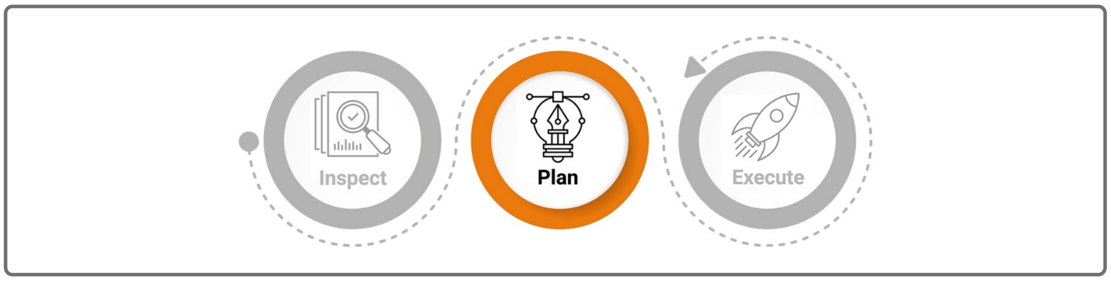
## 8.3.7 Remove Duplicate Rows

**Now** that the columns are tidied up, time to move on to the rows!

There are some data-cleaning tasks that are easier to perform on a DataFrame, such as removing duplicate rows. Luckily, we just created a process to turn our JSON data into a reasonable DataFrame. In fact, we'll start by removing duplicate rows.



Since we're going to be using the IMDb ID to merge with the Kaggle data, we want to make sure that we don't have any duplicate rows, according to the IMDb ID. First, we need to extract the IMDb ID from the IMDb link.

To extract the ID, we need to learn regular expressions.

**IMPORTANT**

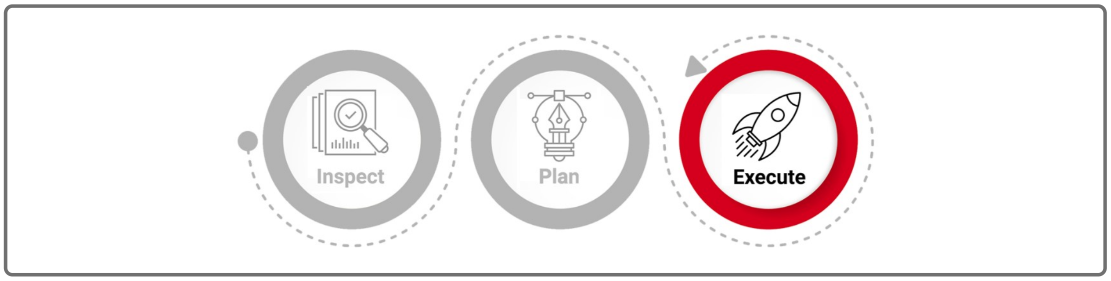
> **Regular expressions,** also known as **regex**, are strings of characters that define a search pattern. While the syntax might be new, this is a concept you're already familiar with in the noncoding world.
>
> For example, "MM/DD/YYYY" is a string of characters that defines a pattern for entering dates. You could say it's a regular expression that you can easily recognize because it follows a well-defined pattern. In the same way, "(###) ###-####" is a pattern for entering U.S. phone numbers. Regular expressions are just a more formal way of defining these kinds of patterns so that our code can find them. We'll expand on regular expressions in a later section. For now, just remember that they're used to search for patterns in text.

First, we'll use regular expressions in Pandas' built-in string methods that work on a Series object accessed with the `str` property. We'll be using `str.extract()`, which takes in a regular expression pattern. IMDb links generally look like "https://www.imdb.com/title/tt1234567/," with "tt1234567" as the IMDb ID. The regular expression for a group of characters that start with "tt" and has seven digits is `"(tt\d{7})"`.

- `"(tt\d{7})"` — The parentheses marks say to look for one group of text.
- `"(tt\d{7})"` — The "`tt`" in the string simply says to match two lowercase Ts.
- `"(tt\d{7})"` — The "`\d`" says to match a numerical digit.
- `"(tt\d{7})"` — The "`{7}`" says to match the last thing (numerical digits) exactly seven times.

Since regular expressions use backslashes, which Python also uses for special characters, we want to tell Python to treat our regular expression characters as a raw string of text. Therefore, we put an `r` before the quotes. We need to do this every time we create a regular expression string. We'll put the extracted IMDB ID into a new column. Altogether, the code to extract the IMDb ID looks like the following:

```
wiki_movies_df['imdb_id'] = wiki_movies_df['imdb_link'].str.extract(r'(tt\d{
```



Now we can drop any duplicates of IMDb IDs by using the `drop_duplicates()` method. To specify that we only want to consider the IMDb ID, use the `subset` argument, and set `inplace` equal to `True` so that the operation is performed on the selected dataframe. Otherwise, the operation would return an edited dataframe that would need to be saved to a new variable. We also want to see the new number of rows and how many rows were dropped. The whole cell should look like the following:

```
wiki_movies_df['imdb_id'] = wiki_movies_df['imdb_link'].str.extract(r'(tt\d{
print(len(wiki_movies_df))
wiki_movies_df.drop_duplicates(subset='imdb_id', inplace=True)
print(len(wiki_movies_df))
wiki_movies_df.head()
```

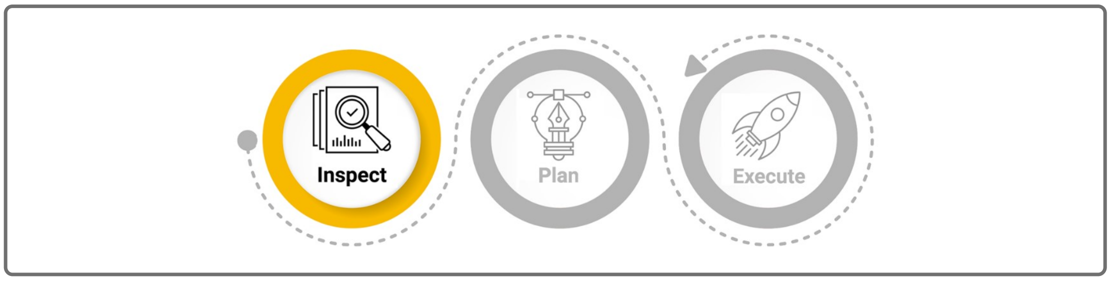
The output says there are now 7,033 rows of data. This means we haven't lost many rows, which is good.

## Remove Mostly Null Columns

Now that we've consolidated redundant columns, we want to see which columns don't contain much useful data. Since this is scraped data, it's possible many columns are mostly null.

**SKILL DRILL**

What code could you write to programmatically see how many null values are in each column? Could you do it in one line of code? Take a minute and think about it before trying it out in code.



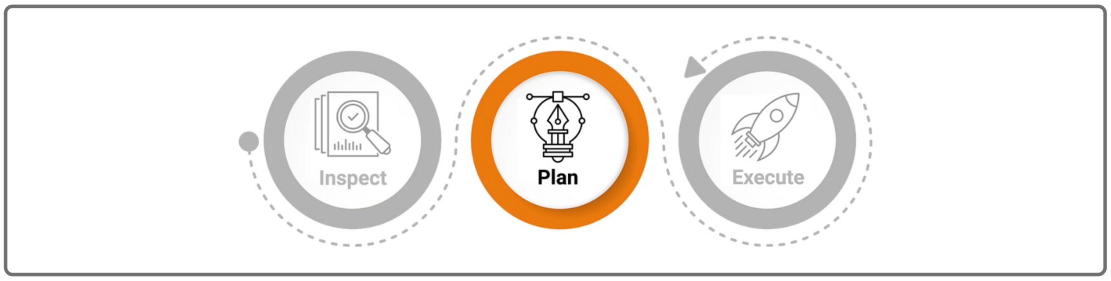
One way to get the count of null values for each column is to use a list comprehension, as shown below.

```
[[column,wiki_movies_df[column].isnull().sum()] for column in wiki_movies_df
```

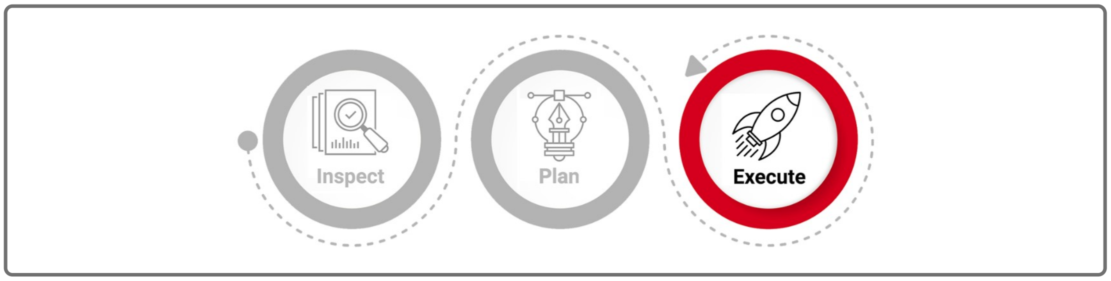
The output would be:

```
[['url', 0],
 ['year', 0],
 ['imdb_link', 0],
 ['title', 1],
 ['Based on', 4852],
 ['Starring', 184],
 ['Narrated by', 6752],
 ['Cinematography', 691],
 ['Release date', 32],
 ['Running time', 139],
 ['Country', 236],
 ['Language', 244],
 ['Budget', 2295],
 ['Box office', 1548],
 ['Director', 0],
 ['Distributor', 357],
 ['Editor(s)', 548],
 ['Composer(s)', 518],
 ['Producer(s)', 202],
 ['Production company(s)', 1678],
```

You could also use a `for` loop and a print statement.



Either way, we can see about half the columns have more than 6,000 null values. We could remove them by hand, but it's better to do it programmatically to make sure we don't miss any. Let's make a list of columns that have less than 90% null values and use those to trim down our dataset.



We just have to tweak our list comprehension.

```
[column for column in wiki_movies_df.columns if wiki_movies_df[column].isnul
```

That will give us the columns that we want to keep, which we can select from our Pandas DataFrame as follows:

```
wiki_columns_to_keep = [column for column in wiki_movies_df.columns if wiki_
wiki_movies_df = wiki_movies_df[wiki_columns_to_keep]
```

**IMPORTANT**

> You may have noticed that the "alt_titles" column we created earlier was deleted by this bit of code. It might feel like all that work we did was futile, but it's not. It's possible that all of the alternate title columns individually had less than 10% non-null values, but collectively had enough data to keep. We wouldn't know that unless we put in that work.
>
> This is normal for data cleaning because it's an iterative process. Sometimes the hard work you put in doesn't seem to make it to the final product, but don't worry, it's in there.

And with that, we've reduced 191 messy columns down to 21 useful, data-filled columns. That's awesome data-cleaning work!