Outline of the Project

This project has seven sections. 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 1.2.** ⭐⭐). In most cases, questions worth two points are longer and more challenging than questions worth one point, though sometimes two point questions are themselves straightforward, but used to test the correctness of a previously implemented function.

Welcome 👋

Welcome to the Final 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 is the final assignment that is due this quarter, but it is due just a few days before the Final Exam. You are especially encouraged to **find a partner** to work through the project with. You can work with anyone from any section of the course, but you must follow the Project Partner Guidelines on the course website. In particular, you are both required to actively contribute to all parts of the project, and you are not allowed to split up the problems and each work on certain problems. If working in a pair, you should submit one notebook to Gradescope for the both of you.

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.

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:

- `babypandas` Notes
- Course Textbook
- DSC 10 Reference Sheet
- `babypandas` Documentation
- Other links in the Resources and Debugging tabs of the course website

Start early, good luck, and let's get started! 😎

In [1]:
```
# Don't change this cell; just run it.
import babypandas as bpd
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
plt.style.use('ggplot')

import otter
grader = otter.Notebook()
```

About the Show 📺

Friends is a TV sitcom that aired from 1994 to 2004. The show revolves around six main characters, a group of friends in their 20s and 30s who live in New York City 🗽🌇. Viewers watch these characters' lives, careers, and relationships develop over a ten year period.

The show was critically well-received and popular among viewers. It set a number of records including:

- the number one television show in 2002
- the fifth-most-watched series finale in television history
- the most-watched episode of the 2000s decade
- Outstanding Comedy Series Primetime Emmy award 🏆
- one of *TV Guide*'s 50 Greatest Shows of All Time

Friends used to be available to stream on Netflix, but with its incredible success, the show became too expensive to continue offering access. The last time *Friends* was on

Netflix in the US, it cost Netflix $100 million dollars 💸 just to offer the show on their streaming platform for one year!

If you'd like to see the show for yourself, all 10 seasons are available on MAX. 📽️

The Reboot 🆕

As an avid fan of *Friends*, you have decided to make your own reboot of the classic sitcom. A reboot of a TV show is a new show that takes place in the same setting as an older show, perhaps updated in some ways. Some reboots have a brand new cast with just a few original characters, meanwhile some reboots get the whole cast of the original show to star in the new version. For example, the reboot *How I Met Your Father* is based on the original *How I Met Your Mother*. Other examples include *That 90's Show*, a reboot of *That 70's Show*, and *Fuller House*, a reboot of *Full House*.

You've noticed that sitcom reboots are on the rise and you want to get in on the action by finding all the best and most successful aspects of *Friends* and turning them into an even better version. In order to make the reboot, you'll have to do a lot of planning and research to present your plan to the production company that will fund the show. Here's a list of all the things you'll want to have prepared by the time you meet with the producers. Each item on the list represents a section of this project.

1. Create a list of writers and directors to contact.
2. Select a top choice for the director of the reboot.
3. Determine whether to assign spoken lines to male and female characters equally.
4. Make a visualization showing the emotional range for each main character to present to the writers.
5. Decide whether the main characters should be romantically involved with one another.
6. Make a list of episode titles for the first season of the new show.
7. Decide how many episodes the reboot should have, based on viewership and rating data.

After you've done all of these things you will have completed your plan for the reboot!

About the Data 💾

Our data, originally from Kaggle, includes several different CSVs about *Friends* episodes, lines, and emotions. If you find any errors in the data, do not attempt to fix them; just analyze the data you are given.

Episodes 🎬

The `episodes` DataFrame has a row for each episode from all 10 seasons of *Friends*. For each episode, we have the

- season number (`'season'`)
- episode number (`'episode'`)
- title (`'title'`)
- director(s) (`'directed_by'`)
- writer(s) (`'written_by'`)
- premiere date (`'air_date'`)
- number of views on premiere date in millions (`'us_views_millions'`)
- rating out of ten (`'imdb_rating'`)

For example, the first episode of the first season is called `'The Pilot'`. It was directed by James Burrows and written by David Crane and Marta Kauffman. When it aired on September 22nd, 1994, there were 21.5 million viewers, and it was rated 8.3/10 on IMDb.

Run the cell below to load in the data.

In [2]:
```
episodes = bpd.read_csv('data/friends_info.csv')
episodes
```

Out[2]:

	season	episode	title	directed_by	written_by	air_date	us_views_milli
0	1	1	The Pilot	James Burrows	David Crane & Marta Kauffman	9/22/94	2
1	1	2	The One with the Sonogram at the End	James Burrows	David Crane & Marta Kauffman	9/29/94	2(
2	1	3	The One with the Thumb	James Burrows	Jeffrey Astrof & Mike Sikowitz	10/6/94	1(
3	1	4	The One with George Stephanopoulos	James Burrows	Alexa Junge	10/13/94	1(
4	1	5	The One with the East German Laundry Detergent	Pamela Fryman	Jeff Greenstein & Jeff Strauss	10/20/94	1(
...	...	...	...	...	...	...	
231	10	14	The One with Princess Consuela	Gary Halvorson	Story by: Robert CarlockTeleplay by: Tracy Reilly	2/26/04	2(
232	10	15	The One Where Estelle Dies	Gary Halvorson	Story by: Mark KunerthTeleplay by: David Crane...	4/22/04	2(
233	10	16	The One with Rachel's Going Away Party	Gary Halvorson	Andrew Reich & Ted Cohen	4/29/04	2
234	10	17	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5(
235	10	18	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5(

236 rows × 8 columns

Lines 🗣️

While the `episodes` DataFrame will give you plenty of information on the writers and directors of the show and the success of each episode, you want to know more about the characters and their interactions on the show. You have detailed line-by-line scripts for the dialogue and scene directions of 30 randomly selected *Friends* episodes, stored in a DataFrame called `lines`. For each spoken line or scene direction of these 30 episodes, the `lines` DataFrame includes the following columns:

- season number (`'season'`)
- episode number (`'episode'`)
- scene number (`'scene'`)
- line number (`'utterance'`)

- character that said the line (`'speaker'`), which is sometimes just `'Scene Directions'`
- text of the line (`'text'`)

For example, the seventh episode of season one opens with character Rachel Green saying `'Everybody? Shh, shhh. Uhhh... Central Perk is proud to present the music of Miss Phoebe Buffay.'`

Run the cell below to load in the data.

In [3]:
```
lines = bpd.read_csv('data/friends_sample.csv')
lines
```

Out[3]:

	text	speaker	season	episode	scene	utterance
0	Everybody? Shh, shhh. Uhhh... Central Perk is ...	Rachel Green	1	7	1	1
1	(applause)	Scene Directions	1	7	1	2
2	Hi. Um, I want to start with a song thats abou...	Phoebe Buffay	1	7	1	3
3	Oh, great. This is just...	Chandler Bing	1	7	2	1
4	(Chandler sees that there is a gorgeous model ...	Scene Directions	1	7	2	2
...	...	...	...	...	...	...
8452	That I can do.	Joey Tribbiani	10	13	13	9
8453	Come on! You can drink a gallon of milk in 10 ...	Phoebe Buffay	10	13	13	10
8454	All right, watch me! Okay, you time me. Ready?	Joey Tribbiani	10	13	13	11
8455	Ready... GO!	Phoebe Buffay	10	13	13	12
8456	You did it!	Phoebe Buffay	10	13	13	13

8457 rows × 6 columns

Emotions

Lastly, you have a dataset of the emotion attached to each line of the show, for most (but not all) lines in the first four seasons of the show. For each line included in our dataset, the line is associated with one of seven different emotions. These seven emotions are:

- `'Joyful'` 😄
- `'Mad'` 😡
- `'Sad'` 😭
- `'Powerful'` 💪
- `'Peaceful'` ✌️
- `'Scared'` 😱

- `'Neutral'` 😑

Classifying each line's emotion is a huge data science task in and of itself. This paper explains how convolutional neural networks were able to do this task, if you're interested in learning more.

Run the cell below to load in the data, which has the same columns as `lines` plus an extra column called `emotion`. For example, the first line of the fourth scene in Season 1, Episode 1 is categorized as `'Mad'`. That means the speaker, Ross, was probably mad when he said it. If you've ever assembled furniture, you might understand this frustration.

In [4]:
```
emotions = bpd.read_csv('data/friends_emotions.csv')
emotions
```

Out[4]:

	text	speaker	season	episode	scene	utterance	emotion
0	I'm supposed to attach a brackety thing to the...	Ross Geller	1	1	4	1	Mad
1	I'm thinking we've got a bookcase here.	Joey Tribbiani	1	1	4	3	Neutral
2	It's a beautiful thing.	Chandler Bing	1	1	4	4	Joyful
3	What's this?	Joey Tribbiani	1	1	4	5	Neutral
4	I would have to say that is an 'L'-shaped brac...	Chandler Bing	1	1	4	6	Neutral
...	...	...	...	...	...	...	...
12601	Ahh, yes, I will have a glass of the Merlot	Rachel Green	4	24	25	2	Neutral
12602	Okay.	Air Hostess	4	24	25	3	Neutral
12603	And uh, he will have a white wine spritzer.	Rachel Green	4	24	25	4	Neutral
12604	Okay, good. Thank you. I'll be back shortly, a...	Air Hostess	4	24	25	5	Joyful
12605	All right. Woo! Hey, look at that, the airport...	Rachel Green	4	24	25	6	Scared

12606 rows × 7 columns

Section 1: The Pilot 🎬

(return to the outline)

To start, we'll try to find the most experienced writers and directors in the show to know who to contact for making the reboot. The `episodes` DataFrame contains the writers and directors for each episode, but it's not in the neatest format.

In [5]:
```
episodes
```

Out[5]:

	season	episode	title	directed_by	written_by	air_date	us_views_milli
0	1	1	The Pilot	James Burrows	David Crane & Marta Kauffman	9/22/94	2
1	1	2	The One with the Sonogram at the End	James Burrows	David Crane & Marta Kauffman	9/29/94	2(
2	1	3	The One with the Thumb	James Burrows	Jeffrey Astrof & Mike Sikowitz	10/6/94	1
3	1	4	The One with George Stephanopoulos	James Burrows	Alexa Junge	10/13/94	1
4	1	5	The One with the East German Laundry Detergent	Pamela Fryman	Jeff Greenstein & Jeff Strauss	10/20/94	1
...	...	...	...	...	...	...	
231	10	14	The One with Princess Consuela	Gary Halvorson	Story by: Robert CarlockTeleplay by: Tracy Reilly	2/26/04	2
232	10	15	The One Where Estelle Dies	Gary Halvorson	Story by: Mark KunerthTeleplay by: David Crane...	4/22/04	2
233	10	16	The One with Rachel's Going Away Party	Gary Halvorson	Andrew Reich & Ted Cohen	4/29/04	2
234	10	17	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5
235	10	18	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5

236 rows × 8 columns

Some episodes were written or directed by more than one person, and many episodes have certain people who wrote the story and other people who wrote the teleplay. So identifying the most experienced writers and directors is not as straightforward as simply grouping and counting. First, we'll have to clean up the data to break up each string of names into a list of individual names.

Question 1.1. We want to first handle any episodes with two directors so that later we can count the number of episodes each director contributed to. Create a function called `parse_names` that turns a string in the `'directed_by'` column to a **list** of one or two names. Values in the `'directed_by'` column are either single names, or two names separated by an ampersand. For example,

- `parse_names('James Burrows')` should return `['James Burrows']`, and

- `parse_names('Kevin S. Bright & Gary Halvorson')` should return `['Kevin S. Bright', 'Gary Halvorson']`.

Then, apply your function to the `'directed_by'` column from the `episodes` DataFrame, and store the resulting Series in the variable `directors_by_episode`. **Do not** modify the `episodes` DataFrame.

In [6]:
```
def parse_names(names):
    '''Returns a list of individual names present in the input string.'''
    sep_names = names.split(" & ")
    return sep_names

directors_by_episode = episodes.get("directed_by").apply(parse_names)
directors_by_episode
```

Out[6]:
```
0         [James Burrows]
1         [James Burrows]
2         [James Burrows]
3         [James Burrows]
4         [Pamela Fryman]
              ...
231     [Gary Halvorson]
232     [Gary Halvorson]
233     [Gary Halvorson]
234    [Kevin S. Bright]
235    [Kevin S. Bright]
Name: directed_by, Length: 236, dtype: object
```

In [7]:
```
grader.check("q1_1")
```

Out[7]:

q1_1 passed!

For the next question, you'll need to know something interesting about how lists work in Python: when you sum two lists together, the output is one giant list that contains all the elements in both lists combined. An example is shown below.

In [8]:
```
['List', 'combining'] + ['is', 'my', 'passion']
```

Out[8]:
```
['List', 'combining', 'is', 'my', 'passion']
```

Question 1.2. ⭐⭐ Now it's time to count the number of episodes each director directed. Make a DataFrame called `directors`, indexed by `'name'`, with one column called `'num_episodes'`. There should be one row for each unique director, and `'num_episodes'` should contain the number of episodes they directed or co-directed. Sort this DataFrame so that the most experienced directors appear at the top.

Hints:

- Our solution involved creating a new, empty DataFrame with `bpd.DataFrame()`, adding columns, and using `groupby`.
- For `groupby` to give meaningful results, you must use it on a DataFrame with at least two columns.

In [9]:
```
summed = directors_by_episode.sum()
array = np.array(summed)
unique_directors =  np.unique (summed, return_counts = True)
name = unique_directors[0]
num_episodes = unique_directors[1]
directors = bpd.DataFrame().assign(name = name, num_episodes = num_episodes)
directors
```

Out[9]:

name	num_episodes
Gary Halvorson	55
Kevin S. Bright	54
Michael Lembeck	24
James Burrows	15
Gail Mancuso	14
...	...
Ellen Gittelsohn	1
David Steinberg	1
Dana DeVally Piazza	1
Arlene Sanford	1
Todd Holland	1

29 rows × 1 columns

In [10]:
```
grader.check("q1_2")
```

Out[10]:

q1_2 passed!

Now we want to repeat the process we've done so far in this section, except for writers instead of directors. This is a little trickier because for some episodes, certain writers wrote the story, meaning they developed the plot of the episode, while others wrote the teleplay, which means they wrote the dialogue and scene directions based on the story. For other episodes, the same writers wrote both the story and teleplay.

There are two ways writers are represented in the `'written_by'` column:

1. One or two names separated by `&` (similar to the `'directed_by'` column). This means these writers wrote both the story and teleplay.
2. One or two names separated by `&` after `Story by:` and one or two names separated by `&` after `Teleplay by:`

Therefore, the six possible templates of strings in the `'written_by'` column are:

1. writer
2. writer & other writer
3. Story by: writerTeleplay by: other writer
4. Story by: writerTeleplay by: other writer & another writer

5. Story by: writer & other writerTeleplay by: another writer
6. Story by: writer & other writerTeleplay by: another writer & yet another writer

Question 1.3. Implement the function `get_teleplay_writers`, which takes as input a string from the `'written_by'` column, corresponding to one episode, and returns a list of teleplay writers for that episode. Remember that if the writers' names aren't divided into story and teleplay, then all writers wrote both the story and the teleplay.

__Hints:__

- Use the `parse_names` function you wrote earlier.
- While there are six templates, you can solve this problem without using any conditional logic (`if`-statements). Your solution doesn't have to be the same as ours, but ours was just one line!

In [11]:
```
def get_teleplay_writers(names):
    '''Returns a list of names of teleplay writers present in the input stri
    if "Teleplay" in names:
        both_names = names.split("Teleplay")[1].split(" by: ")[1].split(" &
        return both_names
    else:
        parsed_names = parse_names(names)
        return parsed_names
```

In [12]:
```
grader.check("q1_3")
```

Out[12]:

q1_3 passed!

Question 1.4. Now we need a function for getting the names of the writers that wrote the story of an episode. Create a function `get_story_writers` that takes a string from the `'written_by'` column and returns a list of story writers. As before, if the writers' names aren't divided into story and teleplay, then all writers wrote both the story and the teleplay.

In [13]:
```
string = "Story by: writer & other writerTeleplay by: another writer & yet a
string.split("Teleplay")[1].split(" by: ")[1].split(" & ")
```

Out[13]:
```
['another writer', 'yet another writer']
```

In [14]:
```
def get_story_writers(names):
    '''Returns a list of names of story writers present in the input string.
    if "Story" in names:
        both_names = names.split("Teleplay")[0].split("Story by: ")[1].split
        return both_names
    else:
        parsed_names = parse_names(names)
        return parsed_names
```

In [15]:
```
grader.check("q1_4")
```

Out[15]:

q1_4 passed!

Question 1.5. Next we want to count how many times each writer wrote or co-wrote the story for an episode, as well as how many times each writer wrote or co-wrote the teleplay for an episode. This will be similar to the process we did in Question 1.2 for directors. Since we want to do something similar for story writers and teleplay writers, let's create a more general function that works for directors, story writers, and teleplay writers.

The function `count_episodes` should take as input a Series of lists, containing for each episode, either the directors, story writers, or teleplay writers for that episode. The function should return a DataFrame indexed by `'name'`, with one column called `'num_episodes'` containing a count of how many episodes each staff member worked on, sorted in descending order of the number of episodes.

For example `count_episodes(directors_by_episode)` should return a DataFrame that is identical to the `directors` DataFrame you created in Question 1.2. But we could also use the same function, with different inputs, to count the number of episodes each story writer and each teleplay writer worked on.

In [16]:
```
story_writers_series = episodes.get("written_by").apply(get_story_writers)
```

In [17]:
```
def count_episodes(staff):
    '''Returns a DataFrame showing how many episodes each staff member worke
    summed = staff.sum()
    array = np.array(summed)
    unique_output =  np.unique (summed, return_counts = True)
    name = unique_output[0]
    num_episodes = unique_output[1]
    df = bpd.DataFrame().assign(name = name, num_episodes = num_episodes).se
    return df

# An example call to your function. Feel free to change this and try out oth
count_episodes(story_writers_series)
```

Out[17]:

name	num_episodes
Ted Cohen	21
Andrew Reich	21
Scott Silveri	19
Shana Goldberg-Meehan	19
Marta Kauffman	19
...	...
Brown Mandell	1
Brian Caldirola	1
Earl Davis	1
Bill Lawrence	1
Judd Rubin	1

47 rows × 1 columns

In [18]:
```
grader.check("q1_5")
```

Out[18]:

q1_5 passed!

Question 1.6. Now, use the function you wrote in Question 1.5 to make a DataFrame called `story_writers`. This DataFrame should be indexed by `'name'`, with one column called `'num_episodes'`. There should be one row for each unique story writer, and `'num_episodes'` should contain the number of episodes for which they wrote the story, either independently or with a co-writer. Sort this DataFrame so that the most experienced story writers appear at the top.

Then do the same for teleplay writers, storing your result in `teleplay_writers`.

In [19]:
```
story_writers = count_episodes(story_writers_series)
story_writers
```

Out[19]:

name	num_episodes
Ted Cohen	21
Andrew Reich	21
Scott Silveri	19
Shana Goldberg-Meehan	19
Marta Kauffman	19
...	...
Brown Mandell	1
Brian Caldirola	1
Earl Davis	1
Bill Lawrence	1
Judd Rubin	1

47 rows × 1 columns

In [20]:
```
grader.check("q1_6_a")
```

Out[20]:

q1_6_a passed!

In [21]:
```
teleplay_writers_series = episodes.get("written_by").apply(get_teleplay_writ
teleplay_writers= count_episodes(teleplay_writers_series)
teleplay_writers
```

Out[21]:

name	num_episodes
Ted Cohen	22
Andrew Reich	22
Scott Silveri	21
Marta Kauffman	20
David Crane	20
...	...
Suzie Villandry	1
Alicia Sky Varinaitis	1
Tracy Reilly	1
Vanessa McCarthy	1
Bill Lawrence	1

46 rows × 1 columns

In [22]:
```
grader.check("q1_6_b")
```

Out[22]:

q1_6_b passed!

Question 1.7. ⭐⭐ Finally, it's time to use the information in `directors`, `story_writers`, and `teleplay_writers` to find the best candidates to help create the reboot. Create an array called `experienced_directors` containing the names of directors that directed at least 20 episodes. Then create an array called `experienced_writers` containing the names of writers that wrote at least 20 stories **and** at least 20 teleplays.

In [23]:
```
experienced_directors_df = directors[directors.get("num_episodes") >= 20]
experienced_directors_df
experienced_directors = np.array(experienced_directors_df.index)
experienced_teleplay_writers_df = teleplay_writers[teleplay_writers.get("num
experienced_story_writers_df = story_writers[story_writers.get("num_episodes
experienced_in_both_df = experienced_story_writers_df.merge(experienced_tele
experienced_writers = np.array(experienced_in_both_df.index)
print(f'The experienced directors are {experienced_directors}')
print(f'The experienced writers are {experienced_writers}')
```

```
The experienced directors are ['Gary Halvorson' 'Kevin S. Bright' 'Michael
Lembeck']
The experienced writers are ['Ted Cohen' 'Andrew Reich']
```

In [24]:
```
grader.check("q1_7")
```

Out[24]:

q1_7 passed!

These are the writers and directors you'll reach out to about your reboot, and hopefully, you can find at least one writer and one director interested in working with you on the reboot! If not, you can always relax your criteria and ask some less-experienced writers and directors.

Section 2: The One with the Best Director 🏆

(return to the outline)

You discovered in Section 1 that the two directors who directed the most episodes were Gary Halvorson and Kevin S. Bright. In this section, we want to discover which of these two directors makes better episodes, as determined by viewers. We'll use the `'imdb_rating'` column from the `episodes` DataFrame to gauge how much people enjoyed an episode.

Question 2.1. Create a DataFrame `mean_by_director` with the director's name as the index and just one column called `'mean_rating'` that contains the mean rating of all of the episodes directed by that director.

__Note:__ There is one episode that had two directors, separated by `'&'`. Exclude this episode, so that we're only looking at episodes with a single director.

In [25]:
```
only_1_director_df = episodes[~episodes.get('directed_by').str.contains("&")
mean_by_director_ratings = only_1_director_df.groupby("directed_by").mean().
mean_by_director_ratings
mean_by_director_series = mean_by_director_ratings.get("imdb_rating")
mean_by_director = mean_by_director_ratings.assign(mean_rating = mean_by_dir
mean_by_director
```

Out[25]:

directed_by	mean_rating
Joe Regalbuto	9.100000
Kevin S. Bright	8.656604
Pamela Fryman	8.650000
Andrew Tsao	8.600000
David Schwimmer	8.550000
...	...
Robby Benson	8.183333
Arlene Sanford	8.100000
Steve Zuckerman	8.100000
Thomas Schlamme	8.050000
Todd Holland	8.000000

29 rows × 1 columns

In [26]:
```
grader.check("q2_1")
```

Out[26]:

q2_1 passed!

Now let's look at how Gary Halvorson's episodes are rated, on average.

In [27]:
```
mean_by_director.get('mean_rating').loc['Gary Halvorson']
```

Out[27]:
```
8.401851851851852
```

How accurate is this number in reflecting Gary's true potential as a director? In this section, we'll consider the data we have to be a sample of each director's potential abilities. Based on this sample, we want to infer each director's ability more broadly.

For example, the average rating of episodes that Gary actually directed was 8.4, but the set of episodes he *might* have directed with additional opportunities (representing his potential as a director) could have had a higher or lower average rating. By looking at a data set of *Friends* episodes directed by Gary, we are only looking at a sample from a larger population.

We'll estimate each director's mean episode rating in the population based on their mean episode in the sample using **bootstrapping**.

Question 2.2. Below, write a function called `simulate_estimates`. It should take 3 arguments:

- `director_df`: A DataFrame with a row for each episode, which includes columns `'directed_by'` and `'imdb_rating'`.
- `director`: The name of the director whose mean episode rating we are trying to estimate.
- `repetitions`: The number of repetitions to perform (the number of resamples to create).

It should take `repetitions` resamples with replacement from the rows of `director_df` that correspond to the given `director`. For each of those resamples, it should compute the mean episode rating for that resample. Then it should return an array containing the values of those means for each resample.

In [28]:
```
def simulate_estimates(director_df, director, repetitions):
    '''Returns an array of length repetitions,
    containing bootstrapped means of the data
    in the imdb_rating column for the given director. '''
#     only_1_director_df = director_df[~director_df.get('directed_by').str.c
#     director_df_columns = only_1_director_df.get(["directed_by", "imdb_rat
    director_df_filtered_by = director_df[director_df.get("directed_by")== c
    resample_means = np.array([])
    for i in np.arange(repetitions):
        resample = director_df_filtered_by.sample(director_df_filtered_by.sh
        mean = resample.get("imdb_rating").mean()
        resample_means = np.append(resample_means, mean)
    return resample_means
```

In [29]:
```
simulate_estimates(episodes, 'Gary Halvorson', 100)
```

Out[29]:
```
array([8.3962963 , 8.3       , 8.33703704, 8.41481481, 8.49814815,
       8.34444444, 8.4462963 , 8.38148148, 8.43888889, 8.41851852,
       8.35740741, 8.40740741, 8.43888889, 8.3537037 , 8.37962963,
       8.31666667, 8.42962963, 8.45555556, 8.38703704, 8.43703704,
       8.40740741, 8.41851852, 8.34444444, 8.33703704, 8.46111111,
       8.43518519, 8.41851852, 8.33333333, 8.3962963 , 8.44259259,
       8.43148148, 8.33518519, 8.39814815, 8.40555556, 8.43333333,
       8.31851852, 8.38148148, 8.42777778, 8.38703704, 8.37222222,
       8.47592593, 8.42037037, 8.38888889, 8.41296296, 8.38333333,
       8.41111111, 8.40555556, 8.27222222, 8.4       , 8.46851852,
       8.41481481, 8.44444444, 8.35925926, 8.37222222, 8.35925926,
       8.37222222, 8.45555556, 8.37592593, 8.41111111, 8.4       ,
       8.38888889, 8.38703704, 8.38518519, 8.37407407, 8.32592593,
       8.36296296, 8.35      , 8.38703704, 8.40925926, 8.35740741,
       8.46666667, 8.47777778, 8.4037037 , 8.4462963 , 8.46851852,
       8.46666667, 8.3962963 , 8.5037037 , 8.40925926, 8.30740741,
       8.33888889, 8.46296296, 8.37777778, 8.38888889, 8.41111111,
       8.41111111, 8.43333333, 8.40740741, 8.46111111, 8.36851852,
       8.38148148, 8.40925926, 8.34259259, 8.32592593, 8.47407407,
       8.32407407, 8.39074074, 8.33333333, 8.40925926, 8.35185185])
```

In [30]:
```
grader.check("q2_2")
```

Out[30]:
q2_2 passed!

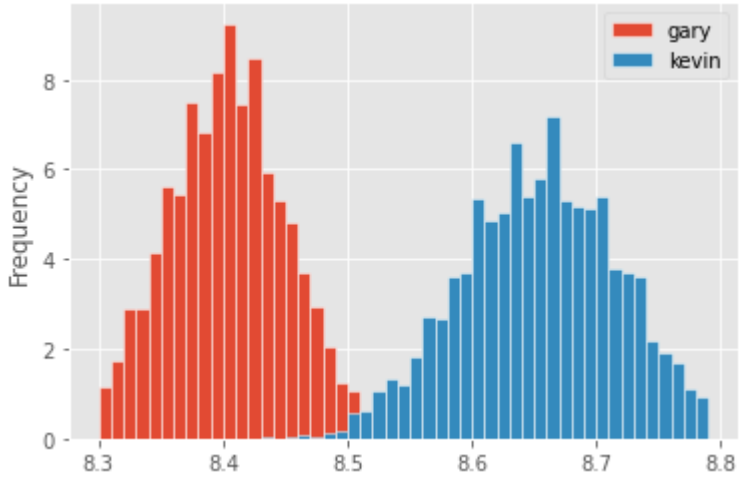
Question 2.3. Use your function `simulate_estimates` to estimate the mean rating of Gary Halvorson's episodes. Use `repetitions = 5000`, and save your array of

bootstrapped means in the variable `gary_boot_means`. Repeat the same process for Kevin S. Bright, storing your results in `kevin_boot_means`.

Then, plot the distributions of these arrays in one overlaid density histogram. Use the optional parameter `alpha=0.5` in your call to `.plot`, which changes the opacity so you can see the distributions more clearly.

In [31]:
```
gary_boot_means = simulate_estimates(episodes, "Gary Halvorson", 5000)
kevin_boot_means = simulate_estimates(episodes, "Kevin S. Bright", 5000)

# Plot your overlaid histogram here.
combined_df = bpd.DataFrame().assign(gary = gary_boot_means, kevin = kevin_b
combined_df
#histogram
combined_df.plot(kind = "hist", density=True, ec = "w", bins = np.arange(8.3
```

Out[31]:
```
<AxesSubplot:ylabel='Frequency'>
```



In [32]:
```
grader.check("q2_3")
```

Out[32]:
q2_3 passed!

Question 2.4. Now we want to calculate a 99% confidence interval for the mean episode rating of each of the two directors. To do this, create a function `confidence_interval_99`, which takes in an array of bootstrapped statistics `boot_stats` and returns a list of length two, containing the left endpoint and the right endpoint of the 99% confidence interval.

In [33]:
```
def confidence_interval_99(boot_stats):
    '''Returns a list of the endpoints of a 99% confidence interval based or
    left = np.percentile(boot_stats, 0.5)
    right = np.percentile(boot_stats, 99.5)
    return [left,right]

print('Gary 99% CI:', confidence_interval_99(gary_boot_means))
print('Kevin 99% CI:', confidence_interval_99(kevin_boot_means))
```

```
Gary 99% CI: [8.27962962962963, 8.529629629629628]
Kevin 99% CI: [8.50377358490566, 8.828301886792453]
```

In [34]:
```
grader.check("q2_4")
```

Out[34]:
q2_4 passed!

From what we've done so far, we've established that Kevin's episodes are generally rated better than Gary's episodes, however there is some overlap in their confidence intervals. This means it could be the case that Gary is just as strong a director as Kevin, but this is just not reflected in our sample.

We'll address this possibility by performing a hypothesis test with the following hypotheses:

- **Null Hypothesis:** The mean rating of Kevin's episodes in the population equals the mean rating of Gary's episodes in the population. Equivalently, the difference in the mean rating for Kevin's and Gary's episodes in the population equals 0.
- **Alternative Hypothesis:** The mean rating of Kevin's episodes in the population does not equal the mean rating of Gary's episodes in the population. Equivalently, the difference in the mean rating for Kevin's and Gary's episodes in the population does not equal 0.

Remember, the population represents all episodes they *might* have directed, or their potential as a director.

Since we were able to set up our hypothesis test as a question of whether our population parameter – the difference in mean rating for Kevin's and Gary's episodes – is equal to a certain value, we can **test our hypotheses by constructing a confidence interval for the parameter**. This is the method we used in Lecture 22 to test whether the mean human body temperature was actually 98.6 degrees Fahrenheit. For a refresher on this method, you can read more about conducting a hypothesis test with a confidence interval in CIT 13.4.

Question 2.5. ⭐⭐ Compute 1000 bootstrapped estimates for the difference in the mean rating for Kevin's episodes and Gary's episodes in the population (subtract in the order Kevin minus Gary). Store your 1000 estimates in the `difference_means` array.

You should generate your resamples of Kevin's episodes by sampling from the set of episodes directed by Kevin, and similarly for Gary, by sampling from the set of episodes directed by Gary.

In [35]:
```
#test stat = mean of kevin - mean of gary
# difference_means = np.array([])
# for i in np.arange(1000):
#     resample_mean_kevin = np.mean(simulate_estimates(episodes, "Kevin S. B
#     resample_mean_gary = np.mean(simulate_estimates(episodes, "Gary Halvor
#     resample_means_diff = abs(resample_mean_kevin - resample_mean_gary)
#     difference_means = np.append(difference_means, resample_means_diff)
# difference_means
difference_means = np.array([])
for i in np.arange(1000):
    resample_gary = episodes[episodes.get("directed_by")== "Gary Halvorson"]
    resample_gary = resample_gary.sample(resample_gary.shape[0], replace = T
    resample_gary_mean = resample_gary.get("imdb_rating").mean()
    resample_kevin = episodes[episodes.get("directed_by")=="Kevin S. Bright'
    resample_kevin = resample_kevin.sample(resample_kevin.shape[0], replace
```

```
    resample_kevin_mean = resample_kevin.get("imdb_rating").mean()
    difference = abs(resample_gary_mean - resample_kevin_mean)
    difference_means = np.append(difference_means, difference)
difference_means
# Just display the first ten differences.
difference_means[:10]
```

Out[35]:
```
array([0.37389937, 0.14192872, 0.24392034, 0.35027952, 0.27990915,
       0.24378057, 0.19468903, 0.25677848, 0.26719078, 0.13955276])
```

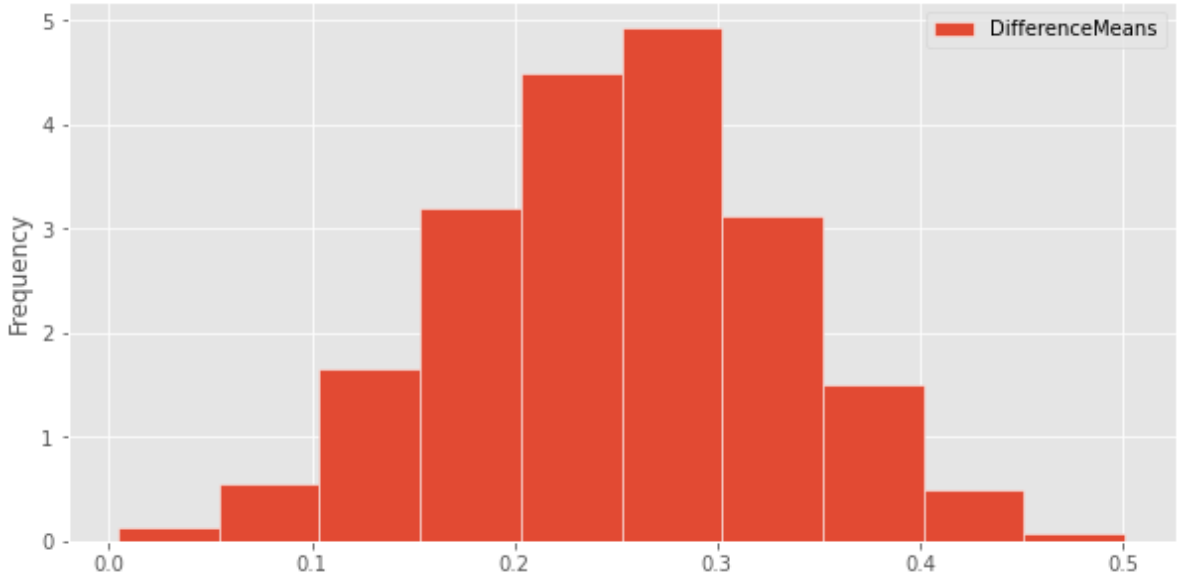
In [36]:
```
grader.check("q2_5")
```

Out[36]:

q2_5 passed!

Let's visualize your estimates:

In [37]:
```
(bpd.DataFrame().assign(DifferenceMeans = difference_means)
 .plot(kind='hist', density=True, ec='w', figsize=(10, 5)));
```



Question 2.6. Use the function `confidence_interval_99` you created before to compute a 99% confidence interval for the difference in the mean rating of Kevin's and Gary's episodes (as before, Kevin minus Gary). Assign to `kevin_gary_difference_CI` a list containing the endpoints of this confidence interval.

In [38]:
```
kevin_gary_difference_CI = confidence_interval_99(difference_means)
kevin_gary_difference_CI
```

Out[38]:
```
[0.04662753319356928, 0.4374594689028666]
```

In [39]:
```
grader.check("q2_6")
```

Out[39]:

q2_6 passed!

Recall the hypotheses we were testing:

- **Null Hypothesis:** The mean rating of Kevin's episodes in the population equals the mean rating of Gary's episodes in the population. Equivalently, the difference in the

mean rating for Kevin's and Gary's episodes in the population equals 0.

- **Alternative Hypothesis:** The mean rating of Kevin's episodes in the population does not equal the mean rating of Gary's episodes in the population. Equivalently, the difference in the mean rating for Kevin's and Gary's episodes in the population does not equal 0.

Question 2.7. Based on the confidence interval you've created, would you reject the null hypothesis at the 0.01 significance level? Set `reject_kevin_gary` to True if you would reject the null hypothesis, and False if you would not.

In [40]:
```
reject_kevin_gary = True
reject_kevin_gary
```

Out[40]: True

In [41]:
```
grader.check("q2_7")
```

Out[41]: **q2_7** passed!

We have now discovered which of these two directors would likely make better episodes for your reboot. However, we also want to know whether Gary and Kevin's episodes have other differences besides the ratings they generate. For example, does one of them tend to direct episodes with more views?

Let's write a single function that works for both rating and viewership. To do this, we want to generalize our simulation code from Question 2.2 so that we can create a confidence interval for either variable.

Question 2.8. Create a function called `compare_gary_kevin`, which takes in 3 inputs:

- `sample_df`, a DataFrame with a row for each episode in our sample, which includes a column called `'directed_by'`.
- `variable`, the column name of the relevant variable whose mean we want to estimate.
- `repetitions`, the number of repetitions to perform (the number of resamples to create).

The function should adhere to these specifications:

1. The function should generate an overlaid density histogram, showing the distributions of bootstrapped means of the given variable, for both Kevin and Gary. Make sure to give your histogram a descriptive title and to use appropriate labels. Use `bins=20`, `alpha=0.5`, and `figsize=(10,5)`.
2. The function should print a statement with the 99% confidence interval for the mean value of the given variable in the population, for both Kevin and Gary. See the example below for the type of statement to print, but the exact formatting is up to you.
3. The function should return nothing.

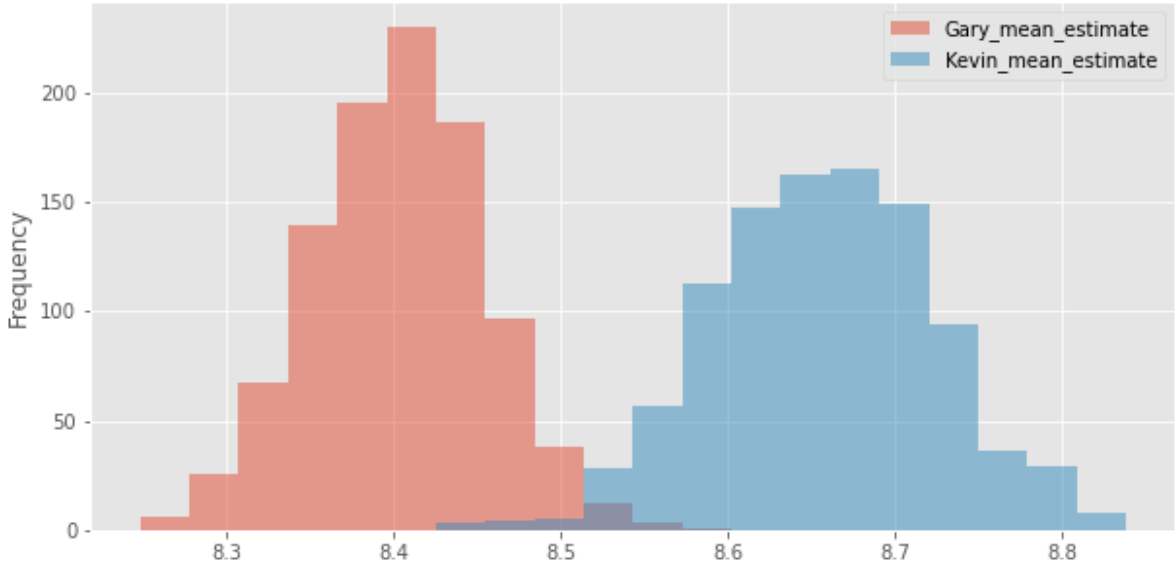
Hints:

- This is designed to be a challenging question, but remember that you can use any of the functions you've already created.
- Our solution does the necessary operations once for Gary and once for Kevin and assigns columns named `'Gary_mean_estimate'` and `'Kevin_mean_estimate'`.

Here is an example output that shows a comparison of estimates for Gary and Kevin's mean `'imdb_rating'`.

In [42]:
```
def compare_gary_kevin(sample_df, variable, repetitions):
    '''For each of Gary and Kevin, display a distribution of bootstrapped me
    a confidence interval for the mean value of the variable from sample_df.
    resample_kevin_means = np.array([])
    resample_gary_means = np.array([])
    for i in np.arange(repetitions):
        resample_gary = sample_df[sample_df.get("directed_by")== "Gary Halvo
        resample_gary = resample_gary.sample(resample_gary.shape[0], replace
        resample_gary_mean = resample_gary.get(variable).mean()
        resample_kevin = sample_df[sample_df.get("directed_by")== "Kevin S.
        resample_kevin = resample_kevin.sample(resample_kevin.shape[0], repl
        resample_kevin_mean = resample_kevin.get(variable).mean()
        resample_kevin_means = np.append(resample_kevin_means, resample_kevi
        resample_gary_means = np.append(resample_gary_means, resample_gary_m
    comparing_df = bpd.DataFrame().assign(Gary_mean_estimate = resample_gary
    comparing_df.plot(kind = "hist", bins = 20, alpha = 0.5, figsize =(10,5)
    kevin_CI = confidence_interval_99(resample_kevin_means)
    gary_CI = confidence_interval_99(resample_gary_means)
    print ("Gary's 99% CI for", variable, ":", gary_CI)
    print ("Kevin's 99% CI for", variable, ":", kevin_CI)
# Try to replicate the graph shown in the example.
compare_gary_kevin(episodes, 'imdb_rating', 1000)
```

```
Gary's 99% CI for imdb_rating : [8.274074074074074, 8.53889814814815]
Kevin's 99% CI for imdb_rating : [8.479160377358491, 8.822660377358488]
```



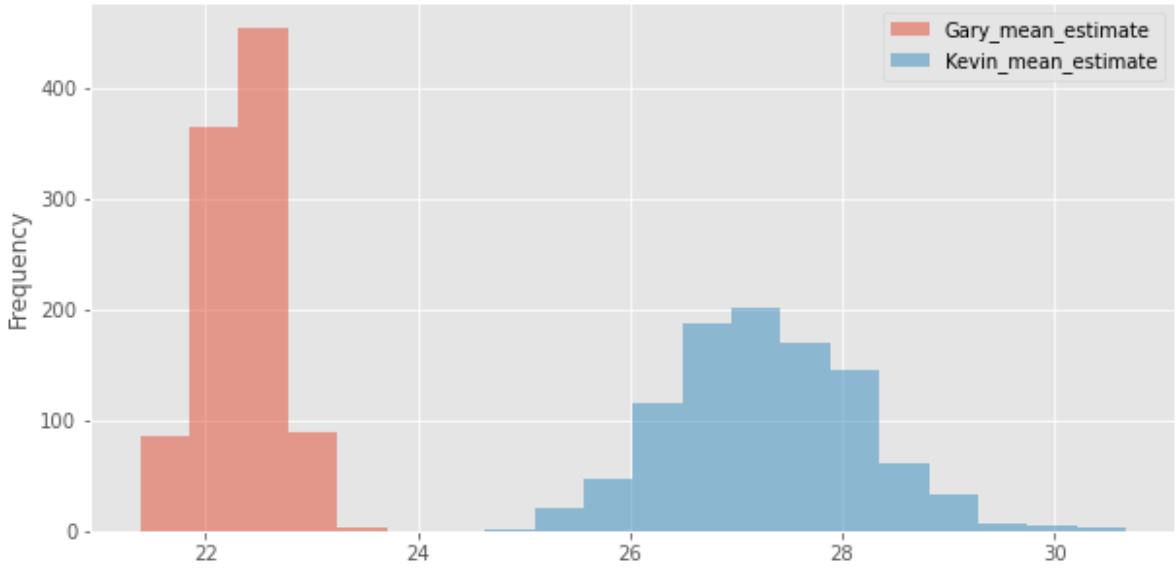
Question 2.9. ⭐⭐ Using the `compare_gary_kevin` function you just wrote, create a plot and confidence intervals that would help you answer the following question.

> Whose episodes have more views on average, Kevin's or Gary's?

This question is not difficult; it is worth two points because it tests your implementation of the function `compare_gary_kevin` which you wrote in the previous question. All you need to do here is make one call to your function with the appropriate inputs. Use `repetitions=1000`.

In [43]:
```
# Make your function call here.Be sure to run this cell before submitting.
compare_gary_kevin(episodes, "us_views_millions", 1000)
```

```
Gary's 99% CI for us_views_millions : [21.496775925925924, 23.230387037037033]
Kevin's 99% CI for us_views_millions : [25.23285660377358, 29.925508490566035]
```



At this point it should be clear which director you'll be reaching out to for your reboot!

Section 3: The One About Gender Balance 👩⚖️👨

(return to the outline)

After watching a couple of episodes of *Friends*, you start to wonder if the three male main characters (`'Ross Geller'`, `'Chandler Bing'`, `'Joey Tribbiani'`) speak more lines than the the three female main characters (`'Rachel Green'`, `'Monica Geller'`, `'Phoebe Buffay'`). You want your reboot to be true to the spirit of the original show, but you don't feel great about creating a television show that has a significant gender imbalance.

Below is the `lines` DataFrame, which contains information on all the lines from a random sample of 30 *Friends* episodes. Notice that `lines` contains **every** line from each one of these 30 episodes, including lines from characters other than the main six, and even the scene directions.

In [44]:
```
lines
```

Out[44]:

	text	speaker	season	episode	scene	utterance
0	Everybody? Shh, shhh. Uhhh... Central Perk is ...	Rachel Green	1	7	1	1
1	(applause)	Scene Directions	1	7	1	2
2	Hi. Um, I want to start with a song thats abou...	Phoebe Buffay	1	7	1	3
3	Oh, great. This is just...	Chandler Bing	1	7	2	1
4	(Chandler sees that there is a gorgeous model ...	Scene Directions	1	7	2	2
...	...	...	...	...	...	...
8452	That I can do.	Joey Tribbiani	10	13	13	9
8453	Come on! You can drink a gallon of milk in 10 ...	Phoebe Buffay	10	13	13	10
8454	All right, watch me! Okay, you time me. Ready?	Joey Tribbiani	10	13	13	11
8455	Ready... GO!	Phoebe Buffay	10	13	13	12
8456	You did it!	Phoebe Buffay	10	13	13	13

8457 rows × 6 columns

Question 3.1. Write a function named `main_char` that takes in the name of a single speaker and determines whether that speaker is one of the six main characters:

- `'Ross Geller'`
- `'Chandler Bing'`
- `'Joey Tribbiani'`
- `'Rachel Green'`
- `'Monica Geller'`
- `'Phoebe Buffay'`

For example, `main_char('Chandler Bing')` should return `True`, but `main_char('Janice Hosenstein')` should return `False`.

Then, use your function to filter the `lines` DataFrame to contain only lines spoken by one of the six main characters, saving the result as `main_lines`.

In [45]:
```
def main_char(speaker):
    '''Returns True if speaker is a main character, False otherwise.'''
    if speaker == "Ross Geller":
        return True
    elif speaker == "Chandler Bing":
        return True
    elif speaker == "Joey Tribbiani":
        return True
    elif speaker == "Rachel Green":
        return True
    elif speaker == "Monica Green":
        return True
```

```
    elif speaker == "Phoebe Buffay":
        return True
    else:
        return False

main_lines = lines[lines.get("speaker").apply(main_char)]
main_lines
```

Out[45]:

	text	speaker	season	episode	scene	utterance
0	Everybody? Shh, shhh. Uhhh... Central Perk is ...	Rachel Green	1	7	1	1
2	Hi. Um, I want to start with a song thats abou...	Phoebe Buffay	1	7	1	3
3	Oh, great. This is just...	Chandler Bing	1	7	2	1
5	Wow, this is so cool, you guys. The entire cit...	Rachel Green	1	7	3	1
7	Wow, you guys, this is big.	Rachel Green	1	7	3	3
...	...	...	...	...	...	...
8452	That I can do.	Joey Tribbiani	10	13	13	9
8453	Come on! You can drink a gallon of milk in 10 ...	Phoebe Buffay	10	13	13	10
8454	All right, watch me! Okay, you time me. Ready?	Joey Tribbiani	10	13	13	11
8455	Ready... GO!	Phoebe Buffay	10	13	13	12
8456	You did it!	Phoebe Buffay	10	13	13	13

5505 rows × 6 columns

In [46]:
```
grader.check("q3_1")
```

Out[46]:

q3_1 passed!

Question 3.2. Using the `main_lines` DataFrame, count the total number of lines spoken by a main character across all episodes in the sample and assign your answer to the variable `line_count`.

Then compute the proportion of these lines that were spoken by male characters (`'Ross Geller'`, `'Chandler Bing'`, `'Joey Tribbiani'`) and the proportion of lines that were spoken by female characters (`'Rachel Green'`, `'Monica Geller'`, `'Phoebe Buffay'`). Assign your answers to the variables `observed_male_prop` and `observed_female_prop`.

In [47]:
```
line_count = main_lines.shape[0]
observed_female_count = main_lines[(main_lines.get("speaker") == "Rachel Gre
observed_female_prop = observed_female_count/line_count
observed_male_count = main_lines[(main_lines.get("speaker")== "Joey Tribbiar
```

```
observed_male_prop = observed_male_count/line_count

print('Number of Lines: ' + str(line_count))
print('Male Proportion: ' + str(observed_male_prop))
print('Female Proportion: ' + str(observed_female_prop))
```

```
Number of Lines: 5505
Male Proportion: 0.6096276112624887
Female Proportion: 0.39037238873751134
```

In [48]:
```
grader.check("q3_2")
```

Out[48]:

q3_2 passed!

You recognize that `observed_female_prop` and `observed_male_prop` are similar but they're not exactly the same. Is this just random chance that your sample happened to include more lines by male main characters? Or is the case that throughout the show *Friends*, lines spoken by one of the six main characters are actually more likely to be spoken by male characters? Let's do a hypothesis test with the following hypotheses:

- **Null Hypothesis**: Throughout the ten seasons of *Friends*, male main characters speak the same number of lines as female main characters.
- **Alternative Hypothesis**: Throughout the ten seasons of *Friends*, male main characters speak more lines than female main characters.

Run the cell below to define a variable `null_distribution` that shows the proportion of each gender according to our model.

In [49]:
```
null_distribution = np.array([0.5, 0.5])
null_distribution
```

Out[49]:
```
array([0.5, 0.5])
```

Question 3.3. To perform our hypothesis test, we will simulate drawing a random sample of size `line_count` from the null distribution, and then compute a test statistic on each simulated sample. We must first choose a reasonable test statistic that will help us determine whether or not to reject the null hypothesis.

From the options below, find **all** valid test statistics that we could use for this hypothesis test. Save the numbers of your choices in a **list** called `gender_test_statistics`. Valid test statistics are ones that would allow us to distinguish between the null and alternative hypotheses.

__Hint:__ To determine whether a test statistic is valid, think about which values of the statistic (high, low, moderate) would make you lean towards the null and which would make you lean towards the alternative.

1. The difference between the proportion of female-spoken lines and 0.5.
2. The difference between the number of male-spoken lines and the number of female-spoken lines.
3. The difference between the number of female-spoken lines and one half of `line_count`.
4. Three times the difference between the proportion of male-spoken lines and 0.5.

5. The total variation distance between the gender distribution of lines spoken and the null distribution.

In [50]:
```
gender_test_statistics = [1,2,3,4]
gender_test_statistics
```

Out[50]:
```
[1, 2, 3, 4]
```

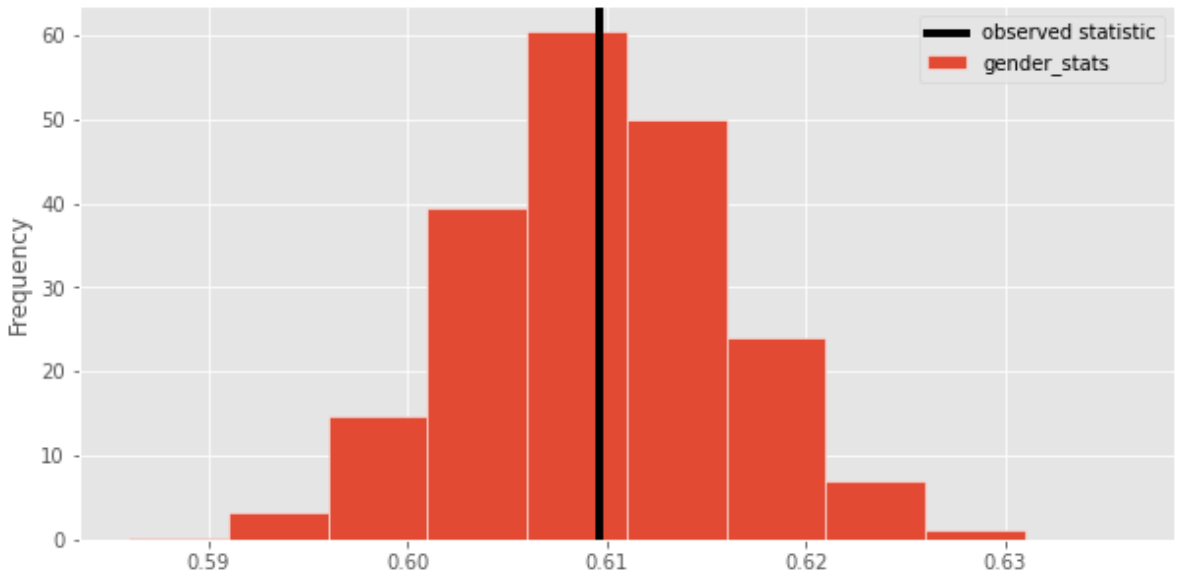
In [51]:
```
grader.check("q3_3")
```

Out[51]:

q3_3 passed!

Question 3.4. For this hypothesis test, we'll use the proportion of male-spoken lines as our test statistic. Write a simulation that runs 10,000 times, each time drawing a random sample of size `line_count` under the assumption of the null hypothesis. Keep track of the simulated test statistics in the `gender_stats` array.

In [52]:
```
gender_stats = np.array([])
for i in np.arange(10000):
    sample = main_lines.sample(main_lines.shape[0], replace = True)
    observed_male_count_sample = (
        sample[(sample.get("speaker")== "Joey Tribbiani") | (sample.get("spe
            sample.get("speaker")== "Ross Geller")].shape[0])
    observed_male_prop_sample = observed_male_count_sample/main_lines.shape[
    gender_stats = np.append(gender_stats, observed_male_prop_sample)


# Visualize with a histogram
bpd.DataFrame().assign(gender_stats=gender_stats).plot(kind='hist', density=
plt.axvline(x=observed_male_prop, color='black', linewidth=4, label='observe
plt.legend();
```



In [53]:
```
grader.check("q3_4")
```

Out[53]:

q3_4 passed!

Question 3.5. Compute the p-value for this hypothesis test, and save the result to `gender_p_value`.

In [54]:
```
gender_p_value = np.count_nonzero(gender_stats >= observed_male_prop)/len(ge
gender_p_value
```

Out[54]: 0.499

In [55]:
```
grader.check("q3_5")
```

Out[55]: **q3_5** passed!

You should find that the p-value is above the standard cutoff of 0.05 for statistical significance. So in this case, we fail to reject the null. You'll use this information when creating your reboot - you'll assign an equal number of lines to the male main characters and the female main characters.

Question 3.6. Conceptually, how would you expect the statistics in `gender_stats` to change if `line_count` were a much larger value and `observed_male_prop` were the same? What effect would that have on the result of the hypothesis test?

From the options below, save the number of your choice in the variable `gender_stats_change`.

1. The values in `gender_stats` would be **less spread out**. We'd be **less** likely to reject the null hypothesis if `observed_male_prop` remained the same.
2. The values in `gender_stats` would be **less spread out**. We'd be **more** likely to reject the null hypothesis if `observed_male_prop` remained the same.
3. The values in `gender_stats` would be **about the same**. We'd be **equally** likely to reject the null hypothesis if `observed_male_prop` remained the same.
4. The values in `gender_stats` would be **more spread out**. We'd be **less** likely to reject the null hypothesis if `observed_male_prop` remained the same.
5. The values in `gender_stats` would be **more spread out**. We'd be **more** likely to reject the null hypothesis if `observed_male_prop` remained the same.

In [56]:
```
gender_stats_change = 1
gender_stats_change
```

Out[56]: 1

In [57]:
```
grader.check("q3_6")
```

Out[57]: **q3_6** passed!

Section 4: The Emotional One 😁😢😨

(return to the outline)

Now it's time to investigate the character dynamics of the show! In order to do this, we'll look at the `emotions` dataset. In data science, we often refer to emotion as "sentiment." We only have sentiment data for certain lines from the first four seasons of

the show, so we'll base our investigation on those lines only. Since we also have knowledge of probability, we'll utilize that as well.

Let's take a look at the data in `emotions`.

In [58]:
```
emotions
```

Out[58]:

	text	speaker	season	episode	scene	utterance	emotion
0	I'm supposed to attach a brackety thing to the...	Ross Geller	1	1	4	1	Mad
1	I'm thinking we've got a bookcase here.	Joey Tribbiani	1	1	4	3	Neutral
2	It's a beautiful thing.	Chandler Bing	1	1	4	4	Joyful
3	What's this?	Joey Tribbiani	1	1	4	5	Neutral
4	I would have to say that is an 'L'-shaped brac...	Chandler Bing	1	1	4	6	Neutral
...	...	...	...	...	...	...	...
12601	Ahh, yes, I will have a glass of the Merlot	Rachel Green	4	24	25	2	Neutral
12602	Okay.	Air Hostess	4	24	25	3	Neutral
12603	And uh, he will have a white wine spritzer.	Rachel Green	4	24	25	4	Neutral
12604	Okay, good. Thank you. I'll be back shortly, a...	Air Hostess	4	24	25	5	Joyful
12605	All right. Woo! Hey, look at that, the airport...	Rachel Green	4	24	25	6	Scared

12606 rows × 7 columns

The `emotions` DataFrame includes the emotion for lines spoken by minor characters like `'Air Hostess'`, but we want to focus on the emotions of the six main characters, whose names are included in the `main` DataFrame defined below.

In [59]:
```
main = bpd.DataFrame().assign(speaker =  ['Monica Geller', 'Ross Geller', 'R
main
```

Out[59]:

	speaker
0	Monica Geller
1	Ross Geller
2	Rachel Green
3	Chandler Bing
4	Phoebe Buffay
5	Joey Tribbiani

Question 4.1. Create a DataFrame called `main_emotions` by merging the `main` and `emotions` DataFrame so that the resulting DataFrame contains the same columns as `emotions`, but only has rows corresponding to lines spoken by one of the six main characters. The rows and columns of `main_emotions` can be in any order.

In [60]:
```
main_emotions = main.merge(emotions,left_on='speaker',right_on='speaker')
main_emotions
```

Out[60]:

	speaker	text	season	episode	scene	utterance	emotion
0	Monica Geller	Oh my God!	1	1	5	1	Mad
1	Monica Geller	My brother's going through that right now, he'...	1	1	5	3	Sad
2	Monica Geller	-leg?	1	1	5	5	Joyful
3	Monica Geller	You actually broke her watch? Wow! The worst t...	1	1	5	7	Powerful
4	Monica Geller	That's right.	1	1	5	9	Powerful
...	...	...	...	...	...	...	...
9771	Joey Tribbiani	Thanks man.	4	24	23	13	Peaceful
9772	Joey Tribbiani	But what about how much taller he is than me?	4	24	23	15	Scared
9773	Joey Tribbiani	I mean, there's no way I can make myself talle...	4	24	23	17	Scared
9774	Joey Tribbiani	Hey, Monica, wow you've been in the bathroom f...	4	24	23	19	Scared
9775	Joey Tribbiani	Had the beef-tips, huh?	4	24	23	21	Powerful

9776 rows × 7 columns

In [61]:
```
grader.check("q4_1")
```

Out[61]:

q4_1 passed!

We'll use `main_emotions` instead of `emotions` for the rest of this section. Let's look into how sentiment varies across the six main characters by investigating some probabilities.

Question 4.2. If we randomly select a line spoken `'Phoebe Buffay'` from `main_emotions`, what is the probability that the line is characterized as `'Joyful'`? Assign your answer to the variable `p_joyful_given_phoebe`.

In [62]:
```
condition1 = main_emotions.get('speaker') == 'Phoebe Buffay'
condition2 = main_emotions.get('emotion') == 'Joyful'

combined_condition = condition1 & condition2
```

```
main_emotions_spkr = main_emotions.loc[combined_condition]

p_joyful_given_phoebe = main_emotions_spkr.shape[0]/main_emotions.shape[0]
p_joyful_given_phoebe
```

Out[62]: 0.039484451718494275

In [63]:
```
grader.check("q4_2")
```

Out[63]: **q4_2** passed!

Question 4.3. It's hard to know from this probability alone whether Phoebe is a particularly joyful character compared to other characters. Let's instead answer a related question: if we randomly select a `'Joyful'` line from `main_emotions`, what is the probability that it was spoken by `'Phoebe Buffay'`? Assign your answer to the variable `p_phoebe_given_joyful`.

In [64]:
```
condition1 = main_emotions.get('emotion')  == 'Joyful'
main_emotions_joy = main_emotions.loc[condition1]
p_phoebe_given_joyful = main_emotions_spkr.shape[0]/main_emotions_joy.shape[
p_phoebe_given_joyful
```

Out[64]: 0.18293838862559242

In [65]:
```
grader.check("q4_3")
```

Out[65]: **q4_3** passed!

Notice that in both of the previous questions, you calculated a conditional probability. First you calculated the probability of an emotion given a speaker, then you calculated the probability of a speaker given an emotion. Next, we want to do these same kinds of calculations for *all* of the main characters and *all* of the emotions. Let's generalize the code for these calculations so that we can more easily compute conditional probabilities with other conditions.

Question 4.4. ⭐⭐ Your job is to implement the function `conditional_probability`. It has two arguments, `find` and `given`, both of which are lists. Let's walk through how this works using an example. Suppose we want to compute the probability that a `'Joyful'` line is spoken by `'Phoebe Buffay'` as we did in the previous question.

- `find` is a list of two elements:
  - The first element in `find` is the column in `main_emotions` that contains the event that we are trying to find the probability of. In our example, this is `'speaker'`.
  - The second element in `find` is the value in the aforementioned column that we're trying to find. In our example, this is `'Phoebe Buffay'`.
- `given` is a list of two elements:
  - The first element in `given` is the column in `main_emotions` that contains the event that is known to be true. In our example, this is `'emotion'`.

- The second element in `given` is the value in the aforementioned column. In our example, this is `'Joyful'`.

Putting this all together, this means that `conditional_probability(['speaker', 'Phoebe Buffay'], ['emotion', 'Joyful'])` should evaluate to your answer from the previous part (but the `conditional_probability` function should work for any example, not just this one).

In [66]:
```
def conditional_probability(find, given):
    '''Returns the conditional probability of an event given a known conditi
    condition1 = main_emotions.get(find[0]) == find[1]
    condition2 = condition1.get(given[0]) == given[1]
    main_emotions_spkr = main_emotions.loc[condition1 & condition2]
    main_emotions_emot = main_emotions.loc[condition1]
    return main_emotions_spkr.shape[0]/main_emotions_emot.shape[0]
# This should evalaute to your answer to Question 4.3. Feel free to try calc
conditional_probability(['speaker', 'Phoebe Buffay'], ['emotion', 'Joyful'])


def conditional_probability(find, given):
    '''Returns the conditional probability of an event given a known conditi
    cond1 = main_emotions[main_emotions.get(given[0]) == given[1]]

    cond2 = cond1[cond1.get(find[0]) == find[1]]

    return cond2.shape[0] / cond1.shape[0]

# This should evalaute to your answer to Question 4.3. Feel free to try calc
conditional_probability(['speaker', 'Phoebe Buffay'], ['emotion', 'Joyful'])
```

Out[66]:
```
0.18293838862559242
```

In [67]:
```
grader.check("q4_4")
```

Out[67]:
q4_4 passed!

Question 4.5. Now, use the `conditional_probability` function you just wrote to find some probabilities:

- `p_mad_given_ross`: The probability that a line spoken by `'Ross Geller'` is `'Mad'`.
- `p_monica_given_neutral`: The probability that a `'Neutral'` line is said by `'Monica Geller'`.

In [68]:
```
p_mad_given_ross = conditional_probability(['speaker', 'Ross Geller'], ['emo
p_monica_given_neutral = conditional_probability(['speaker', 'Monica Geller'

print('P(Mad given Ross) = ' + str(p_mad_given_ross))
print('P(Monica given Neutral) = ' + str(p_monica_given_neutral))
```

```
P(Mad given Ross) = 0.17
P(Monica given Neutral) = 0.16713483146067415
```

In [69]:
```
grader.check("q4_5")
```

Out[69]:

q4_5 passed!

Question 4.6. ⭐⭐ Now you want to focus on a particular emotion and order the characters from most likely to say a line of that emotion to least likely. Create a function called `rank_speakers_by_emotion` that takes as input a string `emotion` representing one of the seven emotions, and a boolean `draw_plot`.

If `draw_plot` is `True`, the function should plot a horizontal bar chart where the bar lengths represent the probability that a line of the given emotion is spoken by each of the the six main characters. The total length of the six bars should sum to 1 and the bars should be ordered from longest to shortest. Give your plot an appropriate title.

When `draw_plot` is `False`, the function should not draw a plot.

Regardless of the value of `draw_plot`, the function should return an array of the six main characters, ranked in descending order of their probabilities of saying a line of the given emotion.

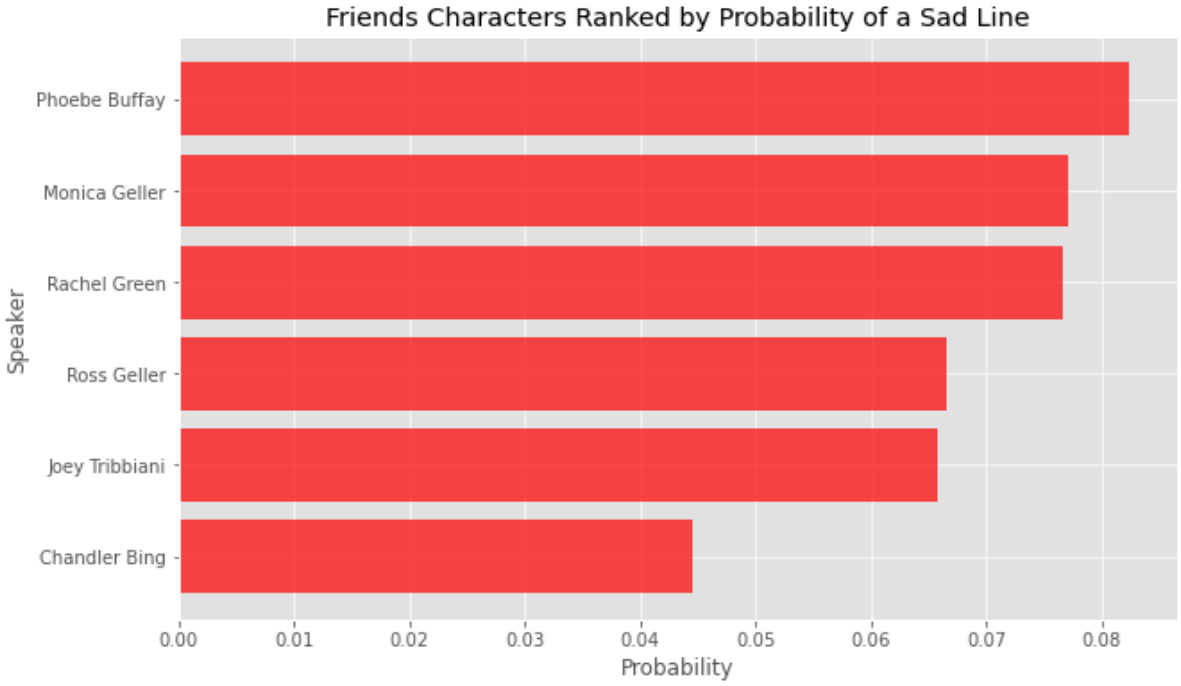
For example, `rank_speakers_by_emotion('Sad', True)` should behave as follows.

__Hint:__ You can solve this problem in at least two different ways. One way involves calling your `conditional_probability` function repeatedly with different inputs. Another way doesn't use the `conditional_probability` function at all, but instead uses `groupby`. You can use either approach, or try both for extra practice.

In [70]:

```
#def rank_speakers_by_emotion(emotion, draw_plot):
# '''Returns an ordered array of main characters, ranked by their conditiona
# speaking a line of the given emotion. Draws a bar graph when draw_plot is
# speaker_array = np.array([])
# prob_array = np.array([])
# for i in np.array(main.get("speaker")):
#     prob = conditional_probability(['emotion', emotion], ["speaker", i])
#     prob_array = np.append(prob_array, prob)
#
#     df = bpd.DataFrame().assign(speaker=main.get("speaker")).assign(probab
# if draw_plot:
#     df.plot(kind="barh")
# return np.array(df.sort_values("probability", ascending=False).index)
# Try to replicate the graph shown in the example.
#rank_speakers_by_emotion('Sad', True)

def rank_speakers_by_emotion(emotion, draw_plot):
    '''Returns an ordered array of main characters, ranked by their conditio
    speaking a line of the given emotion. Draws a bar graph when draw_plot i

    filtered = main_emotions[main_emotions.get('emotion') == emotion]

    speakers = main_emotions.get('speaker').unique()
    total_lines = []
    for speaker in speakers:
        count = (main_emotions.get('speaker') == speaker).sum()
        total_lines.append(count)
```

```
    emotion_lines = []
    for speaker in speakers:
        count = (filtered.get('speaker') == speaker).sum()
        emotion_lines.append(count)

    probabilities = np.array(emotion_lines) / np.array(total_lines)

    sorted_indices = np.argsort(probabilities)[::-1]
    ranked_speakers = np.array(speakers)[sorted_indices]
    sorted_probabilities = probabilities[sorted_indices]

    plt.figure(figsize=(10, 6))
    plt.barh(ranked_speakers, sorted_probabilities, color='red', alpha=0.7)
    plt.xlabel('Probability')
    plt.ylabel('Speaker')
    plt.title(f'Friends Characters Ranked by Probability of a {emotion} Line
    plt.gca().invert_yaxis()
    plt.show()

    return ranked_speakers

rank_speakers_by_emotion('Sad', True)
```



Out[70]:
```
array(['Phoebe Buffay', 'Monica Geller', 'Rachel Green', 'Ross Geller',
       'Joey Tribbiani', 'Chandler Bing'], dtype=object)
```

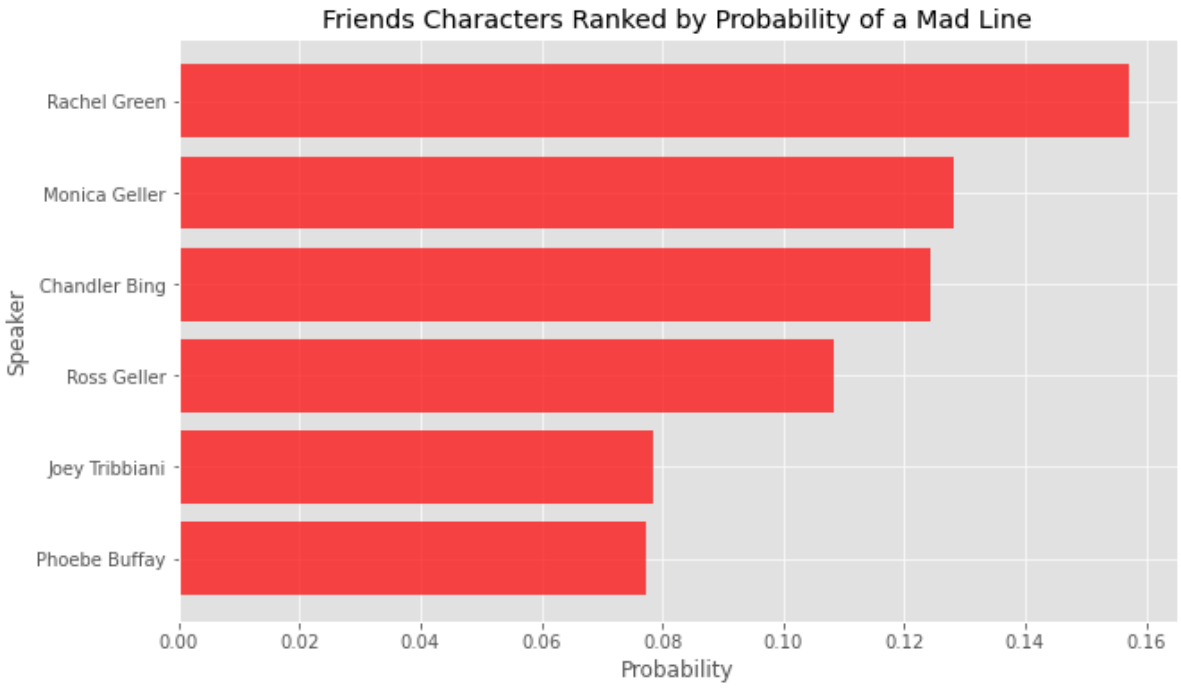
In [71]:
```
grader.check("q4_6")
```

Out[71]:

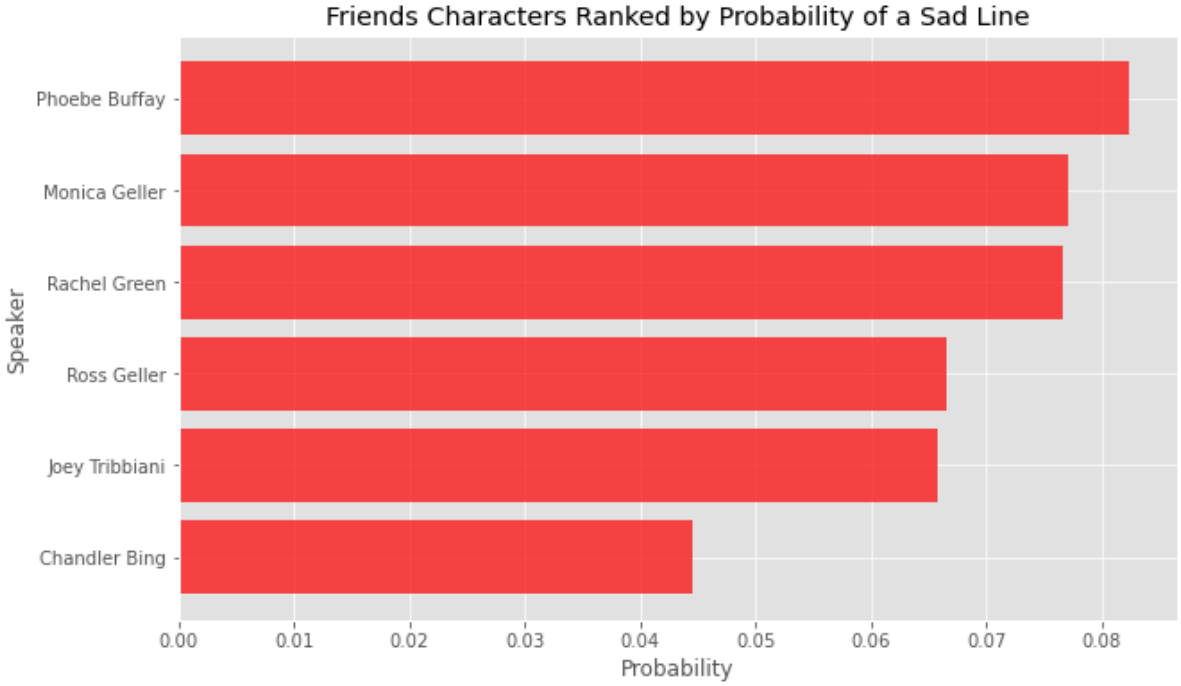
q4_6 passed!

Now, let's try using our function on all seven emotions.

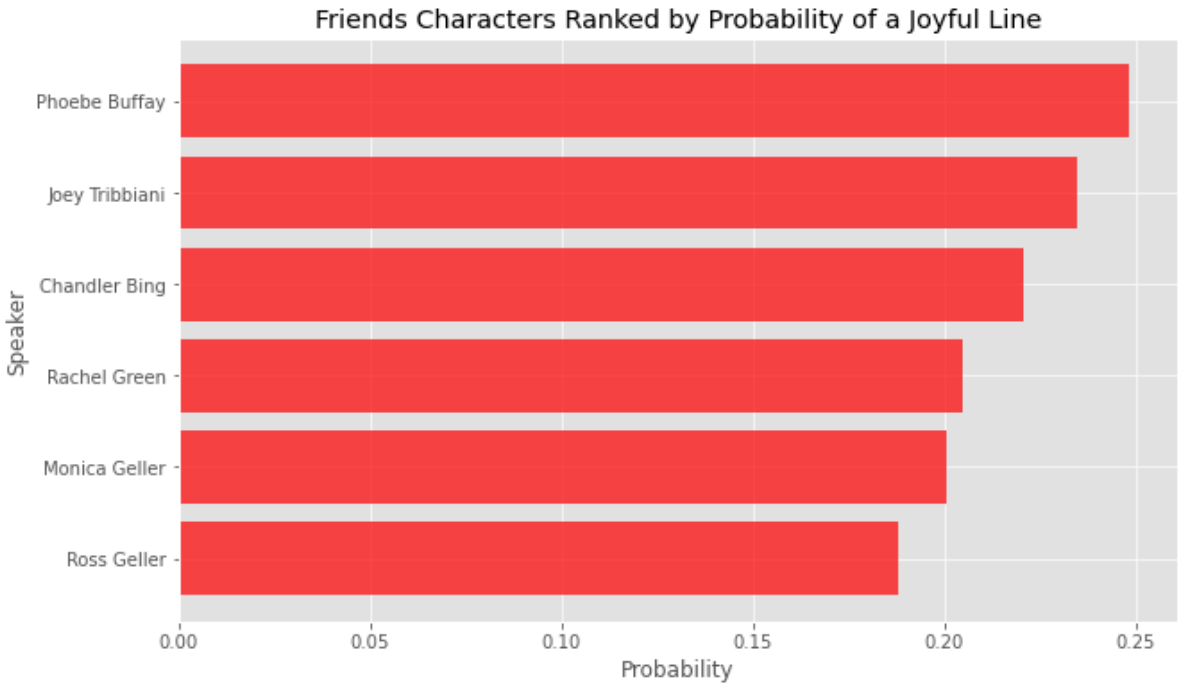
In [72]:
```
seven_emotions = main_emotions.get('emotion').unique()
for emotion in seven_emotions:
    print('Characters in descending order of '+emotion+': \n', rank_speakers
```



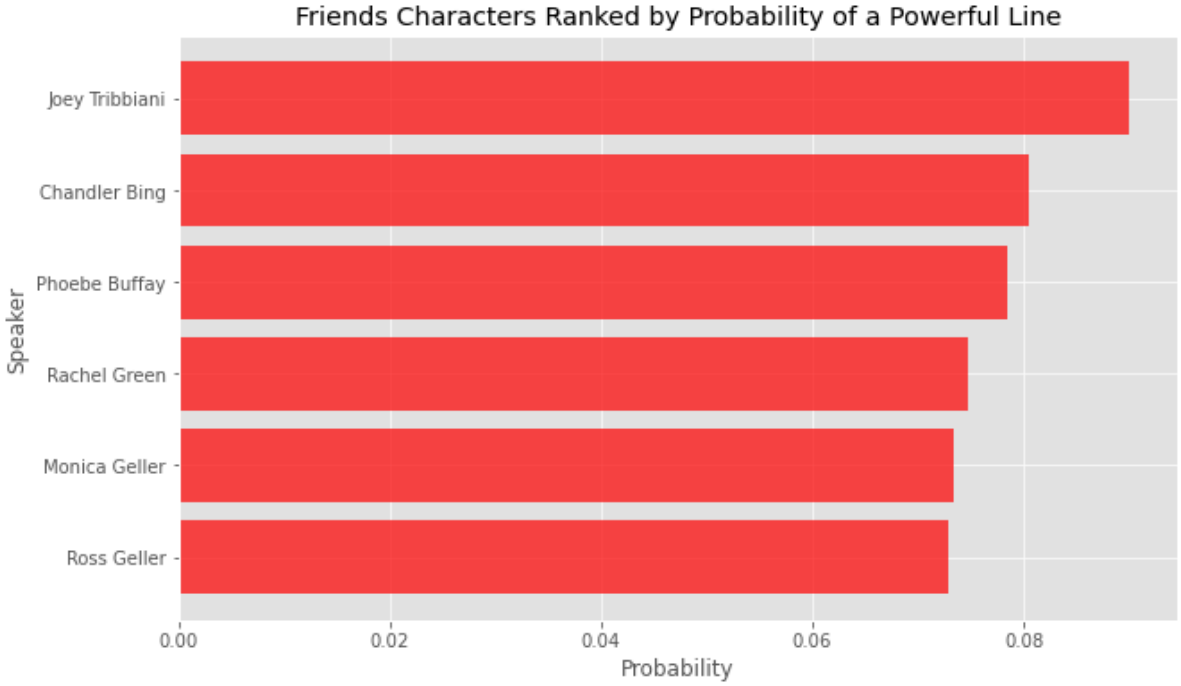

```
Characters in descending order of Mad:
 ['Rachel Green' 'Monica Geller' 'Chandler Bing' 'Ross Geller'
 'Joey Tribbiani' 'Phoebe Buffay']
```



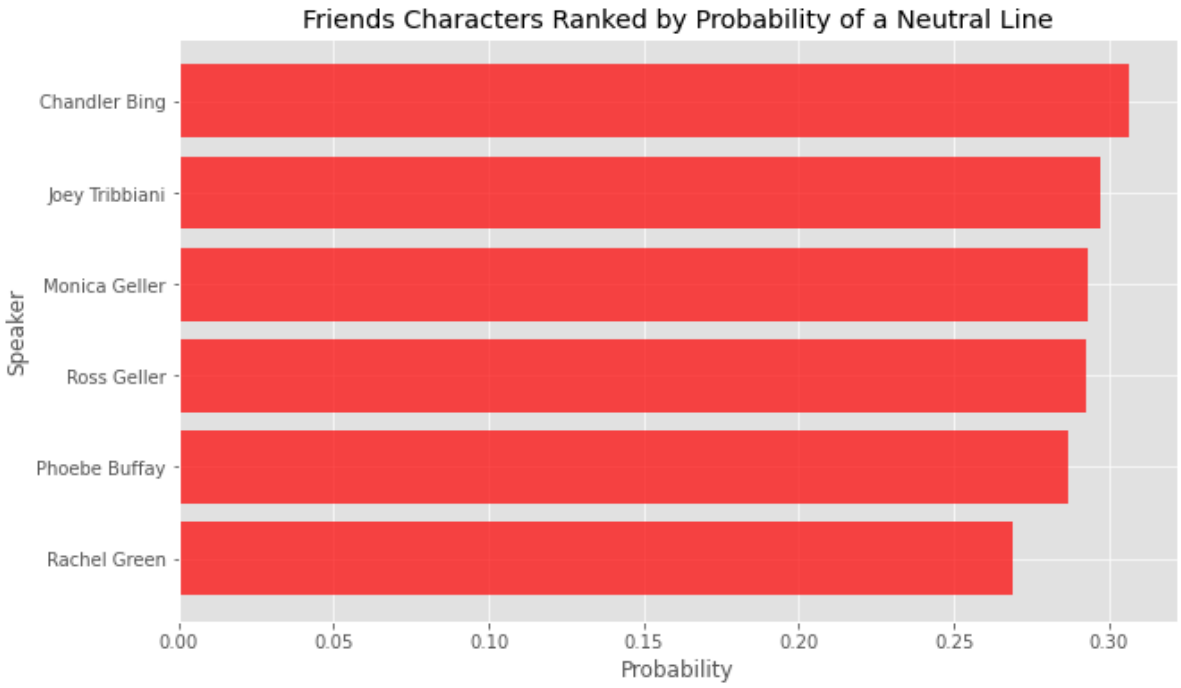

```
Characters in descending order of Sad:
 ['Phoebe Buffay' 'Monica Geller' 'Rachel Green' 'Ross Geller'
 'Joey Tribbiani' 'Chandler Bing']
```



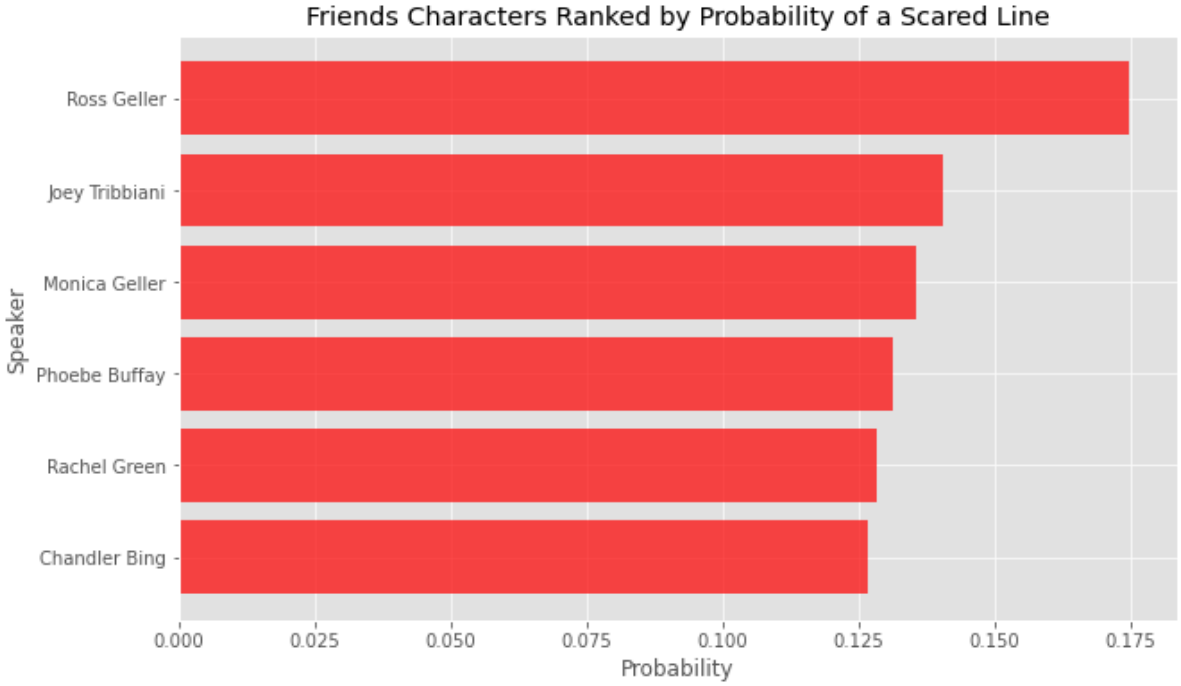

```
Characters in descending order of Joyful:
 ['Phoebe Buffay' 'Joey Tribbiani' 'Chandler Bing' 'Rachel Green'
 'Monica Geller' 'Ross Geller']
```



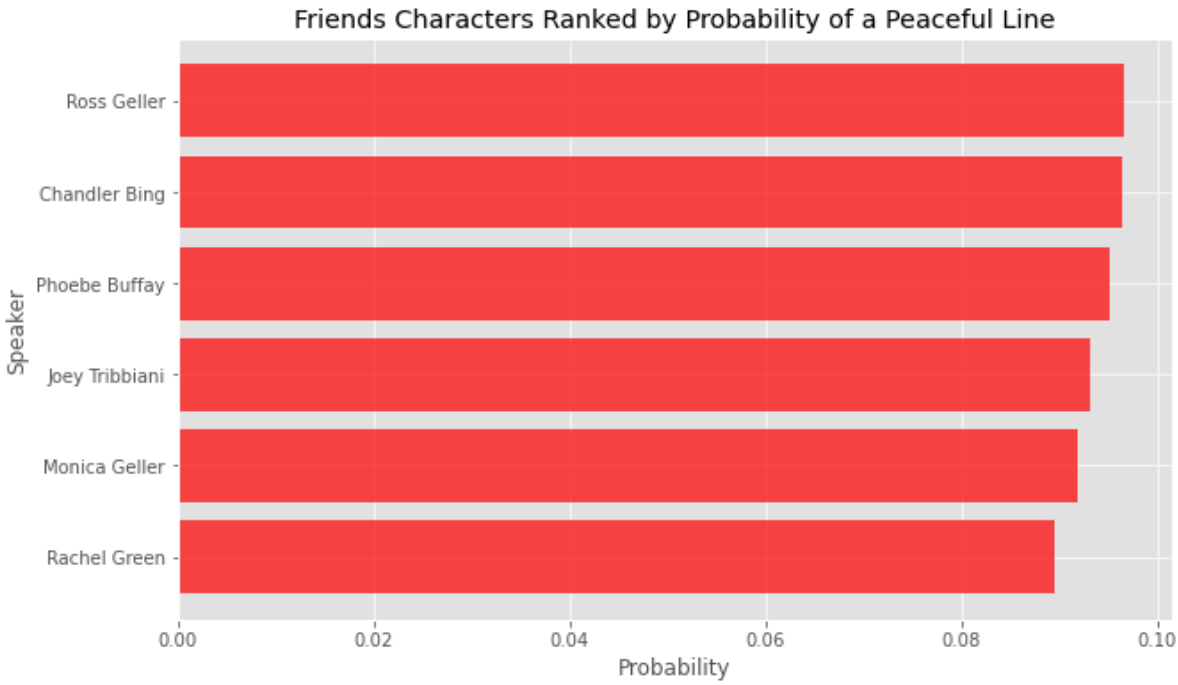

```
Characters in descending order of Powerful:
 ['Joey Tribbiani' 'Chandler Bing' 'Phoebe Buffay' 'Rachel Green'
 'Monica Geller' 'Ross Geller']
```




```
Characters in descending order of Neutral:
 ['Chandler Bing' 'Joey Tribbiani' 'Monica Geller' 'Ross Geller'
 'Phoebe Buffay' 'Rachel Green']
```




```
Characters in descending order of Scared:
 ['Ross Geller' 'Joey Tribbiani' 'Monica Geller' 'Phoebe Buffay'
 'Rachel Green' 'Chandler Bing']
```




```
Characters in descending order of Peaceful:
 ['Ross Geller' 'Chandler Bing' 'Phoebe Buffay' 'Joey Tribbiani'
 'Monica Geller' 'Rachel Green']
```

Now, you want to create a visualization that will represent the distribution of different emotions for each character. You plan to give this visualization to the writers you will hire for the reboot, as they will need to understand the emotional range of each character to write a reboot that is true to the spirit of the original *Friends* sitcom.

Question 4.7. Your task is to create a function `emotions_by_speaker` that will take in a `speaker` and calculate the probability of that character saying a line of each of the seven emotions. The function should return a DataFrame, indexed by `'emotion'`, with one column `'probability'`, containing values that sum to 1. The rows should appear **alphabetically** by `'emotion'`

For example, `emotions_by_speaker('Chandler Bing')` should return this DataFrame:

	probability
emotion	
Joyful	0.220890
Mad	0.124429
Neutral	0.306507
Peaceful	0.096461
Powerful	0.080479
Sad	0.044521
Scared	0.126712

In [73]:
```
def emotions_by_speaker(speaker):
#     '''Returns an DataFrame of emotions and their associated probabilities
    spkr_lines = main_emotions[main_emotions.get('speaker') == speaker]
```

```
    spkr_lines2 = spkr_lines.groupby('emotion').count()
    spkr_lines3 = bpd.DataFrame().assign(probability = spkr_lines2.get('spea
    return spkr_lines3

# An example call to your function. Feel free to change this and try out oth
emotions_by_speaker('Chandler Bing')
```

Out[73]:

emotion	probability
Joyful	0.220890
Mad	0.124429
Neutral	0.306507
Peaceful	0.096461
Powerful	0.080479
Sad	0.044521
Scared	0.126712

In [74]:
```
grader.check("q4_7")
```

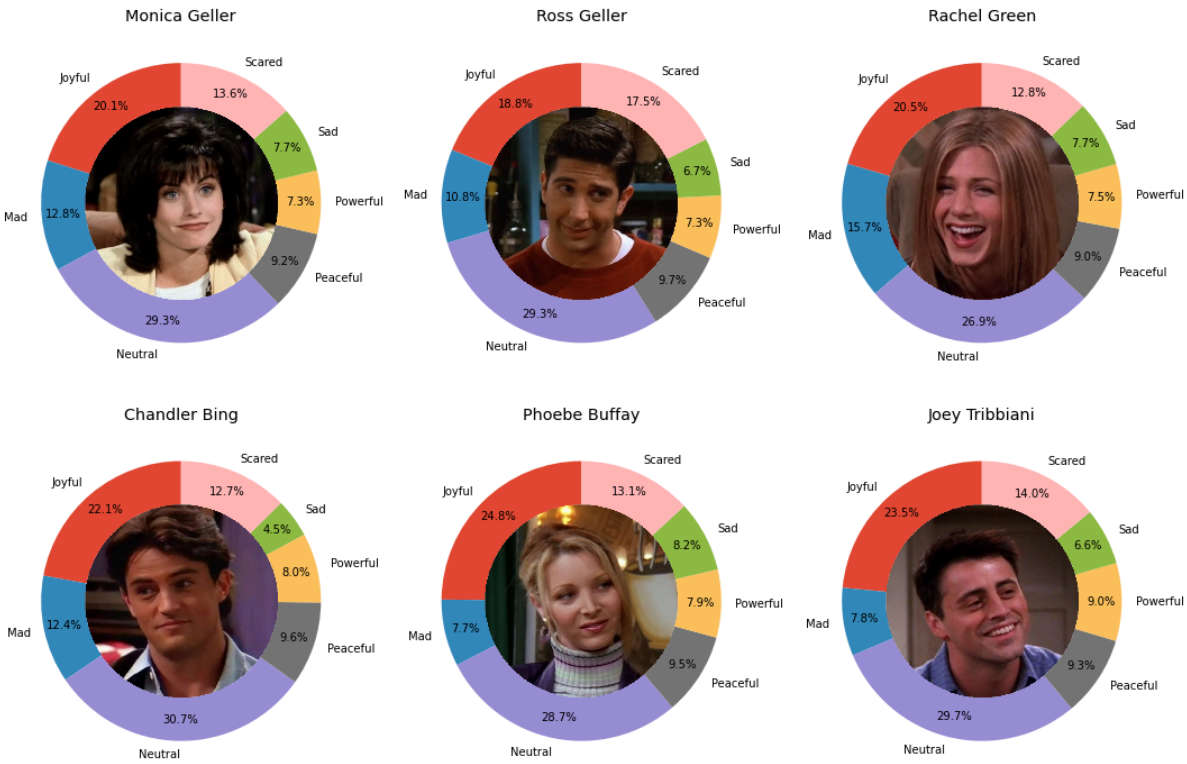
Out[74]:

q4_7 passed!

Question 4.8. Now, we'll use your function to create a visualization of speaker emotions, which should show the similarities and differences between the types of emotions each character uses when they speak. We've completed the visualization code for you, you just need to fill in a few missing lines as indicated by the comments. You do not need to know how the code for this visualization works!

In [75]:
```
# Set characters to an array of the names of the six main characters.
characters = ['Monica Geller', 'Ross Geller', 'Rachel Green', 'Chandler Bing

dfs = {}
for speaker in characters:
    # Call your emotions_by_speaker function on each speaker, and save the r
    dfs[speaker] = emotions_by_speaker(speaker)

def first_name(character):
    # Return the first name of the input character's name, converted to lowe
    # For example, RacheL Green -> rachel.
    return character.split()[0].lower()

fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.flatten()

max_height = max_width = 0
for character in characters:
    img = plt.imread(f'images/face_pics/{first_name(character)}.png')
    h, w, _ = img.shape
    max_height = max(max_height, h)
    max_width = max(max_width, w)

for ax, character in zip(axes, characters):
    df = dfs[character]
    pie, _, _ = ax.pie(df.get('probability'), labels=df.index, startangle=90
```

```
    ax.set_title(character)

    img = plt.imread(f'images/face_pics/{first_name(character)}.png')

    h, w, _ = img.shape
    radius = min(h, w) // 2
    y, x = np.ogrid[:h, :w]
    mask = ((y - h // 2) ** 2 + (x - w // 2) ** 2) > radius ** 2
    img[mask] = [1, 1, 1, 0]

    zoom_factor = max_height / h if h > w else max_width / w
    imagebox = OffsetImage(img, zoom=zoom_factor * 0.43)
    ab = AnnotationBbox(imagebox, (0.5, 0.5), frameon=False, pad=0, xycoords
    ax.add_artist(ab)

plt.tight_layout()
plt.show()
```



In [76]:
```
grader.check("q4_8")
```

Out[76]:

q4_8 passed!

This visualization will serve as a great template for the writers of the new reboot!

Section 5: The One with the Spoilers 🙊

(return to the outline)

In this section, we will use permutation testing to compare different groups of episodes from *Friends* in terms of their viewership and ratings. This might help us make some decisions about what kinds of episodes our reboot should include to boost viewership and ratings!

We'll start by looking at how earlier seasons of *Friends* (seasons 1-5) compared to later seasons (seasons 6-10) in terms of viewership. Check out the information you'll be working with in the `episodes` DataFrame:

In [77]:
```
episodes
```

Out[77]:

	season	episode	title	directed_by	written_by	air_date	us_views_milli
0	1	1	The Pilot	James Burrows	David Crane & Marta Kauffman	9/22/94	2
1	1	2	The One with the Sonogram at the End	James Burrows	David Crane & Marta Kauffman	9/29/94	2(
2	1	3	The One with the Thumb	James Burrows	Jeffrey Astrof & Mike Sikowitz	10/6/94	1(
3	1	4	The One with George Stephanopoulos	James Burrows	Alexa Junge	10/13/94	1(
4	1	5	The One with the East German Laundry Detergent	Pamela Fryman	Jeff Greenstein & Jeff Strauss	10/20/94	1(
...	...	...	...	...	...	...	
231	10	14	The One with Princess Consuela	Gary Halvorson	Story by: Robert CarlockTeleplay by: Tracy Reilly	2/26/04	2(
232	10	15	The One Where Estelle Dies	Gary Halvorson	Story by: Mark KunerthTeleplay by: David Crane...	4/22/04	2(
233	10	16	The One with Rachel's Going Away Party	Gary Halvorson	Andrew Reich & Ted Cohen	4/29/04	2
234	10	17	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5(
235	10	18	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5(

236 rows × 8 columns

You'll notice that we have data about viewership and rating for 236 episodes (10 seasons). Let's start by labeling the episodes in `episodes` to make it easier to answer our question:

> *Which half of the show has more viewers?*

Question 5.1. Create a DataFrame called `halves` which has the same data as `episodes` and an additional column called `'which_half'`. In this column, values

should be either `'first half'` (for seasons 1-5) or `'second half'` (for seasons 6-10).

In [78]:
```
def halving(value):
        if (value <= 5):
            return 'first half'
        elif (value <= 10):
            return 'second half'

halves = episodes.assign(which_half = episodes.get('season').apply(halving))
halves
```

Out[78]:

	season	episode	title	directed_by	written_by	air_date	us_views_milli
0	1	1	The Pilot	James Burrows	David Crane & Marta Kauffman	9/22/94	2
1	1	2	The One with the Sonogram at the End	James Burrows	David Crane & Marta Kauffman	9/29/94	2(
2	1	3	The One with the Thumb	James Burrows	Jeffrey Astrof & Mike Sikowitz	10/6/94	1
3	1	4	The One with George Stephanopoulos	James Burrows	Alexa Junge	10/13/94	1
4	1	5	The One with the East German Laundry Detergent	Pamela Fryman	Jeff Greenstein & Jeff Strauss	10/20/94	1
...	...	...	...	...	...	...	
231	10	14	The One with Princess Consuela	Gary Halvorson	Story by: Robert CarlockTeleplay by: Tracy Reilly	2/26/04	2
232	10	15	The One Where Estelle Dies	Gary Halvorson	Story by: Mark KunerthTeleplay by: David Crane...	4/22/04	2
233	10	16	The One with Rachel's Going Away Party	Gary Halvorson	Andrew Reich & Ted Cohen	4/29/04	2
234	10	17	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5
235	10	18	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5

236 rows × 9 columns

In [79]:
```
grader.check("q5_1")
```

Out[79]:

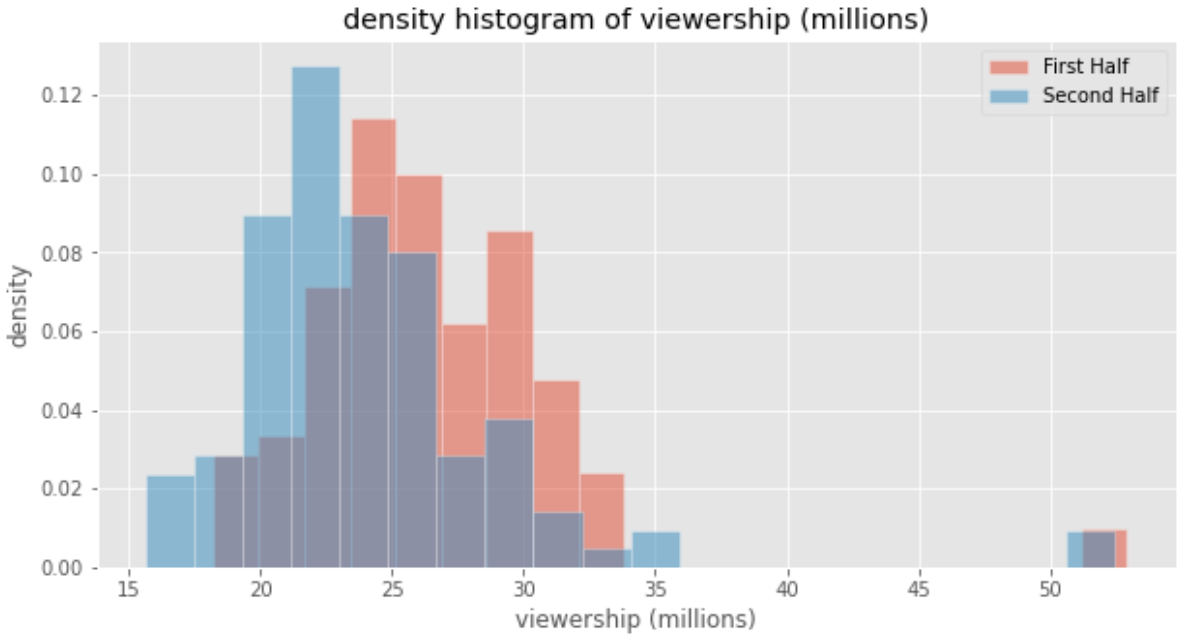
q5_1 passed!

Question 5.2. Before we start permutation testing, let's visualize the difference in viewership between the first half and second half of *Friends* episodes. Create an overlaid density histogram that compares the distribution of `'us_views_millions'` for first half episodes with that of second half episodes. Add a title, legend, and appropriate axis labels to your plot.

Then, assign the difference in **average** viewership (in millions) between the two groups to the variable `view_difference`. Here, `view_difference` should represent how much **greater** the first half viewership is than the second half, on average.

__Hint:__ Refer to the smoking and birth weight example in Lecture 22 for help creating the overlaid histogram! Think about what bin size is best suited to show the differences between these two distributions. You might want to tweak the bins until the differences become clearer.

In [80]:
```
#if ((halving(episodes, 'us_views_millions')) == 'first half'):
#    first_half_episodes = episodes.get('us_views_millions').plot(kind='hist

#elif ((halving(episodes, 'us_views_millions')) == 'second half'):
#    second_half_episodes = episodes.get('us_views_millions').plot(kind='his

first_half_views = halves[halves.get('which_half') == 'first half'].get('us_
second_half_views = halves[halves.get('which_half') == 'second half'].get('u

plt.figure(figsize=(10, 5))
plt.hist(first_half_views, bins=20, alpha=0.5, label='First Half', density=T
plt.hist(second_half_views, bins=20, alpha=0.5, label='Second Half', density

plt.title('density histogram of viewership (millions)')
plt.xlabel('viewership (millions)')
plt.ylabel('density')
plt.legend()
plt.show()

view_difference = first_half_views.mean() - second_half_views.mean()
view_difference

#add appropriate title, legend, and axis label
```



Out[80]: 2.322696370822854

In [81]:
```
grader.check("q5_2")
```

Out[81]:

q5_2 passed!

You should see that viewership was higher in the first half of *Friends* than in the second. Is this difference significant or was it just by chance? Let's perform a permutation test to find out.

Question 5.3. ⭐⭐ Let's clean up our DataFrame to focus only on the columns `'us_views_millions'` and `'which_half'`. To start, create a DataFrame `'views_by_group'` that only includes these two columns from `halves`.

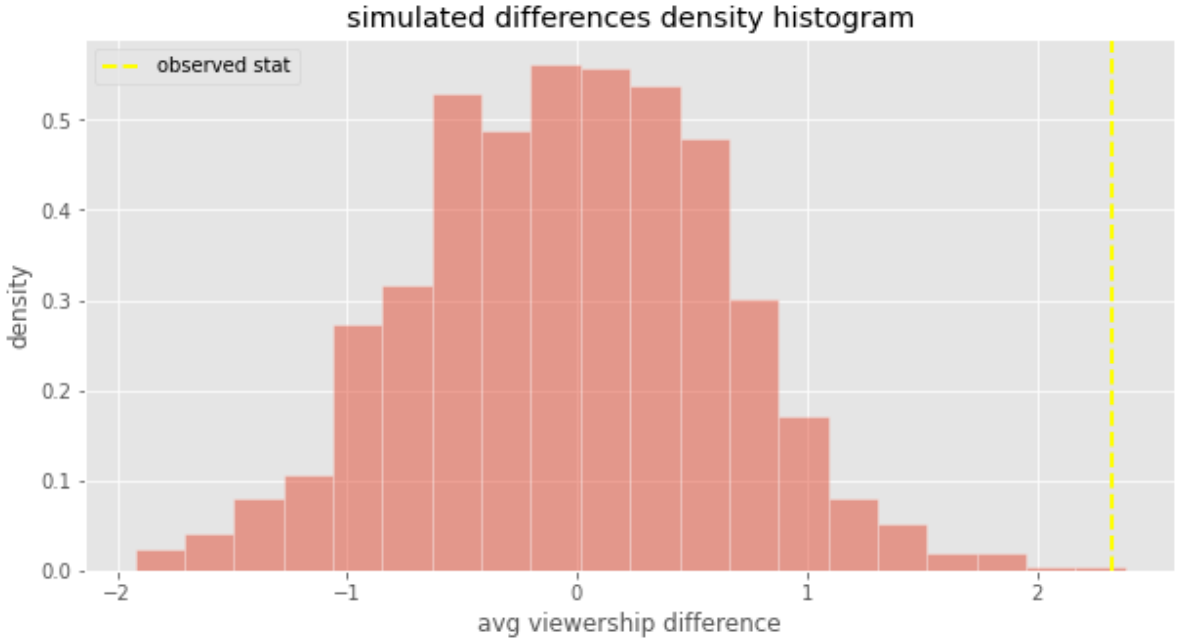
We would like to know whether the first half of episodes had significantly **more** viewership than the second half on average. Think about what your null and alternative hypotheses would be to answer this question.

In the previous question, you computed the *observed statistic*, the average number of views by which the first half exceeds the second half. Perform a permutation test, simulating 1000 differences and storing them in the array `simulated_differences`.

Once you're done, generate a density histogram of the simulated differences, which also shows your observed statistic for comparison. See Lecture 19 for an example of the type of plot you'll need to create.

In [82]:
```
#selected_columns = np.array(['us_views_millions', 'which_half'])
#views_by_group = halves.groupby[selected_columns]

#n_permutations = 1000
#for i in range (n_permutations):
#        simulated_differences = simulated_differences.append(np.random.perm
#simulated_differences.plot(kind='hist', bins=30, density=True, ec='w')


#$first_half_views = halves[halves.get('which_half') == 'first half'].get('u
```

```
#second_half_views = halves[halves.get('which_half') == 'second half'].get('


views_by_group = halves.get(['us_views_millions', 'which_half'])

first_half_avg = views_by_group[views_by_group.get('which_half') == 'first h
second_half_avg = views_by_group[views_by_group.get('which_half') == 'second
observed_diff = first_half_avg - second_half_avg

simulated_differences= np.array([])

n_permutations = 1000
for _ in range(n_permutations):
    shuffled_labels = np.random.permutation(views_by_group.get('which_half')
    shuffled = views_by_group.assign(which_half=shuffled_labels)

    first_half_avg1 = shuffled[shuffled.get('which_half') == 'first half'].g
    second_half_avg1 = shuffled[shuffled.get('which_half') == 'second half']

    simulated_difference = first_half_avg1 - second_half_avg1
    simulated_differences = np.append(simulated_differences, simulated_diffe

plt.figure(figsize=(10, 5))
plt.hist(simulated_differences, bins=20, alpha=0.5, ec='w', density=True)
plt.axvline(observed_diff, color='yellow', linestyle='dashed', linewidth=2,
plt.title('simulated differences density histogram')
plt.xlabel('avg viewership difference')
plt.ylabel('density')
plt.legend()
plt.show()

view_diff = observed_diff
view_diff
```



Out[82]: 2.322696370822854

In [83]:
```
grader.check("q5_3")
```

Out[83]:

q5_3 passed!

Question 5.4. Calculate a p-value for this permutation test using the observed statistic and the simulated differences, and assign the result to `p_halves`. Then, interpret these results at the 0.05 significance level by assigning `True` to the variable `significant_difference` if you believe we can **reject** the null hypothesis and `False` otherwise.

In [84]:
```
#p_halves = np.count_nonzero(simulated_differences)/n_permutations

p_halves = np.count_nonzero(simulated_differences >= observed_diff) / n_perm
#len(simulated_differences)

p_halves
```

Out[84]:
```
0.001
```

In [85]:
```
grader.check("q5_4_a")
```

Out[85]:
q5_4_a passed!

In [86]:
```
significant_difference = False
significant_difference
```

Out[86]:
```
False
```

In [87]:
```
grader.check("q5_4_b")
```

Out[87]:
q5_4_b passed!

We've answered our first question with permutation testing, which helped us figure out whether there was a significant difference in viewership between two groups of episodes. We can use this insight when determining how many episodes our reboot should have. We'll explore the length of our reboot more in Section 7 of this project.

Now, let's ask a different question that can be solved by the same tools:

> *Do people prefer episodes where at least two of the main characters are in a romantic relationship over episodes where there aren't any couples?*

First, we need to label which episodes have an ongoing romantic relationship.

⚠ Spoiler alert! ⚠

In the show, there are two couples that each date for a period of time.

- Ross and Rachel are in a relationship from Season 2, Episode 14 to Season 3, Episode 15, inclusive.
- Monica and Chandler are in a relationship from Season 4, Episode 24 to Season 10, Episode 18, inclusive, which is through the end of the show.

Question 5.5. ⭐⭐ Create a new DataFrame called `relationship` that has the same information as `episodes` and an additional column called `'couple'` that contains

`True` if the episode falls in one of the intervals mentioned above, and `False` otherwise.

__Hints:__

- To solve this problem with `apply`, you need to get around the limitation that you can only `apply` a function of one parameter. Consider writing a function that takes strings of the form `'x,y'` where `x` represents the season number and `y` represents the episode number.
- It is also possible to solve this problem without using `apply`. For a challenge, try both ways and see which way is simpler!

In [88]:
```
#intervals=np.array([{"start_season":1,"start_episode":2,"end_season":2,"enc
#def is_relationship(season_episode):
#    season,episode=season_episode.split(",")
#    for interval in intervals:
#        if(
#            (season>interval["start_season"] or (season==interval["start_se
#            and
#            (season<interval["end_season"] or (season==interval["end_seasor
#        ):
#            return True
#    return False

#relationship = episodes.assign(couple=episodes.get('season').get('episode')
#relationship

couple_val = []

for i in range(episodes.shape[0]):
    season = episodes.get('season').iloc[i]
    episode = episodes.get('episode').iloc[i]

    ross_rachel = (season == 2 and episode >= 14) or (season == 3 and episoc
    monica_chandler = (season == 4 and episode >= 24) or (5 <= season <= 9)

    couple_val.append(ross_rachel or monica_chandler)

relationship = episodes.assign(couple=couple_val)
relationship
```

Out[88]:

	season	episode	title	directed_by	written_by	air_date	us_views_milli
0	1	1	The Pilot	James Burrows	David Crane & Marta Kauffman	9/22/94	2
1	1	2	The One with the Sonogram at the End	James Burrows	David Crane & Marta Kauffman	9/29/94	2(
2	1	3	The One with the Thumb	James Burrows	Jeffrey Astrof & Mike Sikowitz	10/6/94	1
3	1	4	The One with George Stephanopoulos	James Burrows	Alexa Junge	10/13/94	1
4	1	5	The One with the East German Laundry Detergent	Pamela Fryman	Jeff Greenstein & Jeff Strauss	10/20/94	1
...	...	...	...	...	...	...	
231	10	14	The One with Princess Consuela	Gary Halvorson	Story by: Robert CarlockTeleplay by: Tracy Reilly	2/26/04	2
232	10	15	The One Where Estelle Dies	Gary Halvorson	Story by: Mark KunerthTeleplay by: David Crane...	4/22/04	2
233	10	16	The One with Rachel's Going Away Party	Gary Halvorson	Andrew Reich & Ted Cohen	4/29/04	2
234	10	17	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5
235	10	18	The Last One	Kevin S. Bright	Marta Kauffman & David Crane	5/6/04	5

236 rows × 9 columns

In [89]:
```
grader.check("q5_5")
```

Out[89]:

q5_5 passed!

Question 5.6. Now, let's state the hypotheses for our permutation test to answer the question:

> *Do episodes where characters are** in a relationship have higher ratings than those where characters **aren't** in a relationship?**

Consider the following choices for statements of the null and alternative hypotheses. For simplicity, let us call the relationship group "R" (for "relationship") and the non-relationship group "S" (for "single").

1. There is no difference in average episode rating between groups R and S.

2. There is a difference in average episode rating between groups R and S.
3. Group R has a higher rating than Group S, on average.
4. Group S has a higher rating than Group R, on average.

Set `null_choice` to a number 1 through 4, corresponding to your choice for the null hypothesis. Set `alternative_choice` to a number 1 through 4, corresponding to your choice for the alternative hypothesis.

In [90]:
```
null_choice = 1
alternative_choice = 3
```

In [91]:
```
grader.check("q5_6")
```

Out[91]:
q5_6 passed!

Question 5.7. Using the `relationship` DataFrame, compute the *observed statistic* representing the difference in mean rating between groups of episodes that have relationships (R) versus those that don't (S), in the order of R minus S. Store your result in the variable `couple_observed`.

In [92]:
```
#rgroup=np.array([])
#sgroup=np.array([])
#n=0
#for interval in intervals:
#        if(
#            (season>interval["start_season"] or (season==interval["start_se
#            and
#            (season<interval["end_season"] or (season==interval["end_seasor
#        ):
#    while interval["start_season"] <= interval["end_season"]:
#        while interval["start_episode"] <= interval["end_episode"]: #create
#            rgroup.extend(episodes.loc[episodes['Season']==interval["start_
#        n=n+1


#couple_observed = rgroup.mean()-sgroup.mean()
#couple_observed


rgroup = relationship[relationship.get('couple') == True]
sgroup = relationship[relationship.get('couple') == False]
couple_observed = rgroup.get('imdb_rating').mean() - sgroup.get('imdb_rating

couple_observed
```

Out[92]:
```
0.10969018932874341
```

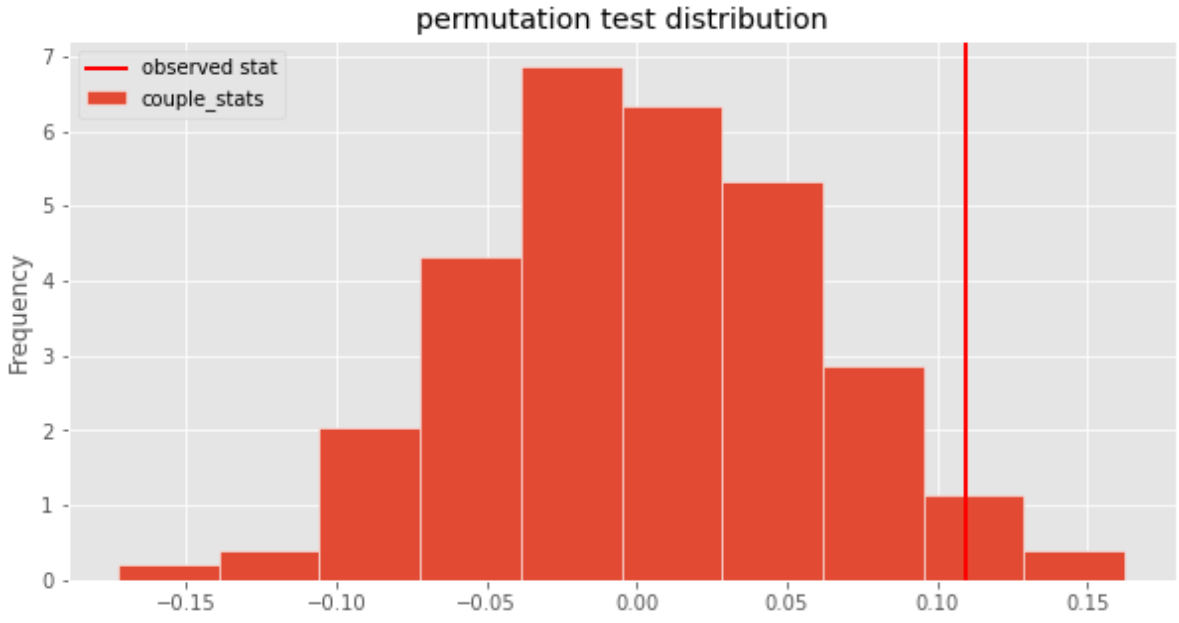
In [93]:
```
grader.check("q5_7")
```

Out[93]:
q5_7 passed!

Question 5.8. Now it's time to run the permutation test. Similarly to how you did in Question 5.3, simulate 1000 differences of group means, storing them in the array `couple_stats`. Then, plot your simulated differences in rating and show how they

compare to your observed statistic. Make sure you calculate your simulated differences by subtracting in the same order as you did when calculating your observed statistic.

In [94]:
```
#couple_stats = np.random.permutation(couple_observed)

#n_permutations = 1000
#for i in range (n_permutations):
#    couple_stats = couple_stats.append(np.random.permutation(couple_observe
#couple_stats.plot(kind='hist', bins=30, density=True, ec='w')

#n_permutations = 1000
#for i in range (n_permutations):
#        simulated_differences = simulated_differences.append(np.random.perm
#simulated_differences.plot(kind='hist', bins=30, density=True, ec='w')

# for inversion of theindicies condijtions

couple_stats = []

n_permutations
for _ in range(n_permutations):
    shuffled_couples = relationship.get('couple').sample(n=relationship.shap
    shuffled_relationship = relationship.assign(couple=shuffled_couples)

    relationship_avg = shuffled_relationship[shuffled_relationship.get('coup
    single_avg = shuffled_relationship[~shuffled_relationship.get('couple')]

    diff = relationship_avg - single_avg #easy lets goo
    couple_stats.append(diff)

bpd.DataFrame().assign(couple_stats=couple_stats).plot( kind='hist', density
plt.axvline(x=couple_observed, color='red', linewidth=2, label='observed sta
plt.legend()
plt.show()
#shuffled_couples
```



In [95]:
```
grader.check("q5_8")
```

Out[95]:

q5_8 passed!

Question 5.9. Calculate your p-value using the observed statistic and simulated differences, storing the result in `p_couple`. Assign `True` to the variable `couple_claim` if you believe we can **reject** the null hypothesis at the 0.05 significance level and `False` otherwise.

In [96]:
```
n_permutations = 1000
#p_couple = np.count_nonzero(couple_stats)/n_permutations
p_couple = np.count_nonzero(couple_stats >= couple_observed) / n_permutation
p_couple
```

Out[96]: 0.03

In [97]:
```
grader.check("q5_9_a")
```

Out[97]: **q5_9_a** passed!

In [98]:
```
couple_claim = True
couple_claim
```

Out[98]: True

In [99]:
```
grader.check("q5_9_b")
```

Out[99]: **q5_9_b** passed!

Although we can't determine from our permutation test whether characters being in a relationship *causes* better ratings, we can detect an association. You'll use this information when planning the storyline of your reboot to determine whether to include a romance angle! 💘

Section 6: The One with the Generated Episode Titles 🖨️

(return to the outline)

In this section, we'll create a tool that randomly generates episode titles for our reboot, based on the words used in the titles of *Friends* episodes. To start, let's see how *Friends* episode titles are formatted. The cell below saves all the *Friends* episode titles in an array called `titles` and prints a random sample of 10 titles.

In [100…
```
titles = np.array(episodes.get('title'))
for i in np.arange(10):
    print(np.random.choice(titles, replace=False))
```

```
The One with the Halloween Party
The One with Unagi
The One with the Metaphorical Tunnel
The One with the Home Study
The One with the Joke
The One with the Sonogram at the End
The One After Vegas
The One in Massapequa
The One Where Ross Can't Flirt
The One with the Two Parties
```

As you can see, most episode titles describe a major plot point of that episode and start with a phrase such as `'The One Where'` or `'The One with'`. In other words, the title of an episode is how you might describe the episode to someone else.

When generating episode titles for our reboot, we want to use many of the same words that appear in the titles of *Friends* episodes, so that our reboot titles stay true to the spirit of the original show and follow a similar structure. The simplest way to do this is to randomly select words, one at a time and independently, from the set of words included `titles`.

The variable `every_word` defined below is an array that contains all of the words in all of the *Friends* episode titles.

In [101…
```
every_word = np.array((' ').join(titles).split(' '))
every_word
```

Out[101]:
```
array(['The', 'Pilot', 'The', ..., 'The', 'Last', 'One'], dtype='<U14')
```

Some words appear in `every_word` much more frequently than others. We'd want these more common words, like `'The'` to appear more frequently in our generated titles, since they appear more frequently in actual *Friends* titles. Let's generate our titles randomly, one word at a time, so that each word is independently selected with a probability proportional to how often the word appears in a *Friends* title.

Question 6.1. Complete the implementation of the function `find_proportion` which takes as input a word, and returns the proportion of words in `every_word` that are equal to the input word. This should be case-sensitive!

Then calculate the proportion of words that are equal to `'Rachel'` and `'One'`, and save your results in `rachel_prop` and `one_prop`, respectively.

In [102…
```
def find_proportion(word):
    '''Returns the proportion of words equal to the input word.'''
    word_number = np.count_nonzero(every_word == word)
    word_prop = word_number/(len(every_word))
    return word_prop

rachel_prop = find_proportion("Rachel")
one_prop = find_proportion("One")
print('Rachel:', rachel_prop )
print('One:', one_prop)
```

```
Rachel: 0.010316875460574797
One: 0.17391304347826086
```

In [103…
```
grader.check("q6_1")
```

Out[103]:

q6_1 passed!

We can interpret each word's proportion of usages as a probability of that word being selected when we randomly generate a reboot title. For example, if we generate a title by independently selecting three words, we can calculate the probability of generating the title `'Rachel One Rachel'` as follows, using the multiplication rule for independent events:

$$P(\text{Rachel One Rachel}) = P(\text{Rachel}) * P(\text{One}) * P(\text{Rachel})$$

In [104…
```
prob_rachel_one_rachel = rachel_prop * one_prop * rachel_prop
prob_rachel_one_rachel
```

Out[104]:
```
1.851094248156703e-05
```

That's about a 1 in 50,000 chance that we'll generate the title `'Rachel One Rachel'`, which is so small primarily because `'Rachel'` is a rare word. The title `'One One One'` has a much larger chance of being generated, about 1 in 200, since `'One'` is a common word.

In [105…
```
prob_one_one_one = one_prop ** 3
prob_one_one_one
```

Out[105]:
```
0.0052601298594559046
```

While the strategy we've outlined gives a way of randomly generating reboot titles so that more common words are more likely to appear in the reboot title, it is unsatisfactory in many ways. Primarily, with this strategy, we can get titles like `'Rachel One Rachel'` and `'One One One'`, which not only don't make sense, but don't follow the structure of *Friends* episode titles that started with phrases like `'The One Where'`.

Instead of considering the probabilities of individual words, let's consider the probability of **pairs** of consecutive words. For example, the pair of words `'The One'` is quite common and the pair of words `'The Last'` is quite rare. This says that when we generate a reboot title randomly one word at a time, if we start with the word `'The'`, we should be more likely to follow it with `'One'` than `'Last'`.

We'll work on calculating the probability distribution of words that immediately follow a given word in *Friends* episode titles. As before, this is case-sensitive. For example, if the word is `'The'`, we'd want to look for want words that follow `'The'` only, not words that follow `'the'`.

Question 6.2. ⭐⭐ Complete the implementation of the function `next_word` which takes as input a single episode `title` and a `word`, both strings. The function should return an array containing every word in `title` that appears immediately after `word`, or an empty array if there are no such words. This could happen either because the `word` is the last word in `title`, or because the `word` is not a part of `title` at all.

The output array can contain duplicates if the same word follows the input `word` multiple times.

Example behavior is shown below.

```
>>> next_word('The One Where the Monkey Gets Away', 'Monkey')
array(['Gets'])

>>> next_word('The One with the Chicken Pox', 'Pox')
array([])

>>> next_word('This is the final project of the course', 'the')
array(['final', 'course']

>>> next_word('This is the final project of the course and there is
also the final exam', 'the')
array(['final', 'course', 'final']
```

In [106…
```
np.arange(8)
```

Out[106]:
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

In [107…
```
def next_word(title, word):

    words = title.split()
    result = np.array([])


    for i in np.arange(len(words)-1):
        if words[i] == word:
            result = np.append(result, words[i+1])
    return result



# An example call to your function. Feel free to change this and try out oth
next_word('The One Where the Monkey Gets Monkey Away', 'Monkey')
```

Out[107]:
```
array(['Gets', 'Away'], dtype='<U32')
```

In [108…
```
grader.check("q6_2")
```

Out[108]:

q6_2 passed!

Question 6.3. Now that we know how to find the subsequent words in a single title, let's extend what we've done to find the subsequent words across all titles in `titles`. Complete the function `words_following`, which takes as input a single `word` and returns an array of all words that immediately follow `word` in a *Friends* episode title. To do this, you should call your `next_word` function once on each title.

As before, the output array can contain duplicates if the same word follows `word` multiple times, and it can be empty if no word follows `word`.

__Hint:__ You can use `np.append` to append a whole array of values onto an existing array, in the same way that you would append a single value onto an existing array.

In [109…
```
def words_following(word):
    '''Given an input word, return an array of all words that
    immediately follow the input word in episode titles in titles.'''
    resulting_array = np.array([])
    for i in np.arange(len(titles)):
        titlespecific_array = next_word(titles[i], word)
        resulting_array = np.append(resulting_array, titlespecific_array)
    return resulting_array


# An example call to your function. Feel free to change this and try out oth
words_following('One')
```

Out[109]:
```
array(['with', 'with', 'with', 'with', 'with', 'with', 'Where', 'Where',
       'with', 'with', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'Where', 'with', 'with', 'with', 'with', 'Where', 'with',
       'with', 'Where', 'with', 'with', 'with', 'Where', 'with', 'with',
       'with', 'with', 'After', 'After', 'with', 'Where', 'Where',
       'Where', 'Where', 'Where', 'Where', 'with', 'with', 'with', 'with',
       'with', 'Where', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'Where', 'Where', 'with', 'Where', 'with', 'Where', 'with',
       'Without', 'with', 'with', 'with', 'with', 'with', 'with', 'with',
       'at', 'with', 'with', 'with', 'with', 'with', 'with', 'Where',
       'with', 'Where', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'with', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'After', 'with', 'Hundredth', 'Where', 'with', 'with',
       'Where', 'with', 'with', 'with', 'with', 'with', 'with', 'Where',
       'with', 'with', 'with', 'Where', 'Where', 'with', 'with', 'with',
       'in', 'in', 'After', 'Where', 'with', 'Where', 'with', 'on',
       'Where', 'with', 'Where', 'with', 'with', 'with', 'with', 'Where',
       'That', 'That', 'with', 'Where', 'with', 'with', 'Where', 'Where',
       'with', 'with', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'with', 'Where', 'with', 'with', 'with', 'Where', 'Where',
       'Where', 'with', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'with', 'with', 'After', 'with', 'Where', 'with', 'with',
       'with', 'with', 'with', 'with', 'with', 'with', 'Where', 'Where',
       'with', 'with', 'Where', 'with', 'in', 'with', 'with', 'with',
       'Where', 'Where', 'Where', 'Where', 'Proposes', 'Where', 'with',
       'with', 'with', 'with', 'with', 'with', 'with', 'with', 'Where',
       'with', 'Where', 'with', 'with', 'with', 'with', 'with', 'with',
       'with', 'with', 'with', 'in', 'in', 'After', 'Where', 'with',
       'with', 'Where', 'with', 'with', 'with', 'with', 'Where', 'Where',
       'with', 'Where', 'with', 'Where', 'with'], dtype='<U32')
```

In [110…
```
grader.check("q6_3")
```

Out[110]:

q6_3 passed!

Now that we know how to find all the words that follow a given word, we can create a probability distribution, where the probability of each subsequent word is just the proportion of times that word follows the given word.

Question 6.4. ⭐⭐ Complete the `find_prob_distribution` function which takes as input a single `word` and returns a DataFrame indexed by subsequent words that can immediately follow `word`, and with one column called `'prob'` that contains the proportion of times the subsequent word follows `word`. Sort the rows in descending order of `'prob'` so that the most likely words are at the top of the DataFrame.

For example, `find_prob_distribution('The')` should look like this:

word	prob
One	0.987288
Last	0.008475
Pilot	0.004237

In [111…
```
def find_prob_distribution(word):
    '''Returns a DataFrame containing the probability distribution
    of words that can follow the given input word.'''
    relevant_array = words_following(word)
    total_word_count = len(relevant_array)
    relevant_array = np.unique(relevant_array, return_counts = True)
    unique_words = relevant_array[0]
    unique_word_counts = relevant_array[1]
    unique_word_probablities = unique_word_counts/total_word_count
    prob_dist_df = bpd.DataFrame().assign(word = unique_words).assign(prob =
    return prob_dist_df
# An example call to your function. Feel free to change this to try out othe
find_prob_distribution('The')
```

Out[111]:

word	prob
One	0.987288
Last	0.008475
Pilot	0.004237

In [112…
```
grader.check("q6_4")
```

Out[112]:

q6_4 passed!

Now, we can use these probability distributions to generate titles randomly. We'll start each title with the word `'The'`, since all *Friends* episodes start with this word. Then, the next word will be `'One'` with probability 0.987288, or `'Last'` with probability 0.008475, or `'Pilot'` with probability 0.004237. Almost certainly, the next word will be `'One'`, and then we'll figure out the next word according to the probabilities in the DataFrame below.

In [113…
```
find_prob_distribution('One')
```

Out[113]:

word	prob
with	0.696581
Where	0.226496
After	0.025641
in	0.021368
That	0.008547
Hundredth	0.004274
Proposes	0.004274
Without	0.004274
at	0.004274
on	0.004274

Perhaps by random chance we'll end up with the next word being `'Proposes'` (though it's quite unlikely). To find the words that could follow `'Proposes'` along with their associated probabilities, we'll use this DataFrame:

In [114…
```
find_prob_distribution('Proposes')
```

Out[114]:

word	prob

Since this DataFrame has no rows, it means `'Proposes'` never appears in a *Friends* episode title with a word after it. That is, it only appears as the last word in an episode title. This means no word should come after it, so we're done generating our title, which in this example was `'The One Proposes'`.

When we generate titles based on pairs of words in this way, the probability of generating `'The One Proposes'` can be computed as:

$$P(\text{The One Proposes}) = P(\text{The}) * P(\text{One}|\text{The}) * P(\text{Proposes}|\text{One})$$

Since all episode titles start with `'The'`, $P(\text{The}) = 1$. The other probabilities are conditional probabilities based on the previous word. For example, $P(\text{One}|\text{The})$, which we read as "the probability of `'One'` given `'The'`", represents the probability of seeing the word `'One'` immediately after the word `'The'`. According to our `find_prob_distribution` function, this probability is quite high at 0.987288.

In [115…
```
find_prob_distribution('The') # This says P(One|The) = 0.987288.
```

Out[115]:

word	prob
One	0.987288
Last	0.008475
Pilot	0.004237

Similarly, we can find $P(\text{Proposes}|\text{One})$ is very small, only 0.004274.

In [116…

```
find_prob_distribution('One') # This says P(Proposes|One) = 0.004274
find_prob_distribution("in")
```

Out[116]:

word	prob
Barbados	0.285714
Vegas	0.285714
Massapequa	0.142857
Tulsa	0.142857
a	0.142857

Therefore, the probability of generating `'The One Proposes'` is:

$$\begin{aligned} P(\text{The One Proposes}) &= P(\text{The}) * P(\text{One}|\text{The}) * P(\text{Proposes}|\text{One}) && (1)\\ &= 1 * 0.987288 * 0.004274 && (2)\\ &\approx 0.004219 && (3) \end{aligned}$$

This is about 1 in 2000.

Question 6.5. Now that we know how to generate titles based on pairs of words, let's write a function that takes no inputs, and returns a randomly generated title by starting with the word `'The'`, then adding a new word based on the conditional probabilities of words that can follow `'The'`. We'll continue adding new words after each previous word, until we encounter a word that never had any words after it in an episode title (like `'Proposes'` in our example above). At that point, we'll stop generating new words.

Since we don't know how many words we'll be adding in advance, it's hard to implement this with a `for`-loop, since we don't know how many iterations we'll need. Instead, we'll use what's known as a `while`-loop, which runs continuously, with as many iterations as needed, until some stopping condition is met. In our case, we'll keep adding new words as long as the DataFrame returned by `find_prob_dist` has at least one row, since that means there are more words we can add to our title. Our stopping condition, therefore, is when `find_prob_dist` produces a DataFrame with no rows.

We've implemented the `while`-loop for you. Your task is to fill in the missing lines so that `generate_title` returns a randomly generated episode title. Some comments are included to help you.

In [117…
```
def generate_title():

    generated_title = 'The'
    prob_dist_df = find_prob_distribution('The')

    # Keep generating new words for our title so long as there are
    # words that can follow the most recent word we've added.
    while (prob_dist_df.shape[0] >= 1):

        # Set new_word to a random selection of the words represented in pro
        # chosen according to their probabilities in prob_dist_df.
        prob_dist_names = np.array(prob_dist_df.index)
        prob_dist_probs = np.array(prob_dist_df.get("prob"))
        new_word = np.random.choice(prob_dist_names, 1, p = prob_dist_probs)

        # Add onto your generated title in the variable generated_title.
        # Make sure to include spaces between words.
        generated_title = generated_title + " " + new_word

        # Update prob_dist_df so that it contains the probability
        # distribution of the word you just added to your title.
        prob_dist_df = find_prob_distribution(new_word)

    return generated_title

generate_title()
```

Out[117]:
```
'The One with the Memorial Service'
```

In [118…
```
grader.check("q6_5")
```

Out[118]:

q6_5 passed!

Question 6.6. Suppose you generate one title using the `generate_title()` function above. What is the probability that the title you generate is `'The One in Vegas'`? Set your answer to `vegas_prob`.

__Note:__ This is a probability question that you should answer based on the conditional probability of seeing one word given the previous. You should not do any simulation for this question.

In [119…
```
vegas_prob = 1 * 0.987288 * 0.021368 * 0.285714
#Simulation run above for the probably dist after in, that is where the 0.28
vegas_prob
```

Out[119]:
```
0.006027528253608578
```

In [120…
```
grader.check("q6_6")
```

Out[120]:

q6_6 passed!

Question 6.7. You want to know how frequently your `generate_title()` function generates a title that actually is a real title for a *Friends* episode. Based on a simulation with 1,000 trials, estimate the probability of a generated title being a real *Friends* title, and store the result in `prob_real`.

In [121…
```
generated_titles_array = np.array([])
# for i in np.arange(1000):
#     generated_title = generate_title()
#     generated_titles_array = np.append(generated_titles_array, generated_t
# count_same = np.count_nonzero(generated_titles_array == titles)
# prob_real = count_same/1000
# prob_real
same_title_count = 0
for i in np.arange(1000):
    title_gen = generate_title()
    if title_gen in titles:
        same_title_count = same_title_count + 1
prob_real = same_title_count/1000
prob_real
```

Out[121]: 0.566

In [122…
```
grader.check("q6_7")
```

Out[122]:

q6_7 passed!

You should find that there's a fairly high probability of generating a title that already exists. This happens because there aren't really that many words in the set of *Friends* episode titles. For example, if we randomly generate the word `'East'`, it's going to have to be followed up with `'German Laundry Detergent'` because the word `'East'` only appears once, which is in the episode title `'The One with the East German Laundry Detergent'`. With a relatively small body of text in *Friends* episode titles, there are many words that can only be followed by one thing. For simplicity, we'll ignore the fact that we are generating many existing episode titles, though we could significantly improve the situation by basing our titles not just on past *Friends* episode titles, but on a larger body of text, such as the full scripts of all *Friends* episodes. This improvement would come at the cost of much more computation!

Now, let's put all our hard work to use and generate episode titles for the first season of the reboot. Most seasons of *Friends* had 24 episodes, so you'll make sure your reboot starts out with a season of 24 episodes. Run the cell below to generate the titles for the first season of the reboot!

In [123…
```
for i in np.arange(1, 25):
    print('Reboot Episode', i, ':', generate_title() )
```

```
Reboot Episode 1 : The One Where Rachel Has a Duck
Reboot Episode 2 : The One Where Chandler Can't Flirt
Reboot Episode 3 : The One Where Chandler Crosses the Giant Poking Device
Reboot Episode 4 : The One with the Embryos
Reboot Episode 5 : The One with the Sharks
Reboot Episode 6 : The One with Mrs. Bing
Reboot Episode 7 : The One with the Kissing
Reboot Episode 8 : The One with the Joke
Reboot Episode 9 : The One with Phoebe's Rats
Reboot Episode 10 : The One with the Screamer
Reboot Episode 11 : The One After the Cheap Wedding Dresses
Reboot Episode 12 : The One Where Joey Moves In
Reboot Episode 13 : The One with Phoebe's Dad
Reboot Episode 14 : The One with Ross's Inappropriate Song
Reboot Episode 15 : The One with the Cuffs
Reboot Episode 16 : The One with the Red Sweater
Reboot Episode 17 : The One with the Superbowl
Reboot Episode 18 : The One with the Cooking Class
Reboot Episode 19 : The One with the Ultimate Fighting Champion
Reboot Episode 20 : The One with the Blind Dates Rachel Tells...
Reboot Episode 21 : The One Where They All the Baby on the Routine
Reboot Episode 22 : The One with the Blackout
Reboot Episode 23 : The One Where Chandler Gets Caught
Reboot Episode 24 : The One with the Worst Best Man Ever
```

In this section, you learned how to use pairs of words to generate text. More generally, sequences of n words (called n-grams) can be used to generate text in a similar way. For example, if $n = 3$, we would base each next word on the conditional probability of that word appearing after the previous two words. To start, we'd consider what words follow the phrase `'The One'` and how frequently. Perhaps we'd randomly choose the next word to be `'Where'`. We'd choose our next word based on words that follow `'One Where'`, and so on.

n-grams are used to capture the patterns and sequences of words in a language, which can be useful for a variety of natural language processing tasks, such as language modeling, text generation, machine translation, and more. While this is a simplified example, modern language models like GPT (Generative Pre-trained Transformer) use more advanced neural network architectures and techniques to capture long-range dependencies between words and generate more coherent and contextual text.

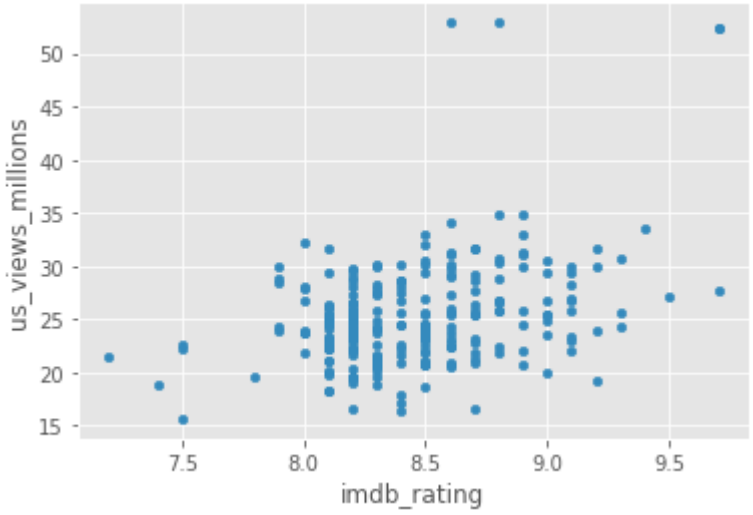
Section 7: The Last One 

(return to the outline)

In this section, you'll determine how many episodes your reboot should have to be successful. The first question is how should you measure success, in terms of views or in terms of ratings? You suspect that if an episode is good, it's only natural that it will get more views *and* higher ratings, right? So there should be a positive correlation between viewership and ratings. You decide to explore the connection between these two variables using regression.

Let's start by visualizing the data with a scatter plot to see if linear regression would make sense for this dataset.

In [124…
```
episodes.plot(kind='scatter', x='imdb_rating', y='us_views_millions');
```



Based on the scatter plot, it seems like there is a slight positive linear association, and so linear regression would be an appropriate tool. Let's continue!

Question 7.1. The first step will be to convert our data into standard units. Complete the function `standard_units` which takes in an array or Series of numerical data and returns an array with the values in standard units. Then use your function to standardize the `'imdb_rating'` and `'us_views_millions'` columns from `episodes`. Store the standardized arrays in the variables `rating_standard` and `views_standard`.

In [125…
```
def standard_units(sequence):
    '''Returns the input sequence as an array in standard units.'''
    # Convert the input to an array, if it is not already.
    sequence = np.array(sequence)
    return (sequence - np.mean(sequence)) / np.std(sequence)

rating_standard = standard_units(episodes.get('imdb_rating'))
views_standard = standard_units(episodes.get('us_views_millions'))
```

In [126…
```
grader.check("q7_1")
```

Out[126]:

q7_1 passed!

Question 7.2. Now that we know how to convert variables to standard units, we can calculate the correlation. Complete the function `correlation`, which should take in:

1. `df`, a DataFrame,
2. `independent`, the column label of the independent (x) variable, as a string, and
3. `dependent`, the column label of the dependent (y) variable, as a string.

The function should output the correlation between the two variables.

Then, use your function to compute the correlation between `'imdb_rating'` and `'us_views_millions'` and store your result in the variable `corr`. Check that the number you get looks reasonable based on the scatter plot above.

In [127…
```
def correlation(df, independent, dependent):
    '''Returns the correlation between the independent and dependent variabl
    x_su = standard_units(df.get(independent))
    y_su = standard_units(df.get(dependent))
    return (x_su * y_su).mean()

corr = correlation(episodes, 'imdb_rating', 'us_views_millions')
corr
```

Out[127]: 0.37741252071774684

In [128…
```
grader.check("q7_2")
```

Out[128]:

q7_2 passed!

Question 7.3. Now construct two functions, `reg_slope` and `reg_intercept`, which each take in the same three inputs as `correlation`. `reg_slope` should return the slope of the regression line and `reg_intercept` should return the intercept of the regression line, in original units.

Use your function to store the slope and intercept of the regression line for `'imdb_rating'` and `'us_views_millions'` in the variables `slope` and `intercept`.

In [129…
```
def reg_slope(df, independent, dependent):
    '''Returns the slope of the regression line in original units.'''
    r = correlation(df, independent, dependent)
    return r * np.std(df.get(dependent)) / np.std(df.get(independent))

def reg_intercept(df, independent, dependent):
    '''Returns the intercept of the regression line in original units.'''
    x_or = df.get(independent).mean()
    y_or = df.get(dependent).mean()
    return y_or - reg_slope(df, independent, dependent) * x_or

slope = reg_slope(episodes, 'imdb_rating', 'us_views_millions')
intercept = reg_intercept(episodes, 'imdb_rating', 'us_views_millions')
slope, intercept
```

Out[129]: (4.953290444993945, -16.542142752578002)

In [130…
```
grader.check("q7_3")
```

Out[130]:

q7_3 passed!

Question 7.4. Create a function called `predict` that takes in the same three inputs as `correlation`. `predict` should return an array of predicted values of the dependent variable calculated from the regression line.

Use your function to create an array of the predicted number of views, in millions, for each episode in the `episodes` DataFrame, based on the episode's rating. Save your answer as `predicted_views`.

In [131…

```
def predict(df, independent, dependent):
    '''Returns an array of predicted values of the dependent variable calcul
    x_su = standard_units(df.get(independent))
    r = correlation(df, independent, dependent)
    y_su_pred = r * x_su
    y_mean = df.get(dependent).mean()
    y_std = np.std(df.get(dependent))

    return y_mean + y_su_pred * y_std

predicted_views = predict(episodes, 'imdb_rating', 'us_views_millions')
predicted_views
```

Out[131]:
```
array([24.57016794, 23.57950985, 24.0748389 , 23.57950985, 25.56082603,
       23.57950985, 28.03747125, 23.57950985, 24.0748389 , 23.57950985,
       24.0748389 , 24.0748389 , 26.55148412, 24.57016794, 24.0748389 ,
       24.0748389 , 25.56082603, 27.04681316, 23.57950985, 22.58885176,
       22.58885176, 24.57016794, 26.55148412, 27.54214221, 25.56082603,
       24.0748389 , 25.06549699, 23.08418081, 24.57016794, 26.05615507,
       28.03747125, 25.56082603, 23.08418081, 23.08418081, 23.57950985,
       26.05615507, 27.04681316, 30.01878743, 27.54214221, 26.05615507,
       24.57016794, 25.56082603, 26.05615507, 24.0748389 , 24.0748389 ,
       28.03747125, 23.57950985, 24.0748389 , 25.06549699, 28.03747125,
       23.57950985, 23.57950985, 23.57950985, 28.5328003 , 24.57016794,
       25.06549699, 28.03747125, 23.57950985, 26.05615507, 24.0748389 ,
       24.0748389 , 22.58885176, 26.05615507, 28.5328003 , 24.57016794,
       25.06549699, 24.0748389 , 23.57950985, 26.55148412, 24.57016794,
       23.57950985, 23.57950985, 27.04681316, 28.5328003 , 23.57950985,
       25.56082603, 24.0748389 , 25.06549699, 25.56082603, 26.55148412,
       28.5328003 , 22.58885176, 23.57950985, 25.56082603, 30.51411647,
       24.0748389 , 24.0748389 , 25.56082603, 24.0748389 , 26.05615507,
       24.57016794, 26.55148412, 25.56082603, 19.12154845, 25.56082603,
       26.55148412, 29.02812934, 27.54214221, 28.03747125, 27.04681316,
       24.57016794, 27.04681316, 23.57950985, 25.06549699, 29.02812934,
       28.5328003 , 24.0748389 , 28.5328003 , 24.57016794, 23.57950985,
       31.50477456, 25.56082603, 26.05615507, 25.56082603, 23.08418081,
       26.55148412, 24.57016794, 25.56082603, 24.57016794, 27.04681316,
       28.5328003 , 26.55148412, 24.0748389 , 24.0748389 , 25.56082603,
       24.0748389 , 25.56082603, 25.06549699, 25.56082603, 29.02812934,
       26.05615507, 23.57950985, 23.57950985, 24.0748389 , 26.05615507,
       25.56082603, 25.56082603, 28.5328003 , 24.57016794, 24.57016794,
       20.11220654, 26.05615507, 28.03747125, 26.05615507, 27.04681316,
       29.52345839, 25.06549699, 24.57016794, 23.57950985, 23.57950985,
       23.57950985, 27.54214221, 25.56082603, 24.0748389 , 23.57950985,
       25.56082603, 26.05615507, 26.55148412, 25.06549699, 26.05615507,
       25.06549699, 26.55148412, 25.06549699, 25.06549699, 26.55148412,
       25.06549699, 20.60753558, 25.06549699, 27.54214221, 29.02812934,
       26.55148412, 28.5328003 , 26.55148412, 29.52345839, 23.57950985,
       25.56082603, 22.58885176, 27.04681316, 29.52345839, 23.57950985,
       23.08418081, 26.05615507, 26.05615507, 24.0748389 , 25.06549699,
       24.57016794, 23.57950985, 24.0748389 , 20.60753558, 24.0748389 ,
       24.0748389 , 24.57016794, 27.04681316, 27.54214221, 26.05615507,
       26.05615507, 24.0748389 , 24.0748389 , 25.56082603, 25.06549699,
       26.55148412, 27.04681316, 24.0748389 , 20.60753558, 23.08418081,
       23.08418081, 24.0748389 , 24.0748389 , 24.57016794, 24.0748389 ,
       25.56082603, 26.05615507, 23.57950985, 24.57016794, 24.0748389 ,
       22.09352272, 25.56082603, 26.55148412, 25.56082603, 27.04681316,
       27.04681316, 24.57016794, 24.0748389 , 24.0748389 , 24.57016794,
       27.54214221, 26.05615507, 23.08418081, 28.03747125, 27.54214221,
       25.56082603, 26.05615507, 25.56082603, 27.54214221, 31.50477456,
       31.50477456])
```

In [132…
```
grader.check("q7_4")
```

Out[132]:

q7_4 passed!

Question 7.5. Use the strategy for overlaying scatter plots described in this tutorial to create an overlaid scatter plot with:

- a blue dot 🔵 for each episode showing the ratings on the x-axis and the views on the y-axis (as in the scatter plot at the beginning of this section), and

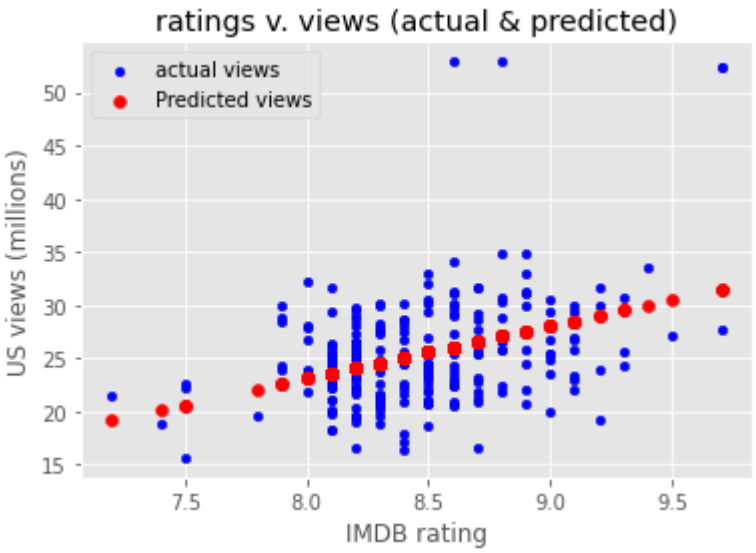
- a red dot 🔴 for each episode showing the ratings on the x-axis and the **predicted** views on the y-axis.

The red dots should form a straight line - that's the regression line!

__Note:__ This is the first time you've been asked to make an overlaid scatter plot, so you'll need to learn something new to answer this question. Read the linked tutorial carefully and try to mimic their example; you can learn how to do a lot of things this way!

In [133…

```
# Create your overlaid scatter plot here.
#episodes_with_predictions = episodes.assign('predicted_views' == predicted_

ax1 = episodes.plot(kind = 'scatter',
                    x = 'imdb_rating',
                    y = 'us_views_millions',
                    color = 'b',
                    label = 'actual views')
#plt.scatter(episodes['imdb_rating'], predicted_views, color='r', label='pre
#ax2 = episodes.plot(kind = 'scatter', x = 'imdb_rating', y = predicted_view
plt.scatter(
    episodes.get('imdb_rating'),
    predicted_views,
    color='r',
    label='Predicted views'
)


ax1.set_title("ratings v. views (actual & predicted)")
ax1.set_xlabel("IMDB rating")
ax1.set_ylabel("US views (millions)")
ax1.legend()

plt.show()
```



Question 7.6. Use the equation of the regression line to answer the following questions. Check that your answers are reasonable using the scatter plot above.

1. What is the predicted number of views, in millions, for an episode with an IMDb rating of 8.6? Save your answer as `question_1`.

2. An episode is predicted to have 30 million US views. What is its IMDb rating? Save your answer as `question_2`.
3. Suppose an episode in season one has an IMDb rating of d and an episode in season two has a IMDb rating of $d + 2$. How many millions more views would we predict the episode in season two to have compared to the episode in season one? Save your answer as `question_3`.

In [134…
```
question_1 = reg_slope(episodes, 'imdb_rating', 'us_views_millions') * 8.6 +
question_2 = (30 - reg_intercept(episodes, 'imdb_rating', 'us_views_millions
question_3 = reg_slope(episodes, 'imdb_rating', 'us_views_millions') * 2
print('Answer to question 1: ', question_1)
print('Answer to question 2: ', question_2)
print('Answer to question 3: ', question_3)

# y= mx + b v. y = (m+2)x + b
#(m)(x + 2) + b --- mx + b = mx + 2m + b - (mx + b) = 2m
```

```
Answer to question 1:  26.05615507436992
Answer to question 2:  9.39620708081351
Answer to question 3:  9.90658088998789
```

In [135…
```
grader.check("q7_6")
```

Out[135]:

q7_6 passed!

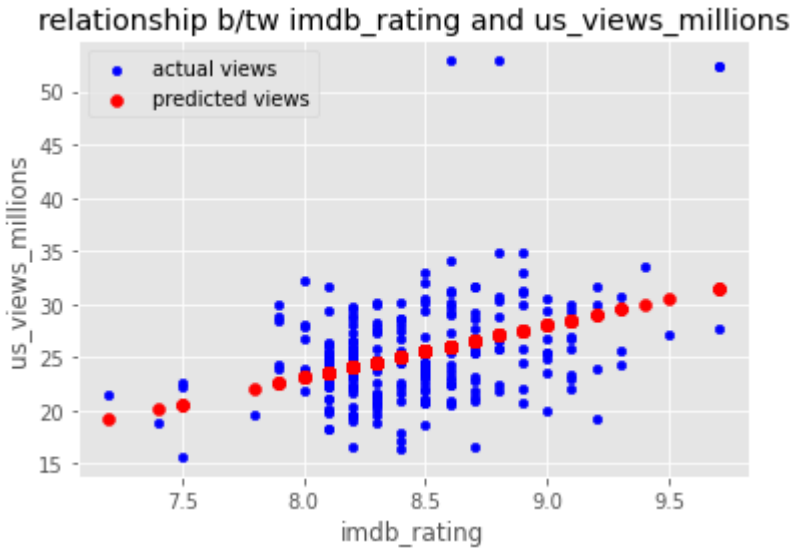
Question 7.7. Now that we have general code to calculate the regression line between any pair of variables in any DataFrame, let's generalize our code for the overlaid scatter plot so we can visualize relationships between other pairs of variables.

Complete the function `display_predictions` below. This function should take in the same three inputs as the `correlation` function, create an overlaid scatter plot similar to the one in Question 7.5, and return a string describing the correlation between the variables and the slope and intercept of the regression line.

Then, use your function to create a scatter plot and calculate the regression line that would allow you to predict IMDb rating based on viewership in millions. Store the result of your function call in the variable `rating_prediction`.

In [136…
```
def display_predictions(df, independent, dependent):
    '''Generates an overlaid scatter plot showing the relationship between t
    Returns a string describing the correlation and the slope and intercept

    ax1 = df.plot(kind = 'scatter',
                  x = independent,
                  y = dependent,
                  color = 'b',
                  label = 'actual views')

    slope = reg_slope(df, independent, dependent)
    intercept = reg_intercept(df, independent, dependent)
    predicted_values = slope * df.get(independent) + intercept

    plt.scatter(
        df.get(independent),
        predicted_values,
```

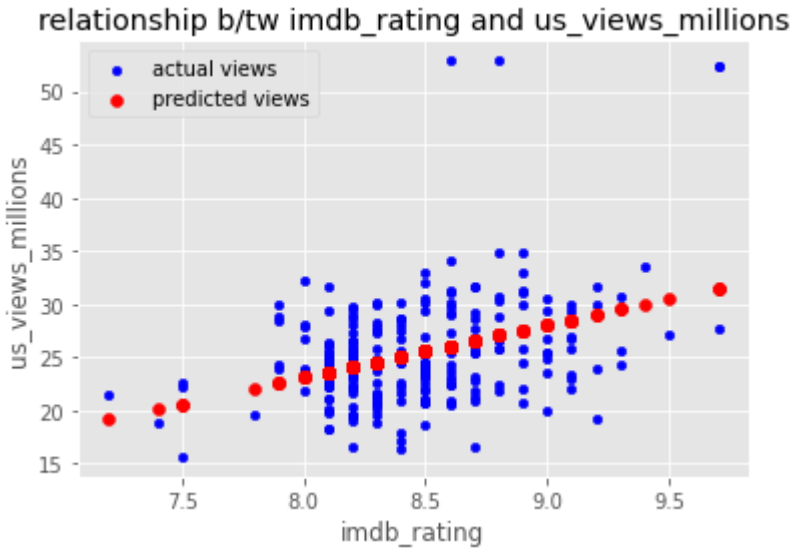
```
        color='r',
        label='predicted views'
    )


    ax1.set_title("relationship b/tw " + independent + " and " + dependent)
    ax1.set_xlabel(independent)
    ax1.set_ylabel(dependent)
    ax1.legend()

    plt.show()

    # We've provided the code for the return statement.
    return ('The correlation between {1} and {0} is {2}. ' +\
           ' The slope of the regression line is {3}.' + \
           ' The intercept of the regression line is {4}.')\
               .format(independent,
                       dependent,
                       str(round(correlation(df, independent, dependent), 2
                       str(round(slope, 2)),
                       str(round(intercept, 2)))

# Your function should produce the same scatter plot as in Question 7.5 on t
# Make sure to test it out on other inputs too.
display_predictions(episodes, 'imdb_rating', 'us_views_millions')
```



Out[136]: 'The correlation between us_views_millions and imdb_rating is 0.38. The s
lope of the regression line is 4.95. The intercept of the regression line
is -16.54.'

In [137…

```
rating_prediction = display_predictions(episodes, 'imdb_rating', 'us_views_m
rating_prediction
```



Out[137]:
```
'The correlation between us_views_millions and imdb_rating is 0.38.  The s
lope of the regression line is 4.95. The intercept of the regression line
is -16.54.'
```

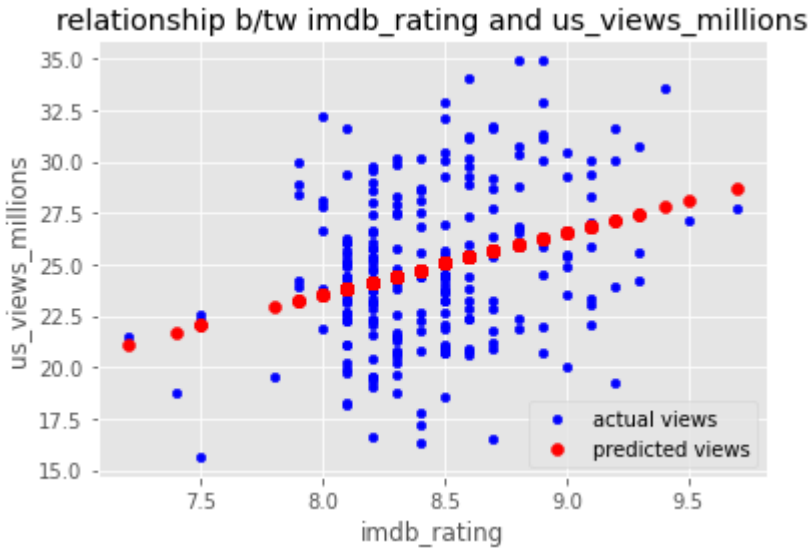
In [138…
```
grader.check("q7_7")
```

Out[138]:

q7_7 passed!

Question 7.8. There seem to be just a few episodes with 40 million views or more. Since outliers can have a big impact on the regression line, let's see how different our regression line would look if we eliminated these outliers. Again using the `display_predictions` function you just wrote, create a scatter plot and calculate the regression line that would allow you to predict IMDb rating based on viewership, but based only on episodes with less than 40 million views. Store the result of your function call in the variable `no_outliers`.

In [139…
```
episodes_no_outliers = episodes.loc[episodes.get('us_views_millions') < 40]
no_outliers = display_predictions(episodes_no_outliers, 'imdb_rating', 'us_v
no_outliers
```



Out[139]:
```
'The correlation between us_views_millions and imdb_rating is 0.3.  The sl
ope of the regression line is 3.03. The intercept of the regression line i
s -0.7.'
```

In [140…
```
grader.check("q7_8")
```

Out[140]:

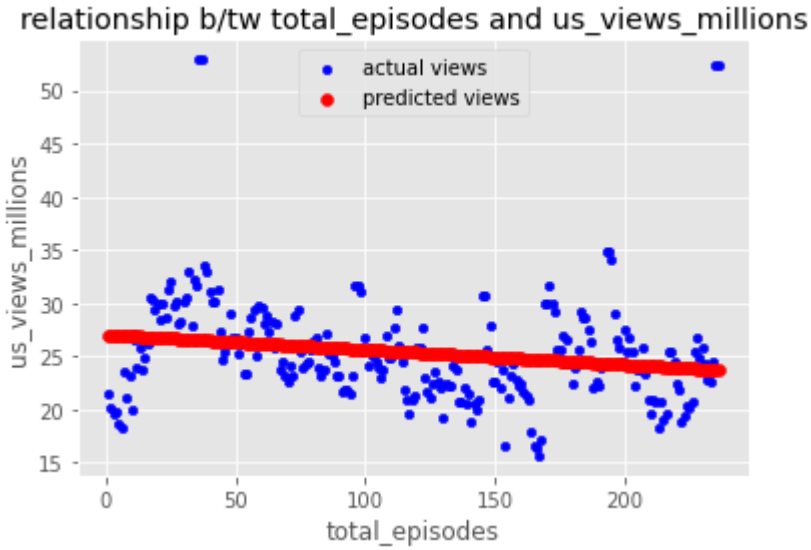
q7_8 passed!

Let's use some of the regression tools we've developed in this section to answer an important question about the reboot: how many episodes should it have? You are about to meet with the producers of the reboot to discuss this. To inform your discussion, you want to see how viewership of *Friends* changed as more episodes were made.

Question 7.9. Call your `display_predictions` function so that it displays the regression line predicting `'us_views_millions'` based on the number of *Friends* episodes produced. Your answer to this question should include outliers that were excluded in the previous question.

Hint: The `episodes` DataFrame does not have a column that counts the episode number from 1 to the total number of episodes. You'll need to add one!

In [141…
```
total_episodes_df = episodes.assign(total_episodes=range(1, episodes.shape[0
display_predictions(total_episodes_df, 'total_episodes', 'us_views_millions'
display_predictions
```



Out[141]:
```
<function __main__.display_predictions(df, independent, dependent)>
```

Since the slope of the regression line is negative, this means that viewership of *Friends* ultimately declined as more episodes were created. But this wasn't always the case. When the show was just starting off, new episodes were getting more views than past episodes. The regression line fit to just the first 30 episodes, for example, would have a strongly positive slope.

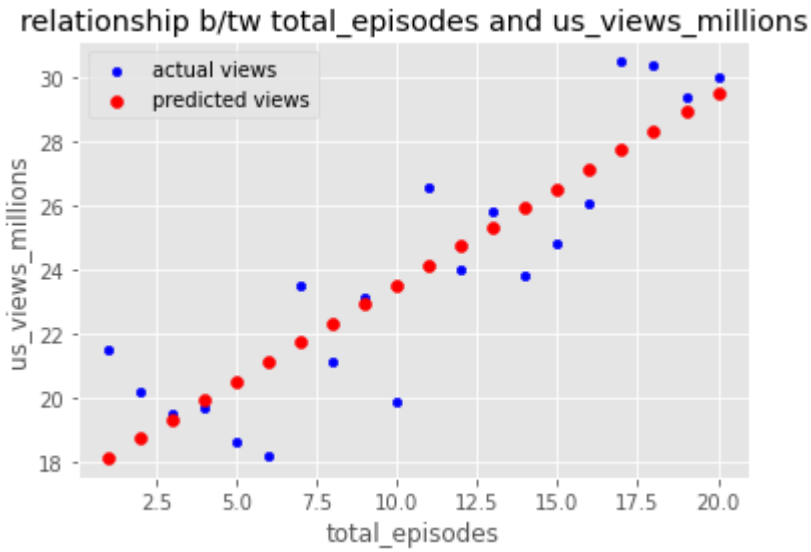
You interpret this data as meaning that maybe *Friends* went on for too long. You want your reboot to have few enough episodes that the excitement of the show doesn't wear off.

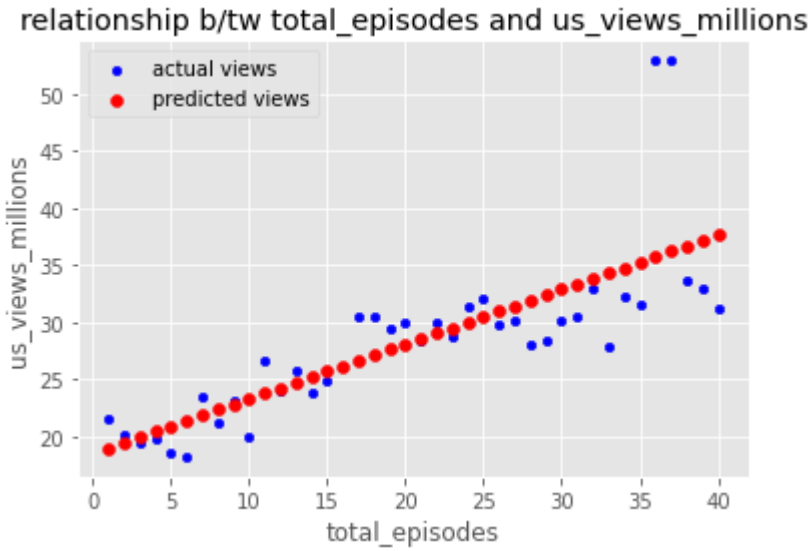
Question 7.10. Use your `display_predictions` function to display the regression lines predicting `'us_views_millions'` based on the first 20, 40, 60, 80, 100, and

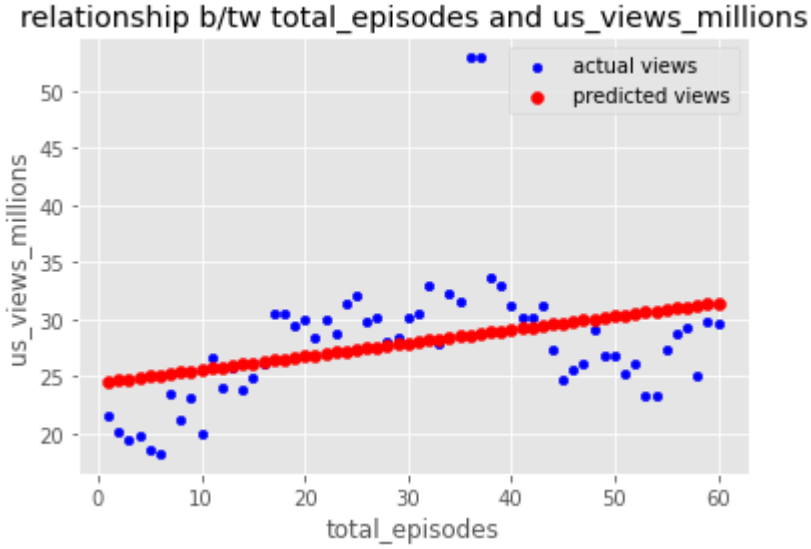
120 episodes. The largest such number of episodes with a positive slope is the number of episodes you think the reboot should have. Save this number of episodes as `reboot_length`. You can type this number in manually after looking at the plots.

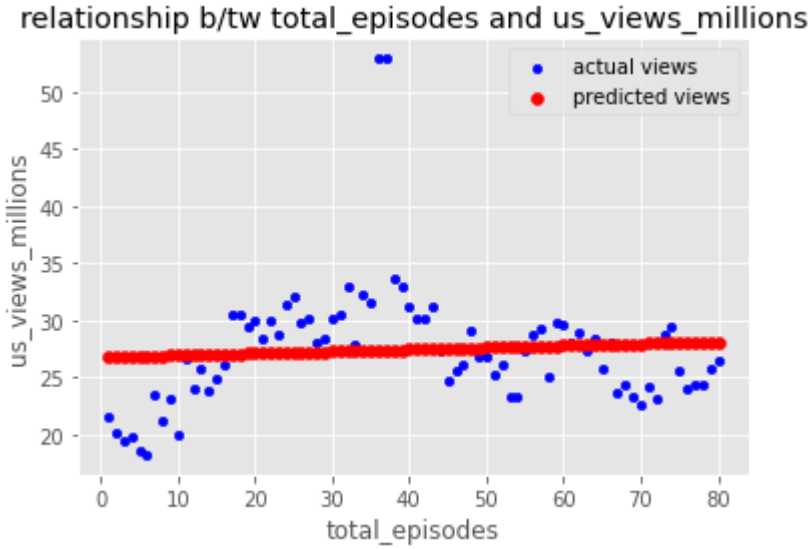
In [142…

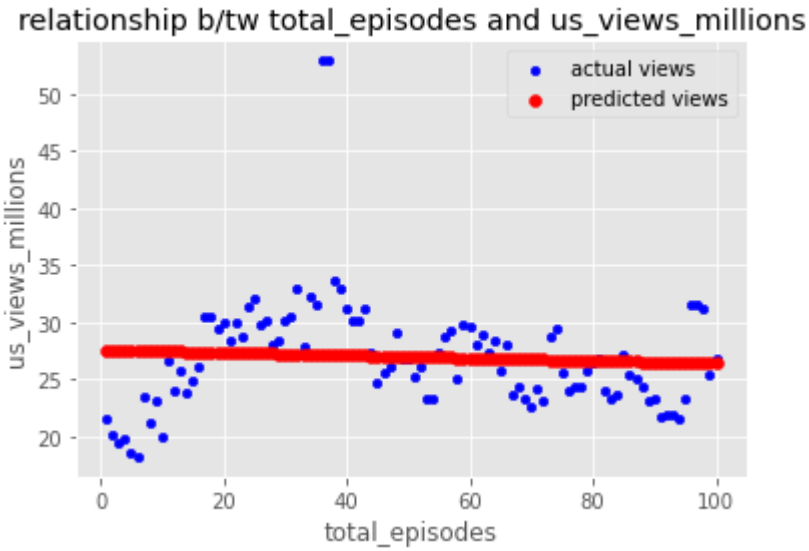
```
# Plot your regression lines here.
def reg_slope(df, independent, dependent):
    '''Returns the slope of the regression line in original units.'''
    r = correlation(df, independent, dependent)
    return r * np.std(df.get(dependent)) / np.std(df.get(independent))


df_1 = total_episodes_df.iloc[:20]
df_2 = total_episodes_df.iloc[:40]
df_3 = total_episodes_df.iloc[:60]
df_4 = total_episodes_df.iloc[:80]
df_5 = total_episodes_df.iloc[:100]
df_6 = total_episodes_df.iloc[:120]
display_predictions(df_1, 'total_episodes', 'us_views_millions')
display_predictions(df_2, 'total_episodes', 'us_views_millions')
display_predictions(df_3, 'total_episodes', 'us_views_millions')
display_predictions(df_4, 'total_episodes', 'us_views_millions')
display_predictions(df_5, 'total_episodes', 'us_views_millions')
display_predictions(df_6, 'total_episodes', 'us_views_millions')
s1 = reg_slope(df_1, 'total_episodes', 'us_views_millions')
s2 = reg_slope(df_2, 'total_episodes', 'us_views_millions')
print(s1)
print(s2)

reboot_length = 20
reboot_length
```

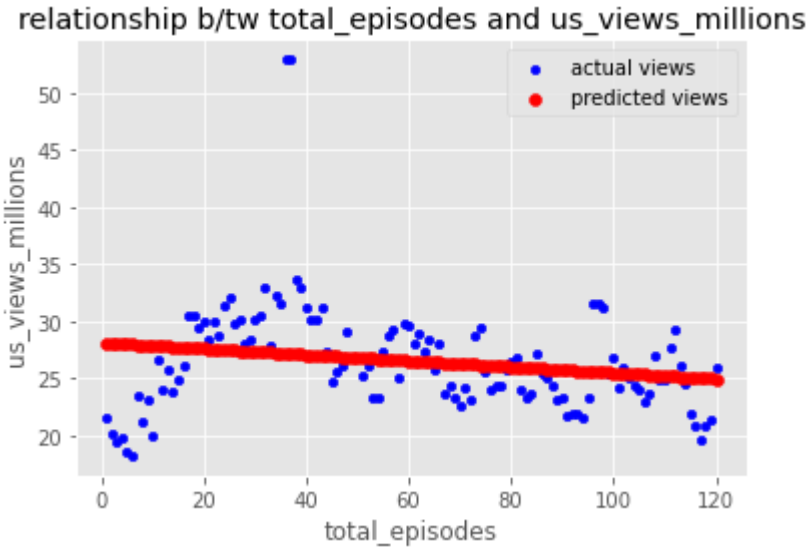



relationship b/tw total_episodes and us_views_millions
actual views
predicted views
us_views_millions
total_episodes


relationship b/tw total_episodes and us_views_millions
actual views
predicted views
us_views_millions
total_episodes


relationship b/tw total_episodes and us_views_millions
actual views
predicted views
us_views_millions
total_episodes






```
0.6005263157894737
0.47850844277673543
```

Out[142]: 20

In [143…
```
grader.check("q7_10")
```

Out[143]:

q7_10 passed!

Now that you know how many episodes your reboot should have, you've completed your plans for the reboot. Well done and best wishes for a successful show!

Finish Line 🏁

Congratulations! 🎉 You've completed the Final Project!

Citations: Did you use any generative artificial intelligence tools to assist you on this assignment? If so, please state, for each tool you used, the name of the tool (ex. ChatGPT) and the problem(s) in this assignment where you used the tool for help.

Please cite tools here.

To submit your assignment:

1. Select `Kernel -> Restart & Run All` to ensure that you have executed all cells, including the test cells.

   ⚠️ **Important!** We will allot 20 minutes of computer time to run your notebook. If your notebook takes longer than this to run, it may not pass the autograder! Select "Kernel -> Restart and Run All" to time how long your notebook takes. A notebook with correct answers should take less than 10 minutes.

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!

5. Stick around while the Gradescope autograder grades your work. Make sure you see that all tests have passed on Gradescope.

6. Check that you have a confirmation email from Gradescope and save it as proof of your submission.

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 assignment's due date has passed.

In [144…
```
grader.check_all()
```

Out[144]:
```
q1_1 results: All test cases passed!

q1_2 results: All test cases passed!

q1_3 results: All test cases passed!

q1_4 results: All test cases passed!

q1_5 results: All test cases passed!

q1_6_a results: All test cases passed!

q1_6_b results: All test cases passed!

q1_7 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!

q2_4 results: All test cases passed!

q2_5 results: All test cases passed!

q2_6 results: All test cases passed!

q2_7 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!

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_5 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_a results: All test cases passed!

q5_4_b results: All test cases passed!

q5_5 results: All test cases passed!

q5_6 results: All test cases passed!

q5_7 results: All test cases passed!

q5_8 results: All test cases passed!

q5_9_a results: All test cases passed!

q5_9_b results: All test cases passed!

q6_1 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!

q7_1 results: All test cases passed!

q7_10 results: All test cases passed!

q7_2 results: All test cases passed!

q7_3 results: All test cases passed!

q7_4 results: All test cases passed!

q7_6 results: All test cases passed!

q7_7 results: All test cases passed!

q7_8 results: All test cases passed!
```