '.')
```

```
The average loudness of songs whose titles include "love" is -6.49.
The average loudness of songs whose titles don't include "love" is -
6.75.
```

In [61]:
```
grader.check("q4_2")
```

Out[61]: **q4_2** passed!

Question 4.3. The audio features listed below are all measured on a 0 to 1 scale.

- `'danceability'`
- `'energy'`
- `'speechiness'`
- `'acousticness'`
- `'instrumentalness'`
- `'liveness'`
- `'valence'`

Let's try and understand how these features differ between songs with and without `'love'` in the `'track_name'`.

Create a DataFrame called `love_means`, indexed by `'has_love'`, that contains the mean value of each of the 7 features above, separately for songs with `'love'` in the `'track_name'` and songs without `'love'` in the `'track_name'`. `love_means` should have 2 rows – one where `'has_love'` is `'False'` and one where `'has_love'` is `'True'` – and 7 columns.

For instance, `love_means.get('energy').loc['False']` should be the mean `'energy'` among songs that don't have `'love'` in the `'track_name'`.

In [62]:
```
love_means = love_and_not.groupby('has_love').mean().drop(columns= ['k
ey', 'mode', 'loudness', 'tempo', 'duration_min'])
love_means
```

Out[62]:

	danceability	energy	speechiness	acousticness	instrumentalness	liveness	vale
has_love							
False	0.669461	0.612243	0.119282	0.266987	0.012123	0.179766	0.522
True	0.616239	0.603620	0.070163	0.286026	0.009477	0.161449	0.441

In [63]:
```
grader.check("q4_3")
```

Out[63]: **q4_3** passed!

`love_means` has all the information we need. However, for the purposes of creating visualizations, we need to change its format so that the columns become the rows and the rows become the columns. This is called *transposing* the DataFrame, and it's very easy to accomplish in `babypandas` by typing `.T` after the name of a DataFrame. Run the next cell to see what happens when we transpose `love_means`.

In [64]:
```
transposed_love = love_means.T
transposed_love
```

Out[64]:

has_love	False	True
danceability	0.669461	0.616239
energy	0.612243	0.603620
speechiness	0.119282	0.070163
acousticness	0.266987	0.286026
instrumentalness	0.012123	0.009477
liveness	0.179766	0.161449
valence	0.522467	0.441172

`transposed_love` has the same information that `love_means` does, it's just presented differently.

Question 4.4. Add a column called `'AbsDiff'` to `transposed_love` containing the absolute difference between the `'False'` and `'True'` columns.

In [65]:
```
transposed_love = transposed_love.assign(AbsDiff = abs(transposed_love
.get('False') - transposed_love.get('True')))
transposed_love
```

Out[65]:

has_love	False	True	AbsDiff
danceability	0.669461	0.616239	0.053222
energy	0.612243	0.603620	0.008623
speechiness	0.119282	0.070163	0.049119
acousticness	0.266987	0.286026	0.019039
instrumentalness	0.012123	0.009477	0.002645
liveness	0.179766	0.161449	0.018317
valence	0.522467	0.441172	0.081295

In [66]:
```
grader.check("q4_4")
```

Out[66]: **q4_4** passed!

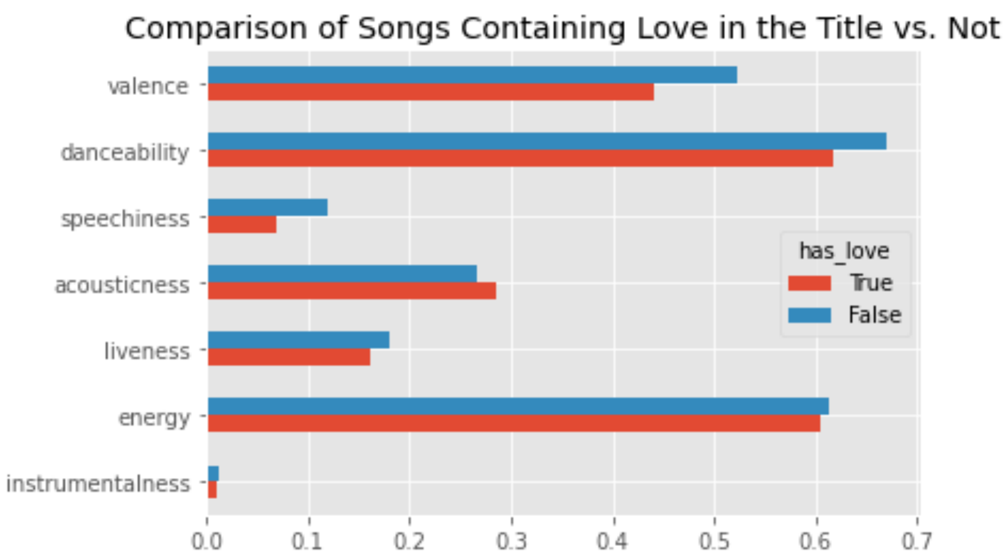
Question 4.5. Using `transposed_love` , create a horizontal bar chart comparing the mean values of each of the 7 audio features for songs with and without `'love'` in the `'track_name'` . Your bar chart should have 14 bars total, 7 for songs with `'love'` and 7 for songs without `'love'` .

Use the `'AbsDiff'` column to arrange the bars in the chart such that the audio feature which is most affected by the presence of the word `'love'` appears at the top and the one that's least affected is at the bottom.

Title your chart `'Comparison of Songs Containing Love in the Title vs. Not'` .

In [67]:
```
# Make your horizontal bar chart here.
transposed_love.sort_values(by='AbsDiff', ascending = True).get(['True','False']).plot(kind='barh', title = 'Comparison of Songs Containing Love in the Title vs. Not')
```

Out[67]:
```
<AxesSubplot:title={'center':'Comparison of Songs Containing Love in the Title vs. Not'}>
```



Question 4.6. ⭐⭐ Let's generalize this analysis to any word, not just `'love'` . Define a function called `word_analysis` that takes two arguments:

- `word` , which can be any word that appears in at least one `'track_name'` . The input word can be capitalized any way; the function should not be case sensitive.
- `draw_plot` , which should be a Boolean value corresponding to whether or not a bar chart should be drawn. By setting `draw_plot=False` in the parameter list in the function definition, we make `draw_plot` an optional argument whose default value is `False` . If not `draw_plot` is not specified

by the caller of the function, the function will not draw the plot.

If `draw_plot` is `True` , this function should produce a horizontal bar chart similar to the one you produced in the last question, except it will group songs based on whether or not their `'track_name'` contains the input word (as opposed to `'love'`). The bars should be ordered in the same way as described in the previous question, and the title of the plot should be of the same format, with just the first letter of the input word capitalized.

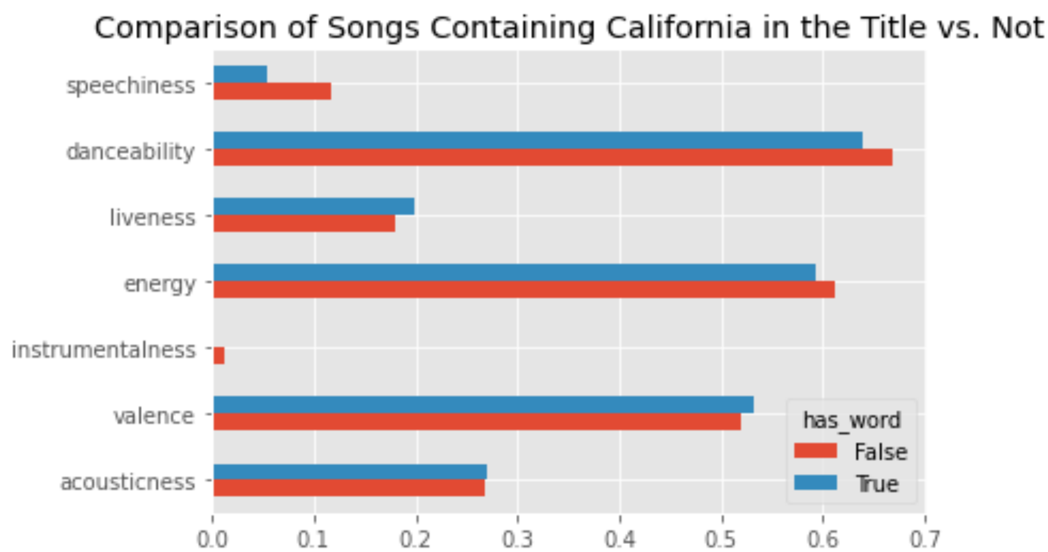
In all cases, `word_analysis` should return a DataFrame with 7 rows, in any order, representing the 7 audio features, and 3 columns:

- The `'False'` column should contain the mean values of all audio features, among songs that do not contain the given word in the `'track_name'` .
- The `'True'` column should contain the mean values of all audio features, among songs that do contain the given word in the `'track_name'` .
- `'AbsDiff'` should contain the absolute difference between the `'False'` and `'True'` columns.

For example, `word_analysis('CaliforNiA', True)` should return the following DataFrame:

has_word	False	True	AbsDiff
acousticness	0.267513	0.269535	0.002022
valence	0.520189	0.532000	0.011811
instrumentalness	0.012068	0.000125	0.011943
energy	0.612034	0.592500	0.019534
liveness	0.179226	0.199125	0.019899
danceability	0.668027	0.638750	0.029277
speechiness	0.118015	0.055175	0.062840

and display the following plot:



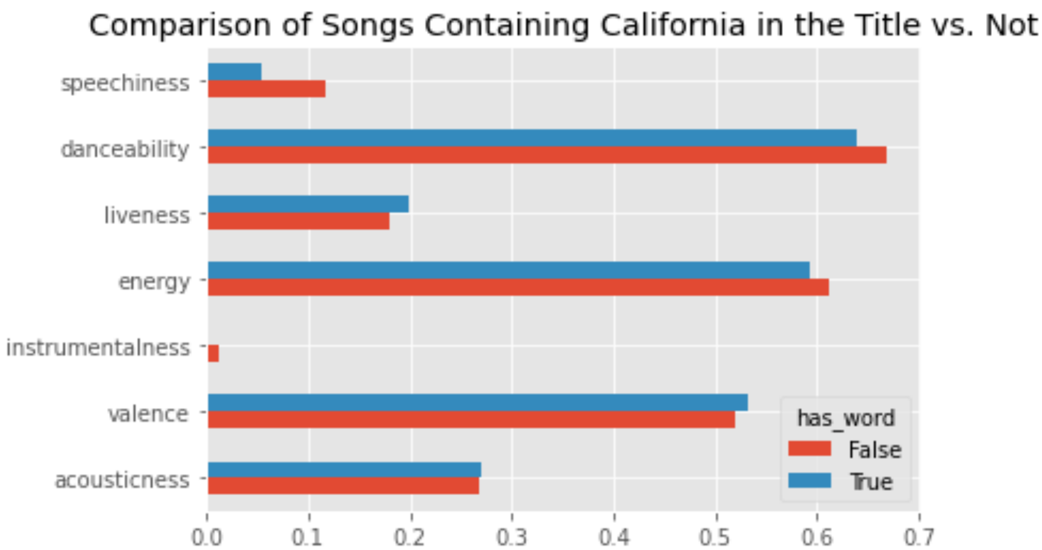
Note: Your function does not need to work on input words not in the title of some song in `songs` . For example, it's okay if `word_analysis('znvlox')` errors.

Hint: To make sure that the first letter of the input word is capitalized when setting the title of your plot, use one of the string methods [detailed here (https://docs.python.org/3/library/stdtypes.html#string-methods)](https://docs.python.org/3/library/stdtypes.html#string-methods).

In [68]:
```
o = np.vectorize(from_bool_to_str)
def word_analysis(word, draw_plot=False):
    g = np.array(songs.get('track_name').str.lower().str.contains(word
.lower()))
    d = songs.assign(has_word = o(g))
    s = d.groupby('has_word').mean().drop(columns= ['key', 'mode', 'lo
udness', 'tempo', 'duration_min'])
    a = s.T
    e = a.assign(AbsDiff = abs(a.get('False') - a.get('True'))).sort_v
alues(by='AbsDiff', ascending = True)
    if draw_plot==True:
        k = e.get(['False','True']).plot(kind='barh', title = 'Compari
son of Songs Containing ' + word.title() + ' in the Title vs. Not')
        return e
        return k
    elif draw_plot==False:
        return e

# Test out your function. Feel free to change these inputs.
word_analysis('CaliforNiA', True)
```

Out[68]:

has_word	False	True	AbsDiff
acousticness	0.267513	0.269535	0.002022
valence	0.520189	0.532000	0.011811
instrumentalness	0.012068	0.000125	0.011943
energy	0.612034	0.592500	0.019534
liveness	0.179226	0.199125	0.019899
danceability	0.668027	0.638750	0.029277
speechiness	0.118015	0.055175	0.062840



In [69]:

```
grader.check("q4_6")
```

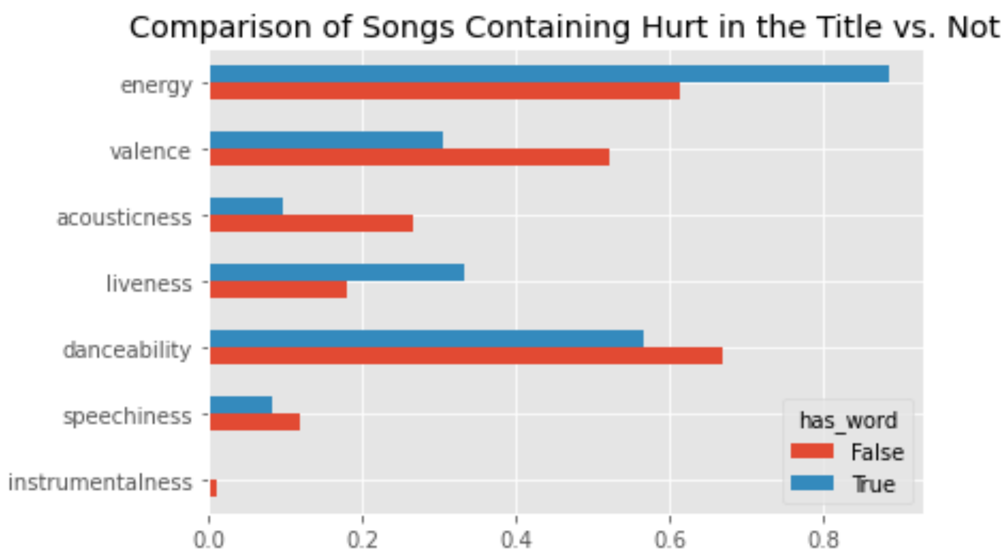
Out[69]: **q4_6** passed!

Make sure to run the cell below before submitting. **Do not edit or delete it**.

In [70]:
```
word_analysis('huRt', True)
```

Out[70]:

has_word	False	True	AbsDiff
instrumentalness	0.012054	0.0001	0.011954
speechiness	0.117931	0.0818	0.036131
danceability	0.668022	0.5660	0.102022
liveness	0.179196	0.3340	0.154804
acousticness	0.267583	0.0958	0.171783
valence	0.520292	0.3040	0.216292
energy	0.611896	0.8850	0.273104



Let's define the *polarity* of a word as the total absolute difference between the `'True'` and `'False'` columns, across all 7 audio features. If a word has high polarity, it means songs containing that word in the `'track_name'` are very musically different from songs without that word in the `'track_name'`. If a word has low polarity, it means songs containing that word and not containing that word in the `'track_name'` are musically similar.

Question 4.7. Define a function `polarity` that takes one input, a string representing a word that that appears in at least one `'track_name'` in `songs`, and returns the polarity of that word.

In [71]:
```
o = np.vectorize(from_bool_to_str)
def polarity(word):
    g = np.array(songs.get('track_name').str.lower().str.contains(word
.lower()))
    d = songs.assign(has_word = o(g))
    s = d.groupby('has_word').mean().drop(columns= ['key', 'mode', 'lo
udness', 'tempo', 'duration_min'])
    a = s.T
    e = a.assign(AbsDiff = abs(a.get('False') - a.get('True'))).sort_v
alues(by='AbsDiff', ascending = True)
    r = e.get('AbsDiff').sum()
    return r

# Test out your function. Feel free to change the input.
polarity('hate')
```

Out[71]: 0.5100413712521734

In [72]:
```
grader.check("q4_7")
```

Out[72]: **q4_7** passed!

On its own, the polarity of a single word doesn't tell us much. Instead, we need to look at the polarities of several words and compare them, to see which words are more polarizing than others.

Run the cell below to load in an array of words.

In [73]:
```
polarity_words = np.array(['love', 'hate', 'miracle', 'dead', 'feel',
'hold', 'about', 'and', 'christmas',
                           'you', 'wonder', 'hello', 'work', 'hurt', '
where', 'quiero', 'shake', 'was'])
```

Question 4.8. Create an array called `polarity_words_ranked` containing the same words as `polarity_words` but ordered in descending order of polarity.

In [74]:
```
z = np.vectorize(polarity)
f = np.array([])
w = np.append(f, z(polarity_words))
column_values = ['polarity', 'words']
Df_pol = bpd.DataFrame(data = [w ,polarity_words] )
DfT = Df_pol.T

polarity_words_ranked = np.array(DfT.sort_values(by= 0, ascending = Fa
lse).get(1))
polarity_words_ranked
```

Out[74]:
```
array(['quiero', 'shake', 'miracle', 'hurt', 'wonder', 'work',
       'christmas', 'hate', 'dead', 'about', 'where', 'hold', 'was',
       'feel', 'hello', 'love', 'you', 'and'], dtype=object)
```

In [75]:
```
grader.check("q4_8")
```

Out[75]: **q4_8** passed!

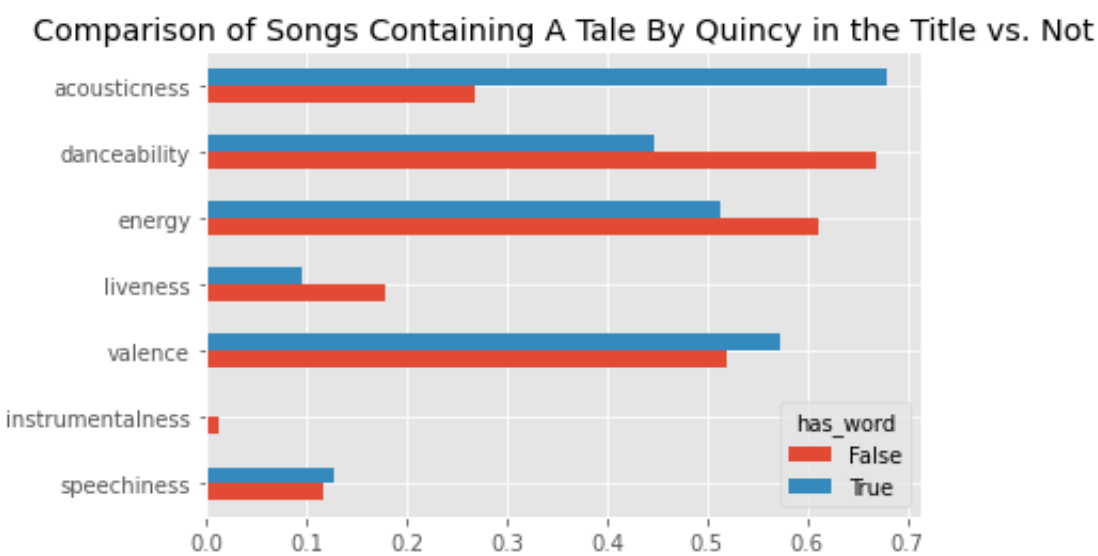
You may notice that very common words, like `'and'` and `'was'`, aren't very polarizing. See if you can come up with other words that are either very polarizing or very "neutral," relative to the words in the array above.

Before we conclude this section, let's stop and notice something we did inadvertently. It turns out we can use some of the analysis we've done here to see how individual songs compare to the rest of the songs in the dataset. For example, let's see how `'The Weeknd'`'s song `'A Tale By Quincy'` stacks up against the rest of the songs in the weekly top 200. Our `word_analysis` function should work even if we pass in phrases, so we can use it to compare songs with `'A Tale By Quincy'` in the title to songs without that string in the title. As you might expect, the only song in the dataset with `'A Tale By Quincy'` as part of the title is `'A Tale By Quincy'` itself.

In [76]:
```
word_analysis('A Tale By Quincy', True)
```

Out[76]:

has_word	False	True	AbsDiff
speechiness	0.117913	0.12700	0.009087
instrumentalness	0.012054	0.00003	0.012024
valence	0.520187	0.57200	0.051813
liveness	0.179289	0.09600	0.083289
energy	0.612042	0.51300	0.099042
danceability	0.668068	0.44700	0.221068
acousticness	0.267355	0.67900	0.411645



The resulting analysis shows, for example, that `'A Tale By Quincy'` is much more acoustic than a typical song on the weekly top 200. Run the cell below to listen for yourself and see if you agree.

In [77]:
```
play_spotify('spotify:track:759ndr57jb0URg4j9YSWml')
```

A Tale By Quincy

The Weeknd

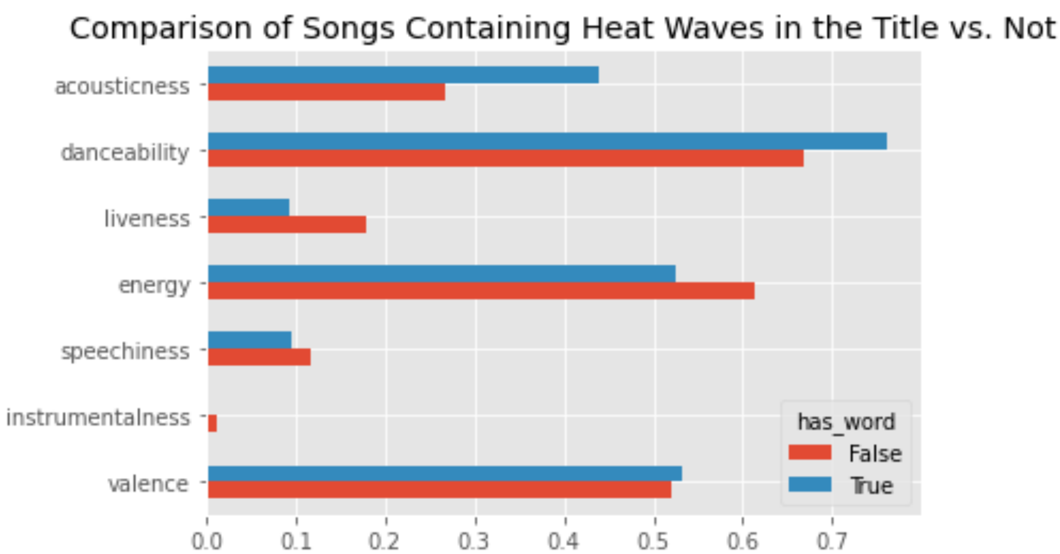
EMBED_PREVIEW E

Of course, we wouldn't be able to isolate specific songs by name like this if multiple artists have a song by the same name, or if the full song title is included in other song titles. But for most songs, this does work! Try it out on one of your favorite songs below to see what makes that song so special.

In [78]:
```
word_analysis('Heat Waves', True)
```

Out[78]:

has_word	False	True	AbsDiff
valence	0.520203	0.531000	0.010797
instrumentalness	0.012054	0.000007	0.012047
speechiness	0.117926	0.094400	0.023526
energy	0.612037	0.525000	0.087037
liveness	0.179291	0.092100	0.087191
danceability	0.667945	0.761000	0.093055
acousticness	0.267448	0.440000	0.172552



Section 5: The Test of Time ⌛

([return to the outline](#))

In the last three sections, we've worked with the audio features of songs. We haven't yet used any of the date information we have available – that is, we haven't looked at the `'week'` or `'release_date'` columns in `charts`. In this section, we'll switch our attention to these columns, to study how the "age" of top songs in `charts` has changed over time.

Run the cell below to load in the `charts` DataFrame again.

In [79]:
```
charts = bpd.read_csv('data/weekly_charts.csv')
charts
```

Out[79]:

	week	rank	track_name	uri	release_date	stream
0	2021-02-04	1	drivers license	spotify:track:7lPN2DXiMsVn7XUKtOW1CS	2021-01-08	205431
1	2021-02-04	2	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	91651
2	2021-02-04	3	Save Your Tears	spotify:track:5QO79kh1waicV47BqGRL3g	2020-03-20	86606
3	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
4	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
...	...	...	...	...	...	..
70178	2022-07-14	196	Get Into It (Yuh)	spotify:track:0W6I02J9xcqK8MtSeosEXb	2021-06-25	159210
70179	2022-07-14	197	Fancy Like	spotify:track:58UKC45GPNTflCN6nwCUeF	2022-01-21	159012
70180	2022-07-14	198	Stick Season	spotify:track:0GNVXNz7Jkicfk2mp5OyG5	2022-07-08	158330
70181	2022-07-14	199	Call Out My Name	spotify:track:09mEdoA6zrmBPgTEN5qXmN	2018-03-30	158323
70182	2022-07-14	200	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	157921

70183 rows × 24 columns

From the DataFrame preview, it looks like `'week'` and `'release_date'` are given as strings in `'YYYY-MM-DD'` format. Unfortunately, some tracks have an incomplete `'release_date'`, in the form `'YYYY'` or `'YYYY-MM'`.

Question 5.1. What proportion of the rows of `charts` have a `'release_date'` of the form `'YYYY'`, with just a year? Save your result as `year_only`. Similarly, what proportion of the rows of `charts` have a `'release_date'` of the form `'YYYY-MM'`, with just a year and month? Save your result as `year_month_only`.

In [80]:
```
def date_len(x):
    e = len(x)
    return e

t = charts.assign(date_length = charts.get('release_date').apply(date_len))
year_only = t[t.get('date_length') == 4].shape[0] / charts.shape[0]
year_month_only = t[t.get('date_length') == 7].shape[0] / charts.shape[0]

print("The proportion of songs in `charts` that have a release_date in the form 'YYYY' is " + str(round(year_only, 7)) + ".")
print("The proportion of songs in `charts` that have a release_date in the form 'YYYY-MM' is " + str(round(year_month_only, 7)) + ".")
```

```
The proportion of songs in `charts` that have a release_date in the form 'YYYY' is 0.0143909.
The proportion of songs in `charts` that have a release_date in the form 'YYYY-MM' is 0.0004844.
```

In [81]:
```
grader.check("q5_1")
```

Out[81]: **q5_1** passed!

For consistency, let's input the missing months and days where necessary, so that all dates in `charts` will be in the same format. We don't actually know when these songs were released, so we'll just choose to handle the missing months and days by replacing them with `'01'`. That is, if a song has just a year listed for its `'release_date'`, we'll assume it was released on January 1st of that year. Similarly, if a song has just a year and month listed, we'll assume it was released on the first of that month.

Question 5.2. Replace the missing months and days in the `'release_date'` column of `charts` with `'01'` as described.

In [82]:
```python
def add_date(x):
    if len(x) == 4:
        return x + '-01-01'
    elif len(x) == 7:
        return x + '-01'
    else:
        return x


charts = charts.assign(release_date = charts.get('release_date').apply
(add_date))
charts
```

Out[82]:

	week	rank	track_name	uri	release_date	stream
0	2021-02-04	1	drivers license	spotify:track:7lPN2DXiMsVn7XUKtOW1CS	2021-01-08	205431
1	2021-02-04	2	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	91651
2	2021-02-04	3	Save Your Tears	spotify:track:5QO79kh1waicV47BqGRL3g	2020-03-20	86606
3	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
4	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	82478
...	...	...	...	...	...	..
70178	2022-07-14	196	Get Into It (Yuh)	spotify:track:0W6I02J9xcqK8MtSeosEXb	2021-06-25	159210
70179	2022-07-14	197	Fancy Like	spotify:track:58UKC45GPNTflCN6nwCUeF	2022-01-21	159012
70180	2022-07-14	198	Stick Season	spotify:track:0GNVXNz7Jkicfk2mp5OyG5	2022-07-08	158330
70181	2022-07-14	199	Call Out My Name	spotify:track:09mEdoA6zrmBPgTEN5qXmN	2018-03-30	158323
70182	2022-07-14	200	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	157921

70183 rows × 24 columns

In [83]:
```python
grader.check("q5_2")
```

Out[83]: **q5_2** passed!

Question 5.3. Find the song in `charts` with the earliest `'release_date'`. Save the name of this song to `oldest_song` and save the `'artist_names'` associated with this song to `oldest_song_artists`.

This song has stood the test of time – you'll see why!

In [84]:
```
oldest_song = charts.sort_values(by='release_date', ascending =True).get('track_name').iloc[0]
oldest_song_artists = charts.sort_values(by='release_date', ascending =True).get('artist_names').iloc[0]

print('The oldest song in `charts` is ' + oldest_song + ' by ' + oldest_song_artists)
```

```
The oldest song in `charts` is White Christmas by Bing Crosby, Ken Darby Singers, John Scott Trotter & His Orchestra
```

In [85]:
```
grader.check("q5_3")
```

Out[85]: **q5_3** passed!

Let’s try to calculate the time between when this song was first released and when this song was in the weekly top 200 most recently. To tackle this problem and others like it, we'll write a general function to calculate the time between any two dates.

Question 5.4. ⭐⭐ Complete the implementation of the function `weeks_between` , which takes in two dates as lists in the form `[year, month, day]` and returns the number of **full weeks** between the two dates. You may assume the second date comes after the first.

Here, we'll define a full week as 7 days. For example, if there are 200 days between two dates, we'd say there are 28 *full* weeks between the two dates, since $\frac{200}{7} = 28.571$.

Example behavior is given below.

```
# There are 11 days between March 14th, 2022 and March 25th, 2022.
# This corresponds to 1 full week.
>>> weeks_between([2022, 3, 14], [2022, 3, 25])
1

# There are 805 days between November 26th, 1998 and February 9th, 2001, n
ot counting leap year days.
# This corresponds to 115 full weeks.
>>> weeks_between([1998, 11, 26], [2001, 2, 9])
115
```

To help you, we've provided a function called `days_between` and a [video walkthrough of how it works (https://www.youtube.com/watch?v=6HOAk0GAqKU)](https://www.youtube.com/watch?v=6HOAk0GAqKU). Make sure you understand what this function does and how it works, because you'll want to make use of it inside `weeks_between` .

Note: **Don't factor in leap years** for the purposes of this question. We'll assume that every year has 365 days.

In [86]:
```
# Run this cell to view the walkthrough video.
YouTubeVideo('6HOAk0GAqKU')
```

Out[86]:

In [87]:
```
# This function is provided. Watch the walkthrough video to understand
what it does and how it works.
def days_between(month1, day1, month2, day2):

    # days_per_month[1] is the number of days in January, days_per_mon
th[8] is the number of days in August, etc.
    days_per_month = np.array([0, 31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31])

    # Case where both months are the same.
    if month1 == month2:
        return day2 - day1

    else:
        total_days = 0

        # First, figure out the number of days left in month1.
        total_days = total_days + days_per_month[month1] - day1

        # Then, add the number of days in the full months between mont
h1 and month2.
        for full_month in np.arange(month1 + 1, month2):
            total_days = total_days + days_per_month[full_month]

        # Then, add the number of days so far in month2.
        total_days = total_days + day2

        return total_days
```

We've already provided an outline for what you need to do in `weeks_between` ; your job is to fill in the the missing pieces.

In [88]:
```
def weeks_between(date1, date2):
    # Store the year, day, and month for each date separately as ints.
    year1 = date1[0]
    month1 = date1[1]
    day1 = date1[2]
    year2 = date2[0]
    month2 = date2[1]
    day2 = date2[2]

    # Main idea: Find the total number of days between the two dates,
then divide that by 7 and round down.
    total_days = 0

    # Case 1: The dates are in the same year.
    if year1 == year2:
        # Calculate the number of days between them.
        total_days = days_between(month1, day1, month2, day2)

    # Case 2: The dates are in different years.
    else:
        # Add 365 for each FULL year between the dates.
        for full_year in np.arange(year1 + 1 , year2):
            total_days = total_days + 365

        # Add the number of days between date1 and the end of year1.
        total_days = total_days + days_between(month1, day1, 12, 31)

        # Add the number of days between the start of year2 and date2.
        total_days = total_days + days_between(1, 1, month2, day2)

        # Add the number of days between December 31st and January 1st
(1).
        total_days = total_days + 1

    # Convert to weeks and round down
    return int(total_days / 7)
```

In [89]:
```
grader.check("q5_4")
```

Out[89]: **q5_4** passed!

Now that we have a function that can compute the number of weeks between any two dates, we can calculate the time between when `oldest_song` was first released and when it was in the weekly top 200 most recently.

Unfortunately, the dates in the `'release_date'` and `'week'` columns of `charts` are not lists in the form `[year, month, day]`, but are strings of the form `'YYYY-MM-DD'`. They need to be transformed before they can be used as input to `weeks_between`.

We've done that work for you in the `convert_date_to_list` function below. It converts an input `date_str` of the form `'1998-11-26'` to a list of the form `[1998, 11, 26]`. Step by step, here's what it does:

1. Splits `date_str` by `'-'`.
   - This takes `'1998-11-26'` and turns it into the list of strings `['1998', '11', '26']`.
2. Converts the list of strings into an array, and converts the data type of each element to an `int`.
   - This takes `['1998', '11', '26']` and turns it into `np.array([1998, 11, 26])`.
3. Converts the array to a list and returns it.
   - The function returns the list `[1998, 11, 26]`.

In [90]:
```python
def convert_date_to_list(date_str):
    return list(np.array(date_str.split('-')).astype(int))

convert_date_to_list('1998-11-26')
```

Out[90]: `[1998, 11, 26]`

Question 5.5. Calculate the time between the following two dates, in weeks:

1. The release date of `oldest_song` by `oldest_song_artists`.
2. The most recent time in our dataset that `oldest_song` by `oldest_song_artists` was in the weekly top 200.

Store the result in `weeks_since_release`.

Hint: It's a good idea to check if your answer makes sense given the `'release_date'` of `oldest_song`.

In [91]:
```
u = charts[(charts.get('track_name') == 'White Christmas') & (charts.get('artist_names') == 'Bing Crosby, Ken Darby Singers, John Scott Trotter & His Orchestra')].sort_values(by='week', ascending=False)
y = u.get('week').iloc[0]
v = convert_date_to_list(y)
b = u.get('release_date').iloc[0]
m = convert_date_to_list(b)
```

In [92]:
```
weeks_since_release = weeks_between(m , v)
weeks_since_release
```

Out[92]: 4171

In [93]:
```
grader.check("q5_5")
```

Out[93]: **q5_5** passed!

Since `weeks_between` is general enough to compute the number of weeks between any two dates, let's use it on the full `'release_date'` and `'week'` columns of `charts`, so that we can see how old each song was every time it was in the weekly top 200.

Unfortunately, the `.apply` method [as we learned it in class (https://dsc10.com/resources/lectures/lec09/lec09.html#.apply)](https://dsc10.com/resources/lectures/lec09/lec09.html#.apply) is a **Series** method, and it only works with functions of one argument. Here, `weeks_between` takes two arguments – specifically, two lists.

It turns out there's another version of `.apply` that works for **DataFrames**, and it works with functions of multiple arguments. Today is really your lucky day - we have implemented all the necessary code below!

The function `weeks_between_wrapper` takes in a single row of a DataFrame, and calls `'weeks_between'` on the `'release_date'` and `'week'` entries of the row. We haven't worked too much with rows of DataFrames, so you don't need to understand how this code works.

In [94]:
```
def weeks_between_wrapper(row):

    release_date = row.get('release_date')
    current_week = row.get('week')

    date1 = convert_date_to_list(release_date)
    date2 = convert_date_to_list(current_week)

    return weeks_between(date1, date2)
```

Now, we'll use `.apply` with the `weeks_between_wrapper` function to determine how old each song on the charts was, at each time it was on the charts! The `axis=1` keyword argument in the line below is telling Python to use `weeks_between_wrapper` on each **row** of `charts` .

In [95]:
```
weeks_old = charts.apply(weeks_between_wrapper, axis=1)
weeks_old
```

Out[95]:
```
0          3
1          5
2         45
3         27
4         27
        ...
70178     54
70179     24
70180      0
70181    223
70182     80
Length: 70183, dtype: int64
```

Let's assign this Series back to the `charts` DataFrame. We'll call the resulting DataFrame `charts_with_ages` .

In [96]:
```
charts_with_ages = charts.assign(weeks_old=weeks_old)
charts_with_ages
```

Out[96]:

	week	rank	track_name	uri	release_date	stream
0	2021-02-04	1	drivers license	spotify:track:7lPN2DXiMsVn7XUKtOW1CS	2021-01-08	2054319
1	2021-02-04	2	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	916510
2	2021-02-04	3	Save Your Tears	spotify:track:5QO79kh1waicV47BqGRL3g	2020-03-20	866067
3	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	824789
4	2021-02-04	4	Mood (feat. iann dior)	spotify:track:3tjFYV6RSFtuktYl3ZtYcq	2020-07-24	824789
...	...	...	...	...	...	..
70178	2022-07-14	196	Get Into It (Yuh)	spotify:track:0W6I02J9xcqK8MtSeosEXb	2021-06-25	1592100
70179	2022-07-14	197	Fancy Like	spotify:track:58UKC45GPNTflCN6nwCUeF	2022-01-21	1590120
70180	2022-07-14	198	Stick Season	spotify:track:0GNVXNz7Jkicfk2mp5OyG5	2022-07-08	1583302
70181	2022-07-14	199	Call Out My Name	spotify:track:09mEdoA6zrmBPgTEN5qXmN	2018-03-30	1583235
70182	2022-07-14	200	Good Days	spotify:track:3YJJjQPAbDT7mGpX3WtQ9A	2020-12-25	1579212

70183 rows × 25 columns

Question 5.6. Create a DataFrame named `top_us` , with one row for each week of data collection, indexed and sorted by `'week'` . The `top_us` DataFrame should have columns called `'track_name'` , `'artist_names'` , and `'release_date'` , containing the relevant information for the **top-ranked (number 1) song each week in the United States**, along with a column called `'weeks_old'` that contains the **age of the song in weeks at that time**.

For instance, the song `'drivers license'` by `'Olivia Rodrigo'` was the top song in the US for the first two weeks of data collection, `'2021-02-04'` and `'2021-02-11'` , so this song should appear in the first two rows of `top_us` . The only difference between the first two rows, other than their indexes, is their values in the `'weeks_old'` column. Since `'drivers license'` was 3 weeks old on `'2021-02-04'` and 4 weeks old on `'2021-02-11'` , `top_us.get('weeks_old').iloc[0]` should be 3 and `top_us.get('weeks_old').iloc[1]` should be 4.

In [97]:
```
top_us = (charts_with_ages[(charts_with_ages.get('rank') == 1) & (charts_with_ages.get('country') == 'United States')]
          .groupby(['week','track_name', 'artist_names', 'release_date','weeks_old', 'rank'])
          .count().reset_index().get(['week','track_name', 'artist_names', 'release_date','weeks_old'])
          .sort_values(by='week').set_index('week'))
top_us
```

Out[97]:

week	track_name	artist_names	release_date	weeks_old
2021-02-04	drivers license	Olivia Rodrigo	2021-01-08	3
2021-02-11	drivers license	Olivia Rodrigo	2021-01-08	4
2021-02-18	Calling My Phone	Lil Tjay, 6LACK	2021-02-12	0
2021-02-25	drivers license	Olivia Rodrigo	2021-01-08	6
2021-03-04	drivers license	Olivia Rodrigo	2021-01-08	7
...	...	...	...	...
2022-06-16	Running Up That Hill (A Deal With God) - 2018 ...	Kate Bush	1985-01-01	1953
2022-06-23	Glimpse of Us	Joji	2022-06-10	1
2022-06-30	Glimpse of Us	Joji	2022-06-10	2
2022-07-07	Running Up That Hill (A Deal With God) - 2018 ...	Kate Bush	1985-01-01	1956
2022-07-14	Running Up That Hill (A Deal With God) - 2018 ...	Kate Bush	1985-01-01	1957

76 rows × 4 columns

In [98]:
```
grader.check("q5_6")
```

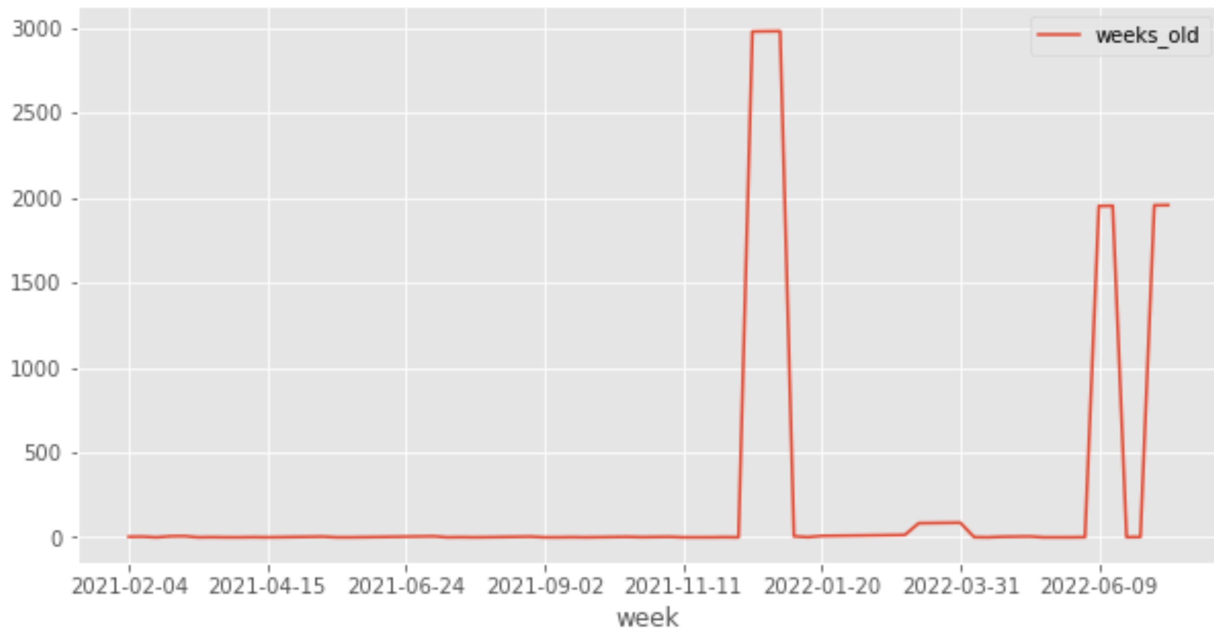
Out[98]: **q5_6** passed!

Let's try to visualize the age of the number 1 song on the US charts each week. However, before we start plotting, there's something we should take into consideration: look at the values in the `'weeks_old'` column in the preview above. Some are relatively small, like 3 or 4, but some are really large, like 1957! Let's see what happens when we plot such a wide range of values together on the same axes.

Question 5.7. Make a line plot that shows the age of the top song on the US charts over time, throughout the period of data collection. Use the argument `figsize=(10, 5)` so you can read the horizontal axis.

In [99]:
```
# Make your line plot here.
top_us.plot(kind='line', figsize=(10,5), y='weeks_old')
```

Out[99]: `<AxesSubplot:xlabel='week'>`



Since some songs are thousands of weeks old, plotting all the data together makes it hard to tell what the trends are for newer songs. To better see what's going on near 0 on the y-axis, we'll "clip", or chop off, the y-axis so that the oldest songs appear to only be 26 weeks (half a year) old.

We've done this for you. The plot we created is interactive, meaning that you can hover over any point on the line to see various pieces of information for each song. Try hovering over the line plot produced to see which songs were at the top of the charts each week, and how old they were when they got usurped by the next best thing.

Access the plot by clicking here. (https://dsc-courses.github.io/dsc10-2022-fa/resources/midterm_project/q5-age-number-1.html)

To test your understanding, see if you can answer these questions from the interactive plot:

1. Why does the line plot shows a bunch of diagonal line segments?
2. Why do some diagonal line segments start at the horizontal axis, and others don't?
3. Why does the line plot you made in Question 5.7 have three large spikes, while this one has four?
4. How many different #1 songs were over 26 weeks old? Why did these songs become super popular? (You may have to do some research.)

You don't have to turn in your answers to the questions above, but you should figure out how to answer them.

Section 6: Party in the USA 💃

(return to the outline)

We concluded Section 5 by looking at the age of the #1 song each week in the US. Let's continue our analysis of the songs that became extremely popular in the US.

Define a **US megahit** to be a song that has met all of the following criteria **in the US**:

- Has been at position 1 or 2 in the top 200 at some point.
- Spent at least 20 weeks in the top 200.
- Had a streak of at least 5 consecutive weeks of being in the top 10.

In this section, we'll work towards determining which songs fit this criteria, and in the next (and final!) section, we'll see how these songs stand apart from the rest musically.

Question 6.1. To start, create a DataFrame called `us_charts` with only one row for each week and each rank. That is, remove duplicate entries for songs with multiple artists. Keep only the `'track_name'`, `'artist_names'`, `'rank'`, and `'week'` columns, in that order.

Arrange the rows chronologically by week, and within each week, in ascending order of rank. Don't forget that we're only using data from the US.

Hint: `us_charts` should have a multiple of 200 rows, since there are 200 songs on the top 200 each week.

In [100]:
```
us_charts = (charts[charts.get('country') == 'United States']
             .groupby(['week','rank','track_name', 'artist_names']).co
unt().reset_index()
             .get(['track_name','artist_names','rank','week']).sort_va
lues(by=['week','rank'], ascending =True))
us_charts
```

Out[100]:

	track_name	artist_names	rank	week
0	drivers license	Olivia Rodrigo	1	2021-02-04
1	Good Days	SZA	2	2021-02-04
2	Streets	Doja Cat	3	2021-02-04
3	Save Your Tears	The Weeknd	4	2021-02-04
4	Whoopty	CJ	5	2021-02-04
...	...	...	...	...
15195	Get Into It (Yuh)	Doja Cat	196	2022-07-14
15196	Fancy Like	Walker Hayes	197	2022-07-14
15197	Stick Season	Noah Kahan	198	2022-07-14
15198	Call Out My Name	The Weeknd	199	2022-07-14
15199	Good Days	SZA	200	2022-07-14

15200 rows × 4 columns

In [101]:
```
grader.check("q6_1")
```

Out[101]: **q6_1** passed!

Question 6.2. How many distinct weeks was data collected for? Store your answer as an `int` in the variable `num_weeks`.

In [102]:
```
num_weeks = int(us_charts.shape[0] / 200)
num_weeks
```

Out[102]: 76

In [103]:
```
grader.check("q6_2")
```

Out[103]: **q6_2** passed!

Question 6.3. Rather than have the week listed as a date, we'd like to simply record it as a week number, between 1 and `num_weeks` (inclusive). For instance, since `'2021-02-18'` is the third week for which we have charts data, it is week number 3.

Add a column called `'week_num'` to `us_charts` that contains the week number for each week.

Hint: With the functions [`np.repeat` (https://numpy.org/doc/stable/reference/generated/numpy.repeat.html)](https://numpy.org/doc/stable/reference/generated/numpy.repeat.html) and `np.arange`, you can do this in one line of code.

In [104]:
```
us_charts = us_charts.assign(week_num = np.repeat(np.arange(1, num_wee
ks +1 , 1), 200))
us_charts
```

Out[104]:

	track_name	artist_names	rank	week	week_num
0	drivers license	Olivia Rodrigo	1	2021-02-04	1
1	Good Days	SZA	2	2021-02-04	1
2	Streets	Doja Cat	3	2021-02-04	1
3	Save Your Tears	The Weeknd	4	2021-02-04	1
4	Whoopty	CJ	5	2021-02-04	1
...	...	...	...	...	...
15195	Get Into It (Yuh)	Doja Cat	196	2022-07-14	76
15196	Fancy Like	Walker Hayes	197	2022-07-14	76
15197	Stick Season	Noah Kahan	198	2022-07-14	76
15198	Call Out My Name	The Weeknd	199	2022-07-14	76
15199	Good Days	SZA	200	2022-07-14	76

15200 rows × 5 columns

In [105]:
```
grader.check("q6_3")
```

Out[105]: **q6_3** passed!

Question 6.4. Our first criteria for a US megahit was that the song has been at position 1 or 2 in the top 200 in the United States at some point. Create an array of the `'track_name'` s of all such songs, without duplicates, and save it as `been_top_two` .

In [106]:
```
been_top_two = us_charts[(us_charts.get('rank') == 1) | (us_charts.get
('rank') == 2)].get('track_name').unique()
been_top_two
```

Out[106]:
```
array(['drivers license', 'Good Days', 'Save Your Tears',
       'Calling My Phone', 'What’s Next',
       'Wants and Needs (feat. Lil Baby)',
       'Peaches (feat. Daniel Caesar & Giveon)', 'As I Am (feat. Kha
lid)',
       'MONTERO (Call Me By Your Name)', 'RAPSTAR',
       'Kiss Me More (feat. SZA)', 'good 4 u',
       'm y . l i f e (with 21 Savage & Morray)', 'deja vu',
       'STAY (with Justin Bieber)', 'INDUSTRY BABY (feat. Jack Harlo
w)',
       'Hurricane', 'Girls Want Girls (with Lil Baby)',
       'Way 2 Sexy (with Future & Young Thug)',
       'Knife Talk (with 21 Savage ft. Project Pat)', 'Easy On Me',
       'One Right Now (with The Weeknd)',
       "All Too Well (10 Minute Version) (Taylor's Version) (From Th
e Vault)",
       'Smokin Out The Window', 'I Hate U',
       "Rockin' Around The Christmas Tree",
       'All I Want for Christmas Is You', 'Jingle Bell Rock',
       "We Don't Talk About Bruno", 'Heat Waves', 'Sacrifice',
       'pushin P (feat. Young Thug)', 'Super Gremlin', 'As It Was',
       'First Class', 'Moscow Mule', 'N95', 'Die Hard',
       'Late Night Talking',
       'Running Up That Hill (A Deal With God) - 2018 Remaster',
       'Glimpse of Us', 'Jimmy Cooks (feat. 21 Savage)'], dtype=obje
ct)
```

In [107]:
```
grader.check("q6_4")
```

Out[107]: **q6_4** passed!

Below, we check that none of the songs in `been_top_two` have the same `'track_name'` but different `'artist_names'` as another song in `us_charts`.

In [108]:
```
# You don't need to edit this code, but you should understand how it w
orks.
def diff_artists(track_name):
    '''Return the number of distinct 'artist_names' associated with a
given track_name in us_charts.'''
    song_only = us_charts[us_charts.get('track_name') == track_name]
    return song_only.groupby('artist_names').count().shape[0]

num_diff_artists = np.array([])
for song in been_top_two:
    num_diff_artists = np.append(num_diff_artists, diff_artists(song))
max(num_diff_artists)
```

Out[108]:
```
1.0
```

Since this set of songs doesn't have the potential for confusion with other songs with the same `'track_name'`, we can safely refer to these songs by their `'track_name'` for the remainder of this section (instead of having to also worry about their `'artist_names'`).

Question 6.5. Create a DataFrame called `possibly_mega` with the same columns as `us_charts`, but with only the rows of `us_charts` where the `'track_name'` is in `been_top_two`.

Hints:

- Add a new column to filter by, then drop it after filtering (i.e. after Boolean indexing).
- Use the Python keyword `in` to determine whether a specific song name is in `been_top_two`.

In [109]:
```
q = np.array(us_charts.get('track_name'))
ans = np.array([])
for x in q:
    if x in been_top_two:
        ans = np.append(ans, 'True')
    else:
        ans = np.append(ans, 'False')
r = us_charts.assign(boolean= ans)
i = r[r.get('boolean') == 'True'].drop(columns='boolean')
possibly_mega = i
possibly_mega
```

Out[109]:

	track_name	artist_names	rank	week	week_num
0	drivers license	Olivia Rodrigo	1	2021-02-04	1
1	Good Days	SZA	2	2021-02-04	1
3	Save Your Tears	The Weeknd	4	2021-02-04	1
25	Heat Waves	Glass Animals	26	2021-02-04	1
200	drivers license	Olivia Rodrigo	1	2021-02-11	2
...	...	...	...	...	...
15137	drivers license	Olivia Rodrigo	138	2022-07-14	76
15151	RAPSTAR	Polo G	152	2022-07-14	76
15163	One Right Now (with The Weeknd)	Post Malone, The Weeknd	164	2022-07-14	76
15170	Save Your Tears	The Weeknd	171	2022-07-14	76
15199	Good Days	SZA	200	2022-07-14	76

1395 rows × 5 columns

In [110]:
```
grader.check("q6_5")
```

Out[110]: **q6_5** passed!

Question 6.6. Our second criteria for a US megahit was that the song spent at least 20 weeks on the top 200 in the US.

Create a function called `calculate_weeks` that takes as input the `'track_name'` of a song in `possibly_mega` and returns the number of weeks the song spent on the top 200 charts in the US (during the period of data collection). Then `apply` the function to the `possibly_mega` DataFrame and add a column to `possibly_mega` called `'weeks_on_charts'` with this information.

In [111]:
```
q = us_charts.groupby('track_name').size()
def calculate_weeks(track_name):
    t = q.loc[track_name]
    return t

possibly_mega = possibly_mega.assign(weeks_on_charts = possibly_mega.g
et('track_name').apply(calculate_weeks))
possibly_mega
```

Out[111]:

	track_name	artist_names	rank	week	week_num	weeks_on_charts
0	drivers license	Olivia Rodrigo	1	2021-02-04	1	76
1	Good Days	SZA	2	2021-02-04	1	71
3	Save Your Tears	The Weeknd	4	2021-02-04	1	74
25	Heat Waves	Glass Animals	26	2021-02-04	1	76
200	drivers license	Olivia Rodrigo	1	2021-02-11	2	76
...	...	...	...	...	...	...
15137	drivers license	Olivia Rodrigo	138	2022-07-14	76	76
15151	RAPSTAR	Polo G	152	2022-07-14	76	66
15163	One Right Now (with The Weeknd)	Post Malone, The Weeknd	164	2022-07-14	76	36
15170	Save Your Tears	The Weeknd	171	2022-07-14	76	74
15199	Good Days	SZA	200	2022-07-14	76	71

1395 rows × 6 columns

In [112]:
```
grader.check("q6_6")
```

Out[112]: **q6_6** passed!

Our third second criteria for a US megahit was that the song had a streak of at least 5 consecutive weeks of being in the top 10 on the charts in the US.

In order to identify these songs, we'll need to be able to calculate, for a given song, the longest streak of consecutive weeks spent in the top 10. The next few questions will help us get there.

Question 6.7. ⭐⭐ Write a function called `calculate_rank_array` that takes as input the `'track_name'` of a song in `possibly_mega` and returns an array of that song's ranks for each week of data collection. The array should be of length `num_weeks` for every possible input song, regardless of whether the song actually appeared in the top 200 for all weeks. If the song is not on the chart in a given week, substitute 201 for its rank that week.

For example, `'As It Was'` by `'Harry Styles'` first appeared in the top 200 in week 62. As a result, the first 61 elements of `calculate_rank_array('As It Was')` should be 201. In weeks 62 through 65, it was at positions 1, 2, 1, and 1, so those should be the next four elements in `calculate_rank_array('As It Was')`. The full expected output of `calculate_rank_array('As It Was')` is given below.

```
>>> calculate_rank_array('As It Was')
array([201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 201.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 201.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 201.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 201.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 201.,
       201., 201., 201., 201., 201., 201.,   1.,   2.,   1.,   1.,   1.,
         3.,   5.,   1.,   1.,   2.,   3.,   3.,   4.,   3.,   3.])
```

Hint: Our solution uses a `for` -loop and the `in` keyword.

In [113]:
```
def calculate_rank_array(track_name):
    week_number = np.arange(1,77)
    j = np.array([])
    q = np.array(us_charts[us_charts.get('track_name') == track_name].
get('week_num'))
    for i in week_number:
        if i in q:
            e = us_charts[(us_charts.get('track_name') == track_name)
& (us_charts.get('week_num') == i)].get('rank').iloc[0]
            j = np.append(j, e)
        else:
            j = np.append(j, 201)
    return j


# Test out your function. Feel free to change this input.
calculate_rank_array('As It Was')
```

Out[113]:
```
array([201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 2
01.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 2
01.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 2
01.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 2
01.,
       201., 201., 201., 201., 201., 201., 201., 201., 201., 201., 2
01.,
       201., 201., 201., 201., 201., 201.,   1.,   2.,   1.,   1.,
1.,
         3.,   5.,   1.,   1.,   2.,   3.,   3.,   4.,   3.,   3.])
```

In [114]:
```
grader.check("q6_7")
```

Out[114]: **q6_7** passed!

Question 6.8. Now, write a function called `longest_streak` that takes two inputs:

- `track_name` , the `'track_name'` of a song in `possibly_mega` .
- `n` , an integer between 1 and 200 (inclusive). By setting `n=10` in the parameter list, we make `n` an optional argument with 10 as its default value if omitted.

The function should return the largest number of consecutive weeks for which the given song ranked in the top `n` songs in the US.

For example, `longest_streak('As It Was', 3)` should evaluate to 6 because the song `'As It Was'` had a 6-week streak of being in the top 3 in the US, and no longer streak.

Note: We've completed a good chunk of the implementation of `longest_streak` for you. A big part of your job is to understand what the role of each variable is. You only need to add the body of the `for` -loop; our solution only adds 5 lines to what is below.

In [115]:
```
def longest_streak(track_name, n=10):
    rank_array = calculate_rank_array(track_name)
    longest = 0
    current = 0
    for num in rank_array:
        if num <= n:
            longest = longest + 1
        elif num > n:
            current = max(current,longest)
            longest = 0
    return max(longest, current)

# Test out your function. Feel free to change these inputs.
longest_streak('As It Was', 3)
```

Out[115]: 6

In [116]:
```
grader.check("q6_8")
```

Out[116]: **q6_8** passed!

Question 6.9. Add a column called `'longest_streak_top_ten'` to `possibly_mega` that contains, for each song, the longest number of consecutive weeks that the song spent in the top 10 in the US.

In [117]:
```
possibly_mega = possibly_mega.assign(longest_streak_top_ten = possibly
_mega.get('track_name').apply(longest_streak))
possibly_mega
```

Out[117]:

	track_name	artist_names	rank	week	week_num	weeks_on_charts	longest_streak_to
0	drivers license	Olivia Rodrigo	1	2021-02-04	1	76	
1	Good Days	SZA	2	2021-02-04	1	71	
3	Save Your Tears	The Weeknd	4	2021-02-04	1	74	
25	Heat Waves	Glass Animals	26	2021-02-04	1	76	
200	drivers license	Olivia Rodrigo	1	2021-02-11	2	76	
...	...	...	...	...	...	...	
15137	drivers license	Olivia Rodrigo	138	2022-07-14	76	76	
15151	RAPSTAR	Polo G	152	2022-07-14	76	66	
15163	One Right Now (with The Weeknd)	Post Malone, The Weeknd	164	2022-07-14	76	36	
15170	Save Your Tears	The Weeknd	171	2022-07-14	76	74	
15199	Good Days	SZA	200	2022-07-14	76	71	

1395 rows × 7 columns

In [118]:
```
grader.check("q6_9")
```

Out[118]: **q6_9** passed!

It took a lot of preparation, but now we can finally identify the songs that qualify as US megahits! As a reminder, we say a song is a US megahit if it has met all of the following criteria **in the US**:

- Has been at position 1 or 2 in the top 200 at some point.
- Spent at least 20 weeks in the top 200.
- Had a streak of at least 5 consecutive weeks of being in the top 10.

Question 6.10. Create a DataFrame called `us_megahits` that is indexed by `'track_name'`, has a single row for each song that qualifies as a US megahit, and has columns `'artist_names'`, `'weeks_on_charts'`, and `'longest_streak_top_ten'`.

In [119]:
```
us_megahits = (possibly_mega[(possibly_mega.get('weeks_on_charts') >=20) &
               (possibly_mega.get('longest_streak_top_ten') >= 5)]
 .groupby(['track_name','artist_names', 'weeks_on_charts', 'longest_streak_top_ten']).count().reset_index()
 .drop(columns=['rank', 'week', 'week_num'])).set_index('track_name')
us_megahits
```

Out[119]:

track_name	artist_names	weeks_on_charts	longest_streak_top_ten
Calling My Phone	Lil Tjay, 6LACK	43	9
Easy On Me	Adele	35	8
Good Days	SZA	71	7
Heat Waves	Glass Animals	76	19
INDUSTRY BABY (feat. Jack Harlow)	Lil Nas X, Jack Harlow	51	15
...	...	...	...
We Don't Talk About Bruno	Carolina Gaitán - La Gaita, Mauro Castillo, Ad...	28	14
deja vu	Olivia Rodrigo	67	9
drivers license	Olivia Rodrigo	76	9
good 4 u	Olivia Rodrigo	61	16
pushin P (feat. Young Thug)	Gunna, Future, Young Thug	22	10

19 rows × 3 columns

In [120]:
```
grader.check("q6_10")
```

Out[120]: **q6_10** passed!

Section 7: Encore 🔁

(return to the outline)

In this final section of the project, we'll analyze some of the audio features of US megahits. There's an issue, though: `us_megahits` doesn't contain any audio features. Fortunately, that information is available in `songs`, as we see below.

In [121]:
```
songs
```

Out[121]:

	track_name	artist_names	danceability	energy	key	mode	loudness	speechiness	acc
0	'Til You Can't	Cody Johnson	0.501	0.815	1.0	1.0	-4.865	0.0436	
1	'Till I Collapse	Eminem, Nate Dogg	0.548	0.847	1.0	1.0	-3.237	0.1860	
2	(Don't Fear) The Reaper	Blue Öyster Cult	0.333	0.927	9.0	0.0	-8.550	0.0733	
3	(Everybody's Waitin' For) The Man With The Bag...	Kay Starr	0.739	0.317	0.0	1.0	-8.668	0.0905	
4	(There's No Place Like) Home for the Holidays ...	Perry Como	0.478	0.341	5.0	1.0	-12.556	0.0511	
...	...	...	...	...	...	...	...	...	
2549	you broke me first	Tate McRae	0.667	0.373	4.0	1.0	-9.389	0.0500	
2550	¿Por Qué Me Haces Llorar?	Juan Gabriel	0.647	0.477	0.0	1.0	-8.157	0.0342	
2551	¿Quién Te Crees?	MC Davo, Calibre 50	0.747	0.780	9.0	0.0	-5.302	0.2160	
2552	Éxtasis	Millonario & W. Corona, Cartel De Santa	0.937	0.791	0.0	1.0	-5.242	0.0871	
2553	Índigo	Camilo, Evaluna Montaner	0.748	0.779	0.0	1.0	-6.659	0.0342	

2554 rows × 15 columns

Question 7.1. Create a DataFrame called `megahits` that contains the same rows and columns as `us_megahits`, plus the additional columns below.

- `'danceability'`
- `'energy'`
- `'key'`
- `'mode'`
- `'loudness'`
- `'speechiness'`
- `'acousticness'`
- `'instrumentalness'`
- `'liveness'`
- `'valence'`
- `'tempo'`
- `'duration_min'`

`megahits`, like `us_megahits`, should be indexed by `'track_name'`.

In [122]:
```
c = us_megahits.reset_index()
megahits = c.merge(songs, on='track_name').assign(artist_names = c.get
('artist_names')).drop(columns=['artist_names_x','artist_names_y', 'te
mpo_name']).set_index('track_name')
megahits
```

Out[122]:

track_name	weeks_on_charts	longest_streak_top_ten	danceability	energy	key	mode	loudn
Calling My Phone	43	9	0.907	0.393	4.0	0.0	-7.
Easy On Me	35	8	0.604	0.366	5.0	1.0	-7.
Good Days	71	7	0.436	0.655	1.0	0.0	-8.
Heat Waves	76	19	0.761	0.525	11.0	1.0	-6.
INDUSTRY BABY (feat. Jack Harlow)	51	15	0.741	0.691	10.0	0.0	-7.
...	...	...	...	...	...	...	
We Don't Talk About Bruno	28	14	0.577	0.450	0.0	0.0	-8.
deja vu	67	9	0.442	0.612	2.0	1.0	-7.
drivers license	76	9	0.561	0.431	10.0	1.0	-8.
good 4 u	61	16	0.563	0.664	9.0	1.0	-5.
pushin P (feat. Young Thug)	22	10	0.773	0.422	1.0	0.0	-4.

19 rows × 15 columns

In [123]:
```
grader.check("q7_1")
```

Out[123]: **q7_1** passed!

Question 7.2. ⭐⭐ Create a DataFrame named `megahit_comparison` that is indexed by `'audio_feature'` and contains the values given in the `audio_features` array below. Each row of `megahit_comparison` will therefore correspond to a different audio feature. `megahit_comparison` should have two columns:

- `'every_song_mean'` should contain the mean value of each feature, among all songs in `songs`.
- `'megahit_mean'` should contain the mean value of each feature, among all songs in `megahits`.

In [124]:
```
audio_features = np.array(['danceability', 'energy', 'speechiness', 'acousticness',
                          'instrumentalness', 'liveness', 'valence', 'loudness', 'tempo', 'duration_min'])
megahit_mean = np.array([])
for x in audio_features:
    q = megahits.get(x).mean()
    megahit_mean = np.append(megahit_mean, q)

every_song_mean = np.array([])
for t in audio_features:
    w = songs.get(t).mean()
    every_song_mean = np.append(every_song_mean, w)

c = ['every_song_mean','megahit_mean']
megahit_comparison = bpd.DataFrame(index=audio_features, columns=c ).assign(every_song_mean = every_song_mean).assign(megahit_mean = megahit_mean)
megahit_comparison
```

Out[124]:

	every_song_mean	megahit_mean
danceability	0.667982	0.680842
energy	0.612003	0.561789
speechiness	0.117917	0.115463
acousticness	0.267516	0.286905
instrumentalness	0.012049	0.000372
liveness	0.179257	0.230526
valence	0.520207	0.452211
loudness	-6.738406	-6.759684
tempo	121.889499	130.403211
duration_min	3.364798	3.436404

In [125]:
```
grader.check("q7_2")
```

Out[125]: **q7_2** passed!

Question 7.3. Finally, draw a horizontal bar chart showing the differences between megahits and all songs in each of the **first 7 features** in `audio_features` . These are the audio features that are measured on a 0 to 1 scale. As with the bar charts you made in Section 4, arrange the bars so that the top bar represents the audio feature which most distinguishes megahits from the rest of the songs on the top 200 charts. Make sure to give your plot an appropriate title.

Hint: Adapt the code you wrote in the `word_analysis` function.

In [126]:
```
x = megahit_comparison.get('every_song_mean') - megahit_comparison.get
('megahit_mean')
y = abs(np.array(x))
```

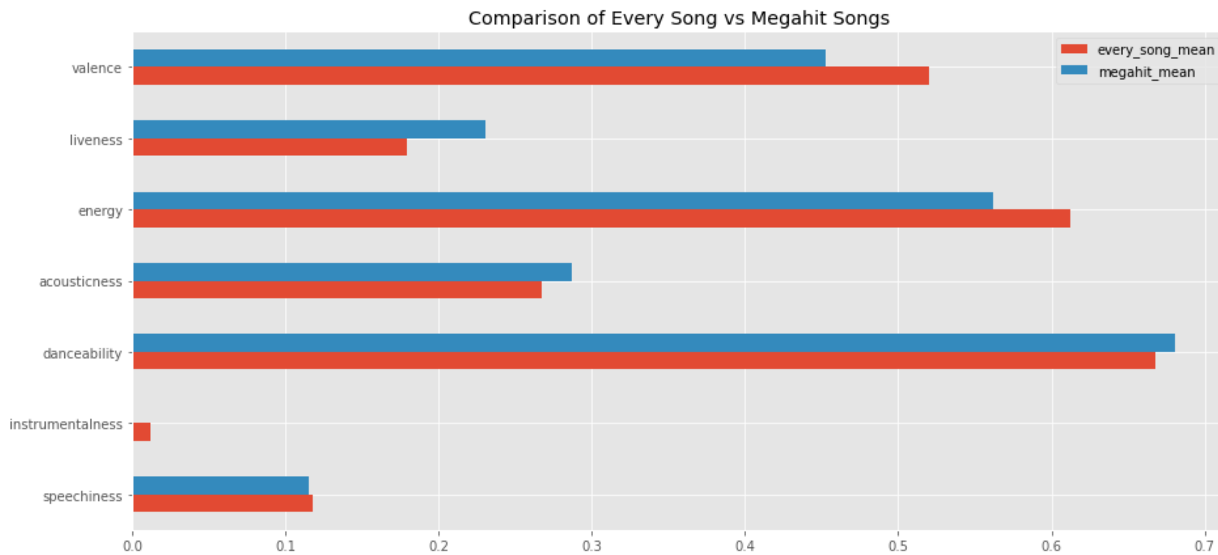
In [127]:
```
megahit_comparison2 = megahit_comparison.assign(abs_diff= y)
megahit_comparison2.sort_values(by='abs_diff', ascending = True)
```

Out[127]:

	every_song_mean	megahit_mean	abs_diff
speechiness	0.117917	0.115463	0.002454
instrumentalness	0.012049	0.000372	0.011677
danceability	0.667982	0.680842	0.012861
acousticness	0.267516	0.286905	0.019389
loudness	-6.738406	-6.759684	0.021278
energy	0.612003	0.561789	0.050214
liveness	0.179257	0.230526	0.051270
valence	0.520207	0.452211	0.067996
duration_min	3.364798	3.436404	0.071605
tempo	121.889499	130.403211	8.513711

In [128]:
```
# Make your horizontal bar chart here.
megahit_comparison2.iloc[0:7].sort_values(by='abs_diff', ascending = True).drop(columns='abs_diff').plot(kind='barh', figsize=(15,7), title = 'Comparison of Every Song vs Megahit Songs')
```

Out[128]: <AxesSubplot:title={'center':'Comparison of Every Song vs Megahit Songs'}>



Were these results what you expected to see? Of course, with music trends changing over time, the characteristics of a megahit are likely to change as well. It would be interesting to repeat this analysis with weekly top 200 charts from other time periods, or in other countries.

Keep in mind that we're also comparing megahits to other popular songs, as all of our data comes from the top 200 charts. Given a broader dataset of songs, we'd likely see a stronger characterization of a megahit, as megahits would likely have a more pronounced difference from all songs when more "unpopular" songs are included.

Parting Thoughts 💭

While you've made it to the end of the project, we've only just scratched the surface in analyzing the `charts` dataset. We encourage you to explore the dataset further by asking and answering your own questions about the Spotify charts. For instance, we didn't use the `'streams'` column in `charts` at all. Maybe you're interested in looking at the number of streams per week for your favorite song. Or maybe you're interested in recreating the interactive plot from the end of Section 5, but instead of looking at the age of each weekly #1 song, you want to look at the number of streams of each week #1 song. Explore, and let us know what you find!

Emoji Quiz 💯

Just for fun, here are some emojis that describe particular songs or artists. See how many you can identify! You may have come across some of these songs or artists while completing this project.

We'll post the answers on EdStem after the project is due.

Songs

1. 🚘🆔
2. ✋🐈‍⬛
3. 🧑‍🚀➡🌊
4. 👵👴🏽💭👧👦🏽
5. 💍💍💍💍💍💍💍
6. 📷🖼️
7. 🏃⬆️⛰️
8. 🕺🐵
9. 🧨🎇
10. 🍉🍭
11. ⚠️☠️

Artists

1. 🥛🐘🐎🦎
2. 🤿🐺
3. 🔴🕐
4. 🌶️🌶️🌶️
5. 2️⃣👄
6. 👎🐇
7. Ⓜ️➕Ⓜ️
8. 🥶▶️
9. ♂️🦆
10. 💣🐻
11. ‼️🕺🎶

Congratulations! You've completed the Midterm Project!

As usual, follow these steps to submit your assignment:

1. Select Kernel -> Restart & Run All to ensure that you have executed all cells, including the test cells.
2. Read through the notebook to make sure everything is fine and all tests passed.
3. Run the cell below to run all tests, and make sure that they all pass.
4. Download your notebook using File -> Download as -> Notebook (.ipynb), then upload your notebook to Gradescope. Don't forget to add your partner to your group on Gradescope!

If running all the tests at once causes a test to fail that didn't fail when you ran the notebook in order, check to see if you changed a variable's value later in your code. Make sure to use new variable names instead of reusing ones that are used in the tests.

Remember, the tests here and on Gradescope just check the format of your answers. We will run correctness tests after the due date has passed.

In [129]:
```
grader.check_all()
```

Out[129]:
```
q1_1 results: All test cases passed!

q1_10 results: All test cases passed!

q1_2 results: All test cases passed!

q1_4 results: All test cases passed!

q1_5 results: All test cases passed!

q1_6 results: All test cases passed!

q1_7 results: All test cases passed!

q1_8 results: All test cases passed!

q1_9 results: All test cases passed!

q2_1 results: All test cases passed!

q2_2 results: All test cases passed!

q2_3 results: All test cases passed!

q3_1 results: All test cases passed!

q3_2 results: All test cases passed!
```

```
q3_3 results: All test cases passed!

q3_4 results: All test cases passed!

q3_5 results: All test cases passed!

q3_6 results: All test cases passed!

q3_7 results: All test cases passed!

q3_8 results: All test cases passed!

q4_1 results: All test cases passed!

q4_2 results: All test cases passed!

q4_3 results: All test cases passed!

q4_4 results: All test cases passed!

q4_6 results: All test cases passed!

q4_7 results: All test cases passed!

q4_8 results: All test cases passed!

q5_1 results: All test cases passed!

q5_2 results: All test cases passed!

q5_3 results: All test cases passed!

q5_4 results: All test cases passed!

q5_5 results: All test cases passed!

q5_6 results: All test cases passed!

q6_1 results: All test cases passed!

q6_10 results: All test cases passed!

q6_2 results: All test cases passed!

q6_3 results: All test cases passed!

q6_4 results: All test cases passed!

q6_5 results: All test cases passed!
```

```
q6_6 results: All test cases passed!

q6_7 results: All test cases passed!

q6_8 results: All test cases passed!

q6_9 results: All test cases passed!

q7_1 results: All test cases passed!

q7_2 results: All test cases passed!
```