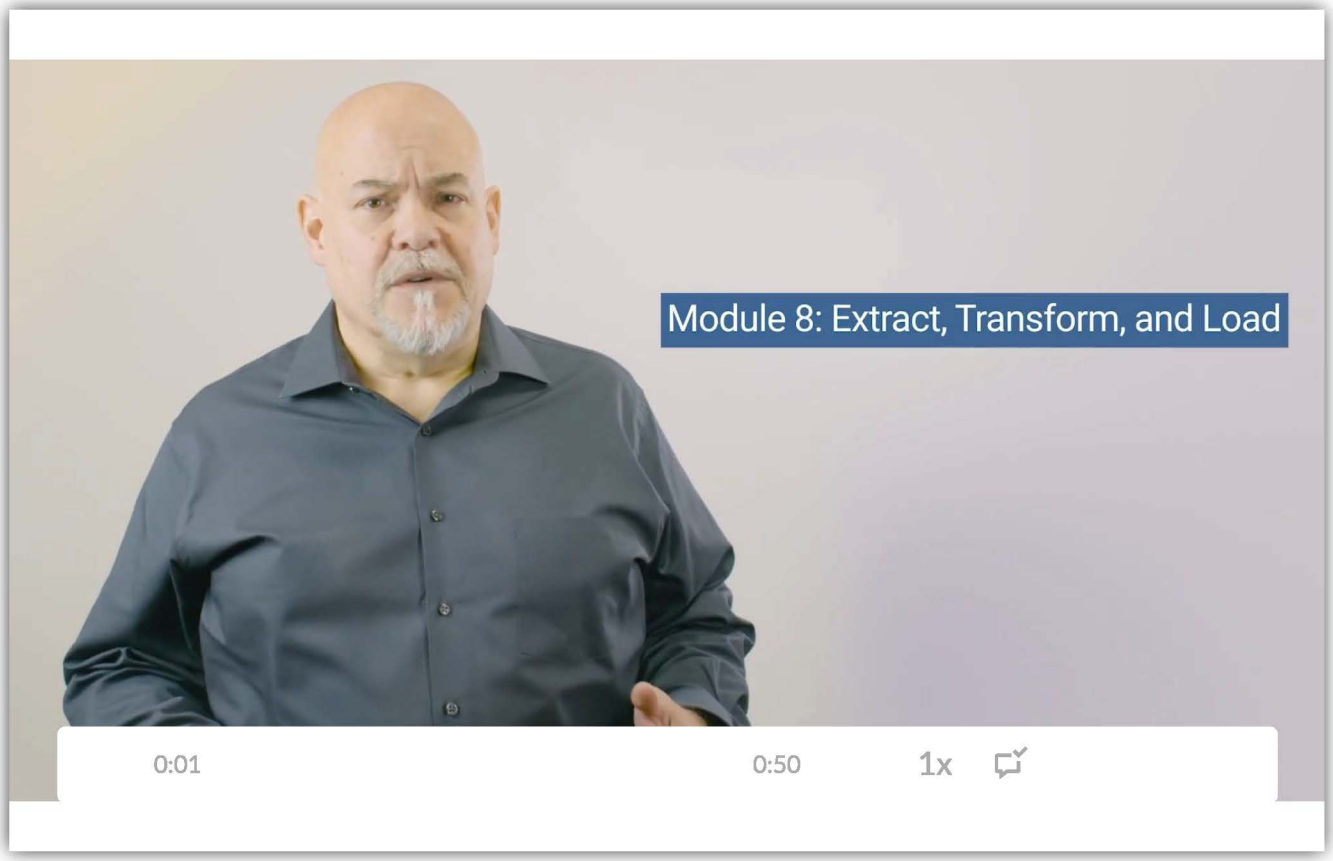
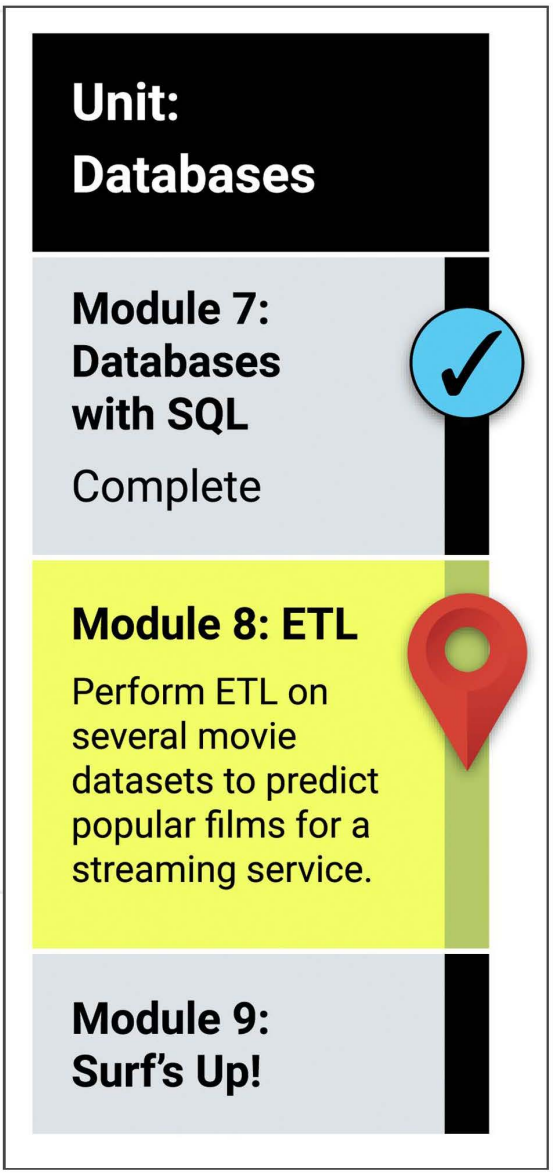
8.0.1: Use ETL to Collect, Import, and Process Data



8.0.2: Module 8 Roadmap

Looking Ahead

In this module, you'll learn how to use the Extract, Transform, Load (ETL) process to create data pipelines. A data pipeline moves data from a source to a destination, and the ETL process creates data pipelines that also transform the data along the way. Analysis is impossible without access to good data, so creating data pipelines is often the first step before any analysis can be performed. Therefore, understanding ETL is an essential skill for data analysis.



What You Will Learn

By the end of this module, you will be able to:

- Create an ETL pipeline from raw data to a SQL database.
- Extract data from disparate sources using Python.
- Clean and transform data using Pandas.

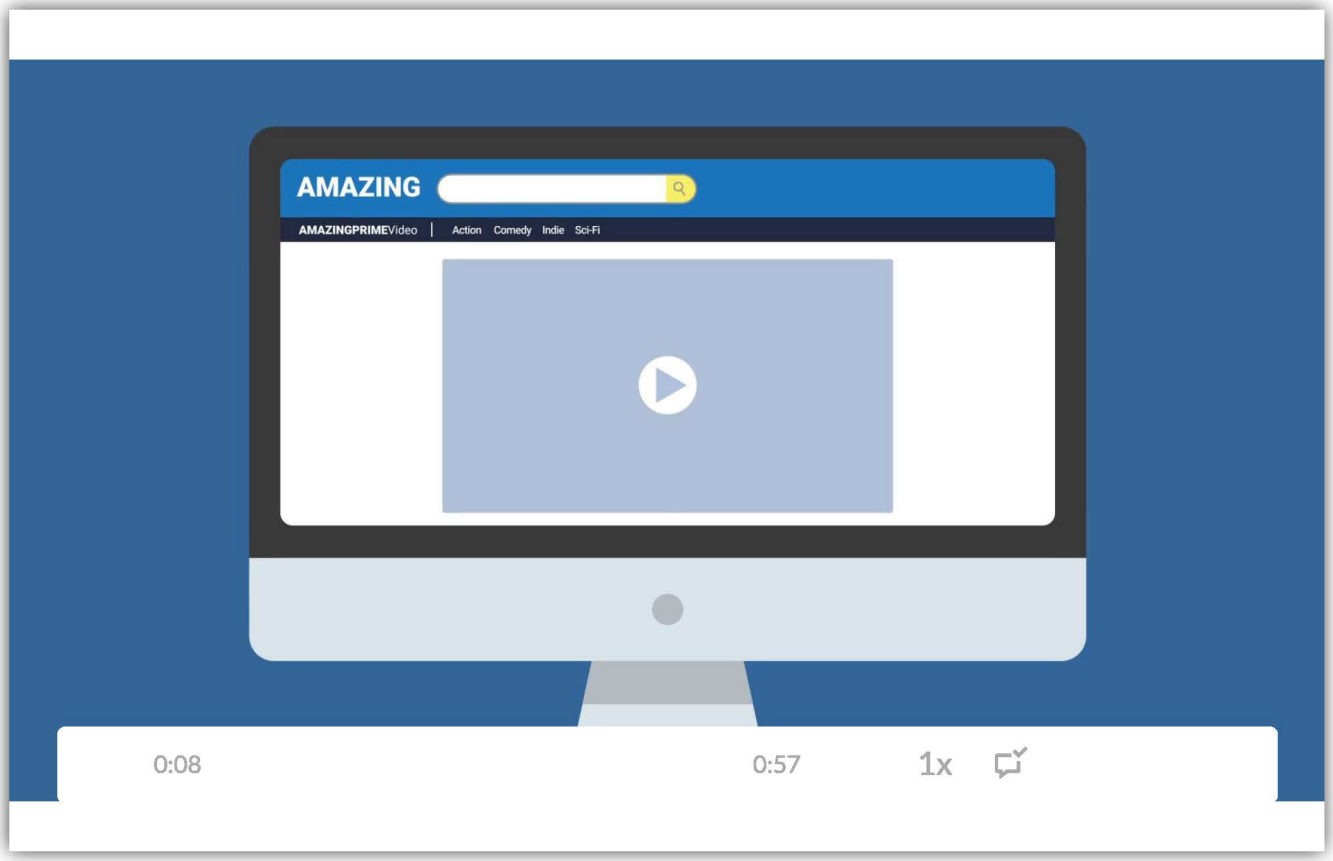
- Use regular expressions to parse data and to transform text into numbers.
- Load data with PostgreSQL.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction (15 minutes)
- Getting Started with ETL (15 minutes)
- Overview of the ETL Process (30 minutes)
- Extract the Data (45 minutes)
- Transform: Clean Individual Datasets (7 hours)
- Transform: Merge Datasets (2 hours)
- Load (1 hour, 30 minutes)
- Application (5 hours)

8.0.3: Lights! Camera! Data!

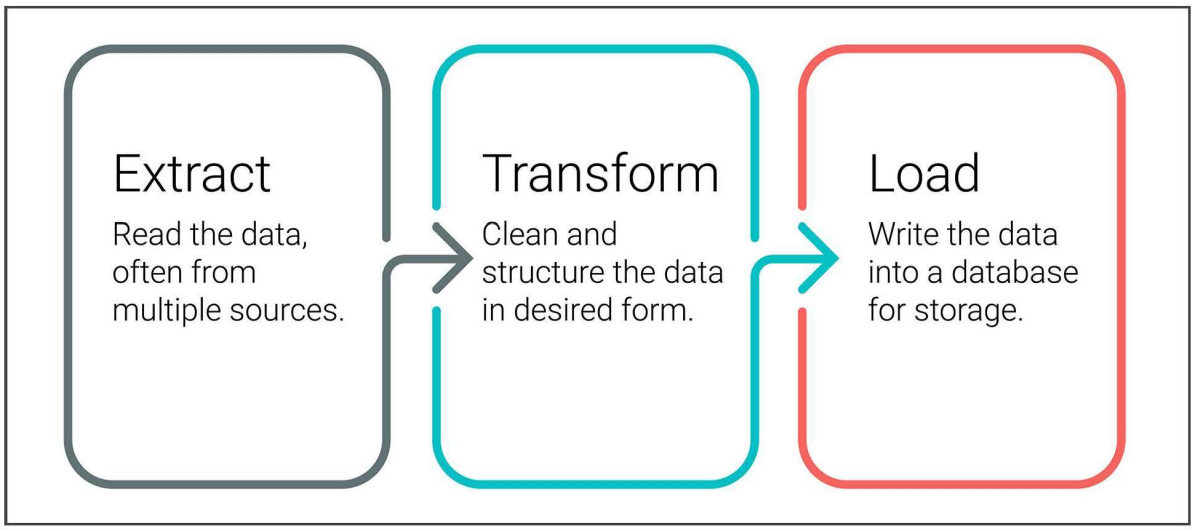


8.1.1: Extract, Transform, Load

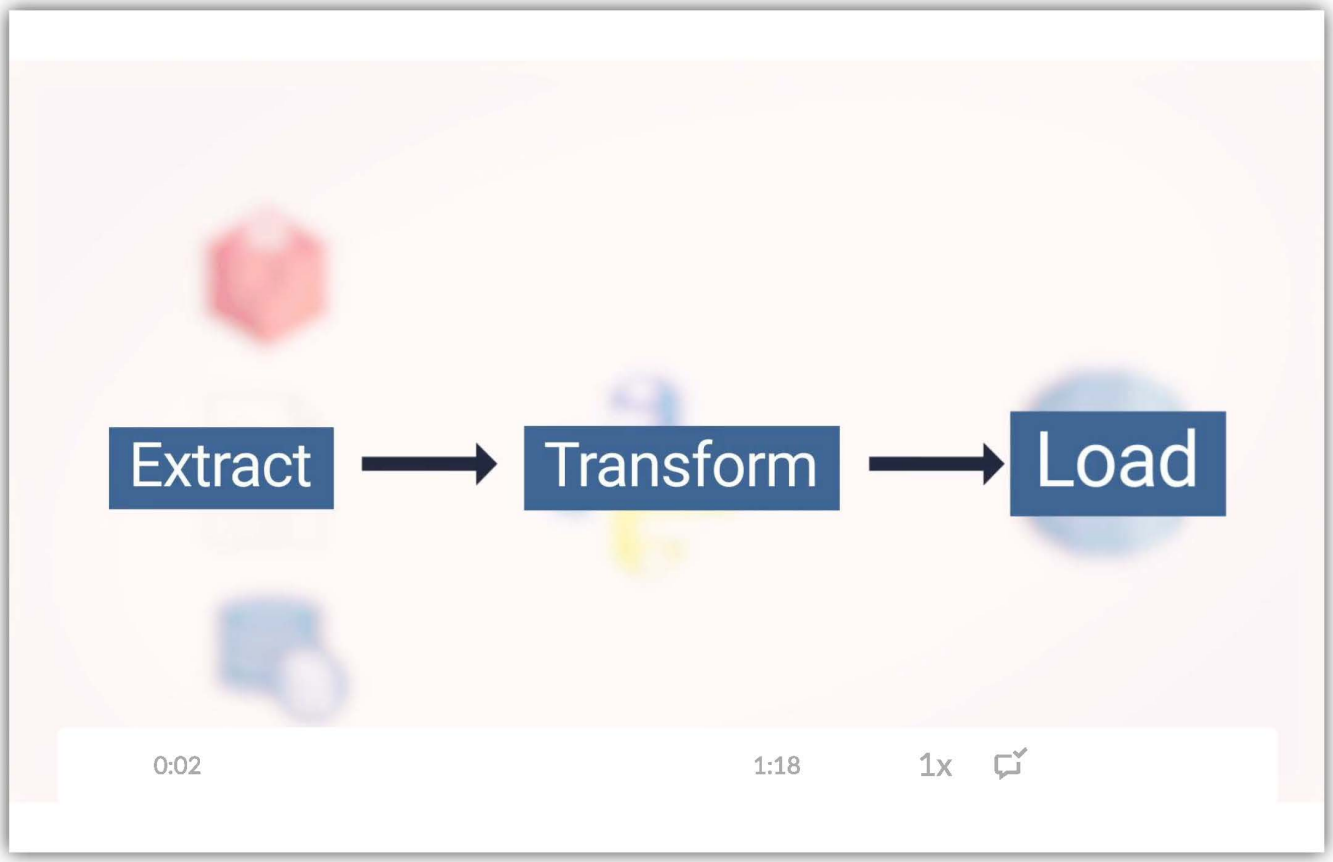
Britta is excited to prepare for the hackathon. In data analysis, a hackathon is an event where teams of analysts collaborate to work intensively on a project, using data to solve a problem. Hackathons generally last several days and teams work around the clock on their projects.

Britta needs to gather data from both Wikipedia and Kaggle, combine them, and save them into a SQL database so that the hackathon participants have a nice, clean dataset to use. To do this, she will follow the ETL process: **extract** the Wikipedia and Kaggle data from their respective files, **transform** the datasets by cleaning them up and joining them together, and **load** the cleaned dataset into a SQL database.

The idea behind ETL is straightforward. Raw data exists in multiple places and needs to be cleaned and structured before it can be analyzed. ETL breaks this problem into three steps, or phases: Extract, Transform, and Load.



Watch the following video to get acquainted with ETL.

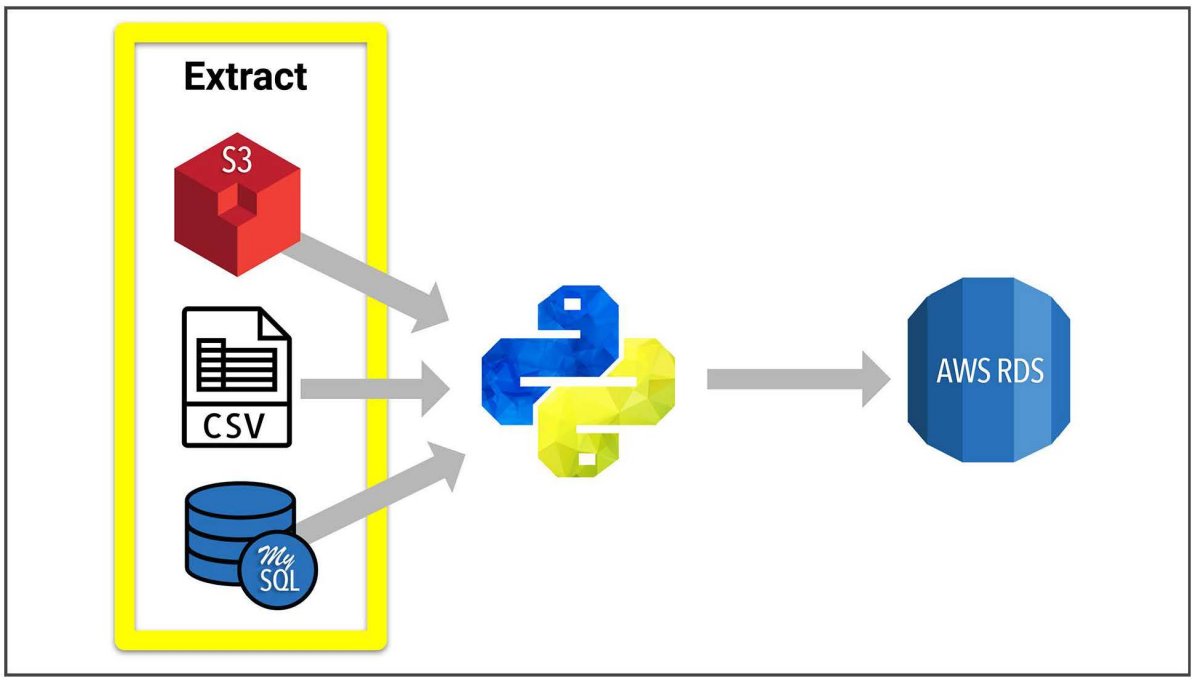


ETL is a flexible process for moving data. It can be as simple as a one-time migration from one database to another, or as complex as an ongoing automated collection of messy, real-time data from many different sources.

In the Extract phase, data is pulled from external or internal sources, possibly disparate. The sources could be flat files, scraped webpages in HTML or JavaScript Object Notation (JSON) format, SQL tables, or even streams of sensor

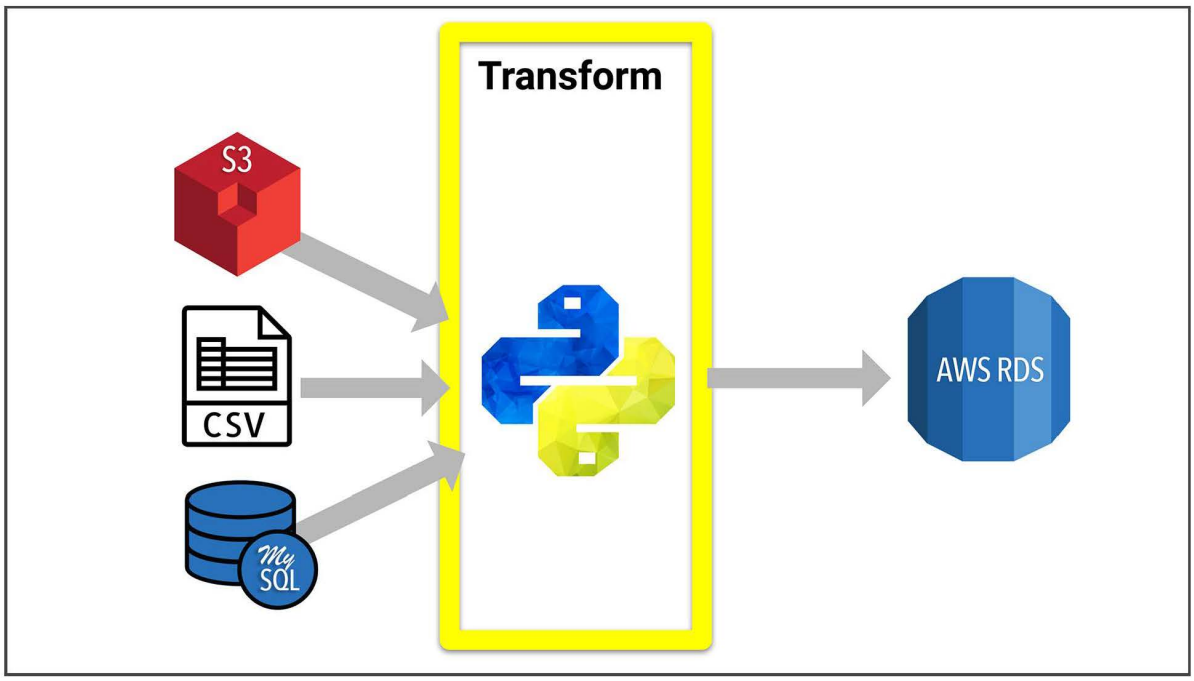
data. The extracted data is held in a staging area in between the data sources and data targets.

For Britta, you'll extract scraped Wikipedia data stored as a JSON, and Kaggle data stored in CSVs.



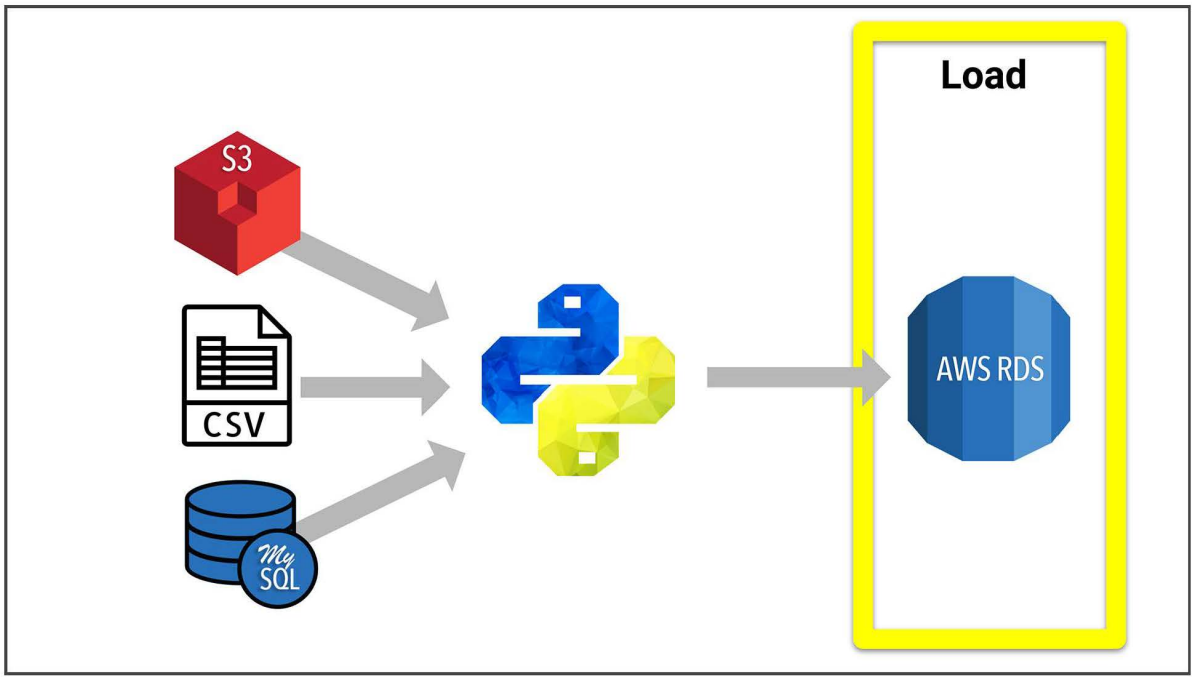
After data is extracted, there are many transformations it may need to go through. The data may need to be filtered, parsed, translated, sorted, interpolated, pivoted, summarized, aggregated, merged, or more. The goal is to create a consistent structure in the data. Without a consistent structure in our data, it's almost impossible to perform any meaningful analysis.

The transformation phase can be accomplished with Python and Pandas, pure SQL, or specialized ETL tools like Apache Airflow or Microsoft SQL Server Integrated Services (SSIS). Python and Pandas are especially good for prototyping an ETL transformation because they provide flexibility and interactivity (especially in a Jupyter Notebook), without enforcing any complicated frameworks. We will use Python and Pandas to explore, document, and perform our data transformation.



Finally, after the data is transformed into a consistent structure, it's loaded into the data target. The data target can be a relational database like PostgreSQL, a non-relational database like MongoDB that stores individual documents, or a data warehouse like Amazon Redshift that optimizes performance specifically for analytics. (We'll look at non-relational databases in more detail later.)

Britta has determined that a SQL database is the best solution for sharing the data in the hackathon, so we'll be loading our data into a PostgreSQL table. SQL databases are often the targets of ETL processes, and because SQL is so ubiquitous, even databases that don't use SQL often have SQL-like interfaces.



Now that you know the phases of the ETL process, let’s get started with the first step and extract some data.

8.2.1: Extract the Wikipedia Movies JSON

Wikipedia has a ton of information about movies, including budgets and box office returns, cast and crew, production and distribution, and so much more. Luckily, one of Britta's coworkers created a script to go through a list of movies on Wikipedia from 1990 to 2018 and extract the data from the sidebar into a JSON. Unfortunately, her coworker can't find the script anymore and just has the JSON file. We'll need to load in the JSON file into a Pandas DataFrame.

> **GitHub**
> Create a new GitHub repository named "Movies-ETL." Then navigate to your class folder and clone your new repo within the folder.

Download the Wikipedia JSON file to your class folder.

wikipedia.movies.json
(https://courses.bootcampspot.com/courses/138/files/14470/download?wrap=1)

Then, activate your coding environment following the steps for your operating system below.

---

macOS

While the JSON file is downloading, follow these steps:

1. Open a new terminal window.
2. Navigate to your class folder.
3. Activate the PythonData environment.
4. Start the Jupyter Notebook server.

> **Rewind**
> Remember, the command to activate the PythonData environment is `conda activate PythonData`. The command to start up the Jupyter Notebook server is `jupyter notebook`.

Windows

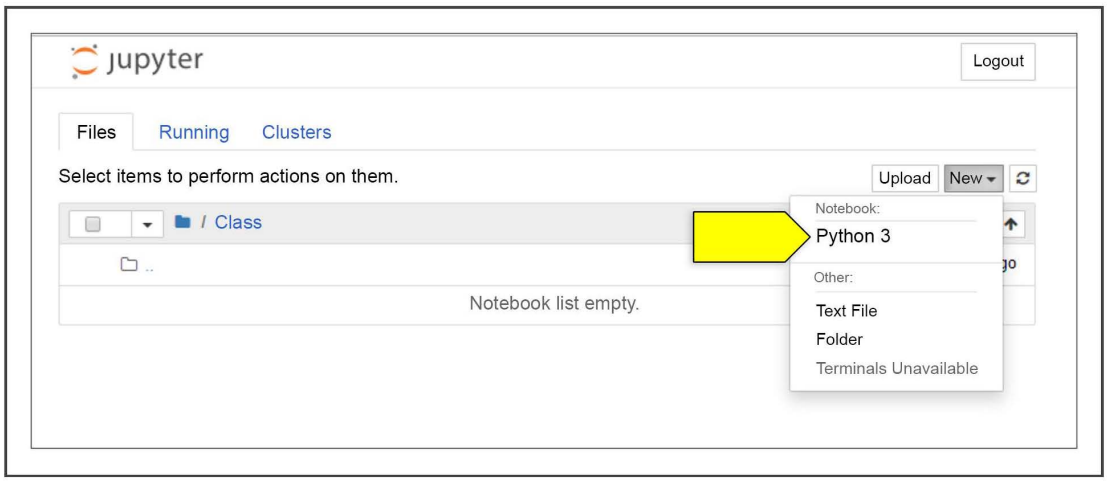
While the JSON file is downloading, follow these steps.

1. Open the Anaconda Prompt (PythonData).
2. Navigate to your class folder.
3. Start the Jupyter Notebook server.

> **Rewind**
> Recall that the command to start the Jupyter Notebook server is `jupyter notebook`.

Once Jupyter Notebook is up and running, create a new Python 3 notebook.

> **Rewind**
> Remember, the New dropdown menu to create a new notebook is in the top right of the Jupyter page.



Eventually, we'll want to create an automated pipeline, which Jupyter Notebooks aren't suited for. But first, we'll need to do some exploratory data analysis, as Jupyter Notebooks are great for exploring data. Then we can copy the code we've created to a Python script.

Find the File

Now that our notebook is up and running and we've downloaded the Wikipedia movie data, we can start writing some code.

> **Rewind**
>
> Remember, if you're going to use any Python dependencies, it's best to import them all at the beginning. As you write your code, if you learn that you need more dependencies, add them to the import statements at the top of your code.

To start, we need to import three dependencies:

1. JSON library to extract the Wikipedia data
2. Pandas library to create DataFrames
3. NumPy library for converting data types

Import all three dependencies in the first cell with the following code:

```
import json
import pandas as pd
import numpy as np
```

In the next cell, we want to import the Wikipedia JSON file. To make our lives easier, define a variable `file_dir` for the directory that's holding our data. The exact file path will depend on the directory in which you've saved your data. Here's what our code looks like:

```
file_dir = 'C:/Users/Username/DataBootcamp/'
```

Now, if you want to open a file in your directory, you can use an f-string (see below) instead of having to type out the whole directory every time. If you move your files, you only need to update the `file_dir` variable.

```
f'{file_dir}filename'
```

You may be tempted to try to read the JSON files directly into a Pandas DataFrame. While technically that may be possible, the `read_json` method that comes built into the Pandas library only works well for data that is already clean—for example, when the JSON data has every field filled in every time it is returned. We call data like this "flat."

Most data you'll work with in real life won't come to you in a flat format. One great thing about the JSON format is it's really flexible, and it can handle raw, messy data. But if you try to read raw, messy JSON data directly into a DataFrame, the DataFrame will be a mess too. It's very difficult to find and fix corrupted data in messy DataFrames, and it's also difficult to consolidate columns without headaches.

As you may have guessed, the type of data we get from doing a scrape of Wikipedia is pretty messy, so it's easier to load the raw JSON as a list of

dictionaries before converting it to a DataFrame.

Load the JSON into a List of Dictionaries

To load the raw JSON into a list of dictionaries, we will use the `load()` method.

> **Rewind**
> Remember, when opening files in Python, we want to use the `with` statement to handle the file resource.

Using the `with` statement, open the Wikipedia JSON file to be read into the variable `file`, and use `json.load()` to save the data to a new variable.

```
with open(f'{file_dir}/wikipedia.movies.json', mode='r') as file:
    wiki_movies_raw = json.load(file)
```

Here, `wiki_movies_raw` is now a list of dicts. Before we take a look at the data, we should check how many records were pulled in. We can use the `len()` function (see below), which returns 7,311 records.

```
len(wiki_movies_raw)
```

> **Pause**
> Is 7,311 a reasonable number of records? We just want to make sure that we don't have an outlandishly large or small number. If we do, there's potentially something seriously wrong with the data that needs to be investigated before moving on.
>
> One way to check that 7,311 is reasonable is to look at the rate of movies

> being released that it implies. Rough mental math here is the key—we want these calculations to be quick enough that these checks become a habit. So, let's say that it's about 7,200 movies over about 30 years. That's 240 movies released per year, or a little less than five movies released per week. That seems a little high if we're considering only major movies, but if for every two major motion pictures, there are three indie films, that doesn't seem like an outlandish number.
>
> Can you think of any other ways to check that this is a reasonable number of movies to have in the dataset? What if you considered how much money all of the movies make every year?

Also, we should always take a look at a few individual records just to make sure that the data didn't come in horribly garbled. With a DataFrame, we'd do this with the `head()` and `tail()` methods, but with a list of dicts, we need to inspect the records directly.

> **Rewind**
> Remember, since we're working with a list, we'll use index slices to select specific chunks of `wiki_movies_raw` to inspect directly. This is also a great use case for **negative index** slices.

To see the first five records, use the following:

```
# First 5 records
wiki_movies_raw[:5]
```

To see the last five records, use the following:

```
# Last 5 records
wiki_movies_raw[-5:]
```

It’s always a good idea to check records in the middle as well. Choose a number somewhere around the halfway mark and look at a handful of records after that index.

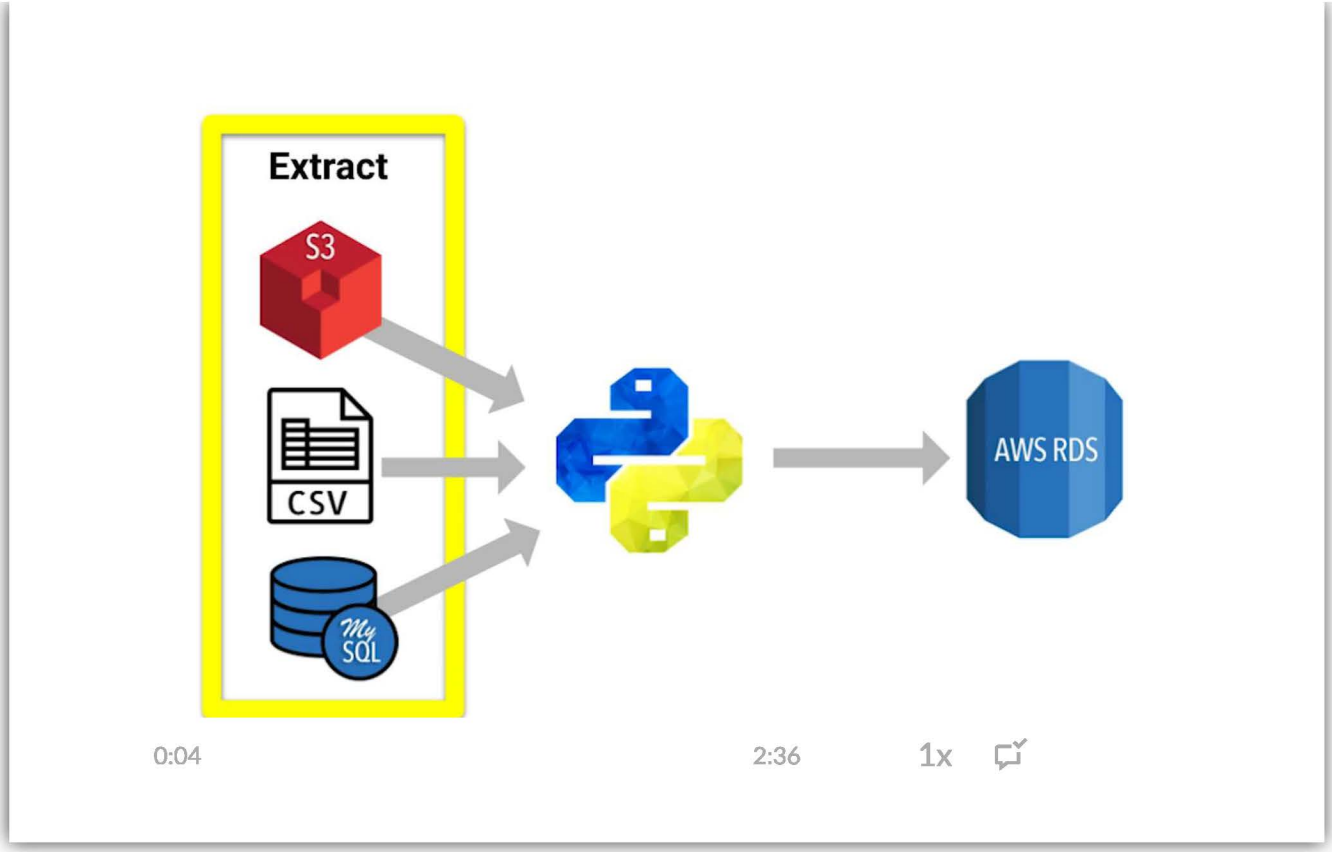
```
# Some records in the middle
wiki_movies_raw[3600:3605]
```

If everything looks good, congratulations! You’re halfway through the Extract step. Now we’ll load in the Kaggle data.

8.2.2: Extract the Kaggle Data

Now that we've loaded the Wikipedia scrape, Britta wants us to include ratings data. However, she knows that her employer, Amazing Prime, won't want to give out their proprietary ratings data to all the hackathon teams. Luckily, she found a dataset on Kaggle that contains ratings data from MovieLens, a site run by the GroupLens research team, which has over 20 million ratings.

Before extracting the Kaggle data, watch the following video to get more acquainted with the Extract step.



MovieLens is a website run by the GroupLens research group at the University of Minnesota. The Kaggle dataset pulls from the MovieLens dataset of over 20 million reviews and contains a metadata file with details about the movies from **The Movie Database (TMDb)** **(https://www.themoviedb.org/)**. Download the **zip file from Kaggle** **(https://www.kaggle.com/rounakbanik/the-movies-dataset/download)**, extract it to your class folder, and decompress the CSV files. We're interested in the `movies_metadata.csv` and `ratings.csv` files.

Since the Kaggle data is already in flat-file formats, we'll just pull them into Pandas DataFrames directly with the following code.

```
kaggle_metadata = pd.read_csv(f'{file_dir}movies_metadata.csv')
ratings = pd.read_csv(f'{file_dir}ratings.csv')
```

Inspect the two DataFrames using the `head()`, `tail()`, and `sample()` methods to make sure that everything seems to be loaded in correctly. (We'll do a deeper dive in the Transform step.)

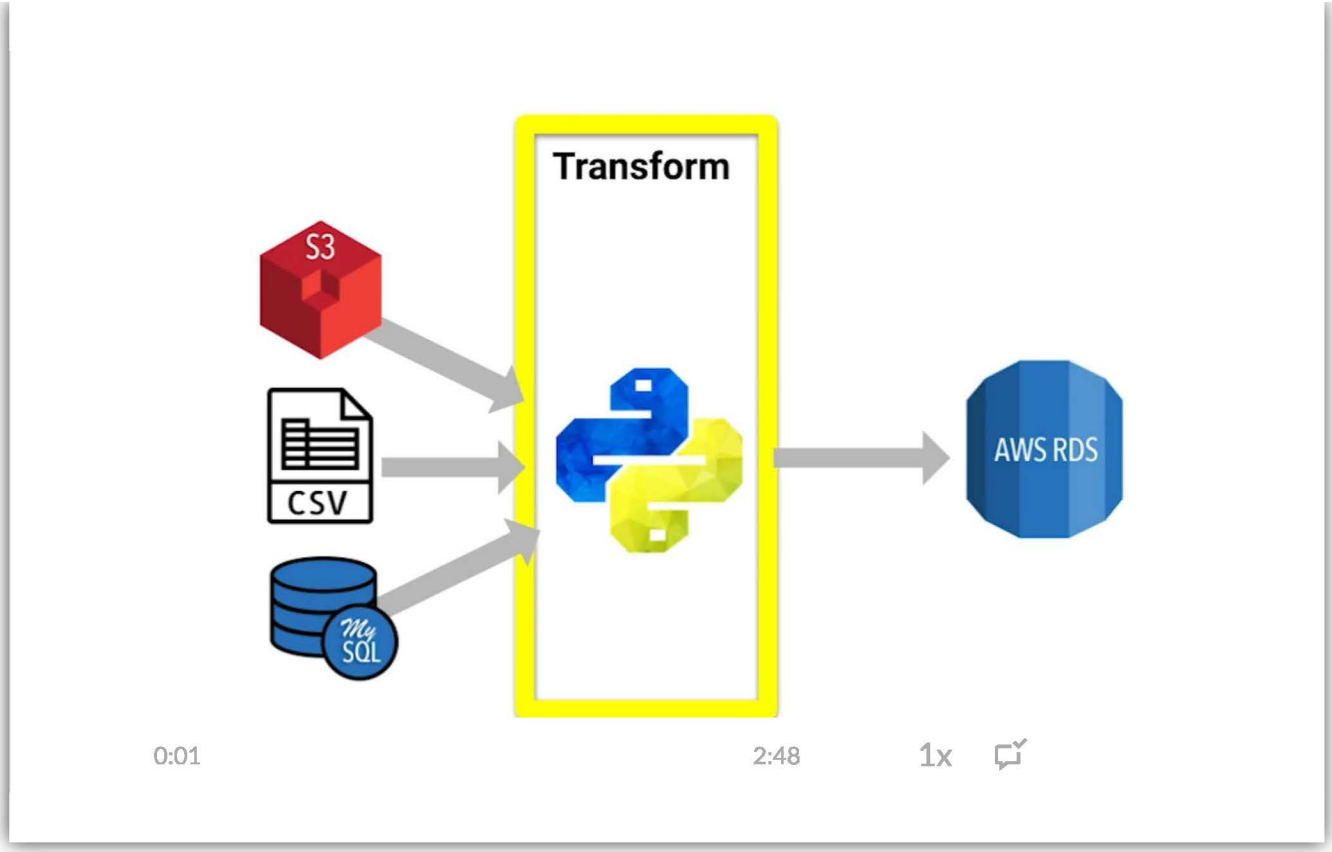
> **Skill Drill**
>
> When creating a new DataFrame, you've probably made a habit of using the `head()` method to get a sense of the data and make sure it's imported correctly, and then using the `tail()` method to ensure the data at the end is imported correctly. However, errors can still occur in the middle of the file, so the best practice is to sample a handful of rows randomly using the `sample()` method. For a DataFrame called `df`, `df.sample(n=5)` will show five random rows from the dataset. We'll cover this in more detail later; for now, just focus on getting a rough sense of the data.

Congratulations! You've just completed the Extract step in ETL. Now we're ready to tackle Transform. And when we get to Load, we're going to use many of the same ideas we just used to extract the data.

8.3.1: Data-Cleaning Strategies

Wikipedia doesn't have strict standards on how movie data is presented, so it needs a lot of work to clean up the data and make it usable. Like most web-scraped data, it's in the flexible JSON format to store all kinds of data, but Britta needs to organize it in a structured format before she can send it to SQL—and she's asked you to assist with this task. (You do have experience in this, after all.) First, explore your options for cleaning the dataset.

The transform step is largely spent on data cleaning. There are other transformations that aren't strictly data cleaning, but for the most part, the transformation step is used to clean up your data.



Every messy dataset presents its own unique challenges. There's no one right way to clean data, but we can still have a rough game plan to follow.

Bad data comes in three states:

- Beyond repair
- Badly damaged
- Wrong form

The state of the data largely determines which strategy you should use to clean it.

Data beyond repair could be data that has been overwritten or has suffered severe data corruption during storage or transfer (such as power loss during writing, voltage spikes, or hard-drive failures). The worst-case example would be having data with every value missing. All the information is lost and unrecoverable. For data beyond repair, all we can do is delete it and move on.

Data that is badly damaged may have good data that we can recover, but it will take time and effort to repair the damaged data. This can be garbled data, with a lot of missing values, from inconsistent sources, or existing in multiple columns.

Consider trade-offs to pick the best solution (even if the "best" solution isn't perfect, but rather the "best-available" solution). To repair badly damaged data, try these strategies:

- Filling in missing data by
  - substituting data from another source,
  - interpolating between existing data points, or
  - extrapolating from existing data
- Standardizing units of measure (e.g., monetary values stored in multiple currencies)
- Consolidating data from multiple columns

Finally, **data in the wrong form** should usually be fixed—that is, the data is good but can't be used in its current form. "Good" data in the wrong form can be data that is too granular or detailed, numeric data stored as strings, or data that needs to be split into multiple columns (e.g., address data). To remedy good data in the wrong form, try these strategies:

- Reshape the data
- Convert data types
- Parse text data to the correct format
- Split columns

These options are all available to us, but knowing when to perform which strategy can feel overwhelming. There is no simple checklist or flowchart we can use to guide us, and ultimately, that's a good thing. In data cleaning, we have to constantly ask ourselves what we might have missed, and following a rigid plan means we won't be asking ourselves those important questions. Data cleaning requires a lot of improvising.

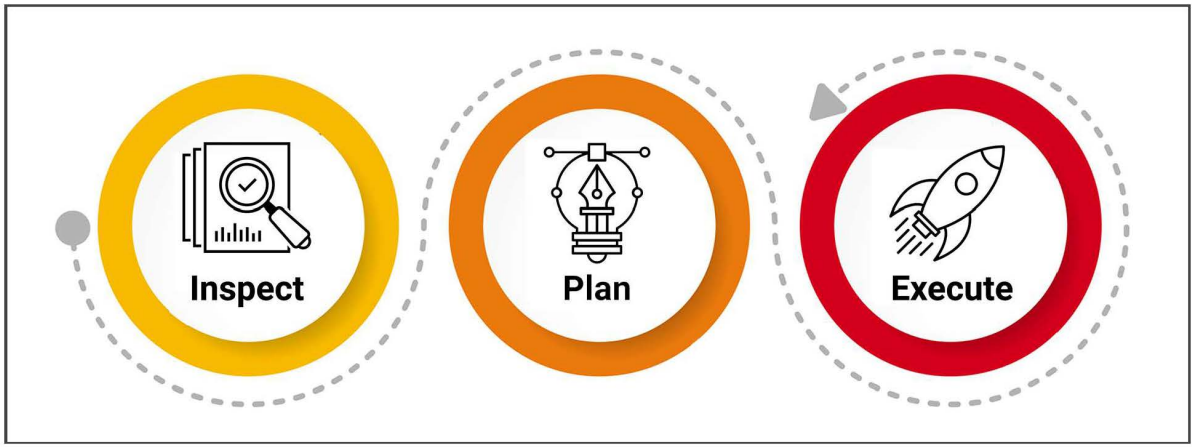
> **Important**
>
> **It's important to document your data cleaning assumptions as well as decisions and their motivations.** Later decisions depend on earlier decisions made, which can be too much to remember. Any assumptions

> that were part of an earlier decision can, if forgotten, ruin later steps.
>
> Transforming a messy dataset into a clean dataset is an iterative process. As you clean one part of the data, you may reveal something messy in another part of the data. Sometimes that means unwinding a lot of work that you've already done and having to redo it with a slight change. Documenting *why* a particular step is necessary will show you how to redo it without introducing more errors.

We're not completely lost—we do have a strategy. We're not going to try and clean all the data at once. Instead, we're going to focus on one problem at a time using an iterative process.

8.3.2: Iterative Process for Cleaning Data

Britta is confident in your data-cleaning strategy. You're going to follow an iterative process based on three key steps: plan, inspect, execute.



The iterative process for cleaning data can be broken down as follows:

- First, we need to **inspect** our data and identify a problem.
- Once we've identified the problem, we need to make a **plan** and decide whether it is worth the time and effort to fix it.
- Finally, we **execute** the repair.

Early iterations focus on making the data easier to investigate: deleting obviously bad data, removing superfluous columns (e.g., columns with only one value or missing an overwhelming amount of data), removing duplicate rows, consolidating columns, and reshaping the data if necessary.

As the data becomes easier to investigate, iterations focus on fixing the most obvious problems first. As obvious problems are resolved, more subtle problems

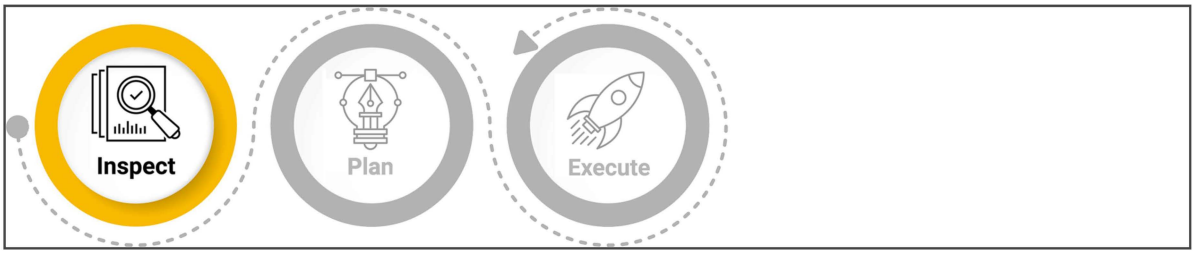
become noticeable.

As the iterations shift toward solving more subtle problems, we might discover an earlier step needs to change as well as all the iterations that follow that step. It's frustrating when work has to be undone, but at least you now have a better understanding of your data.

> **Note**
> In general, earlier iterations try to handle big chunks of data at one time, such as removing columns and rows, while later iterations focus on smaller chunks of data, such as parsing values.

It's rare to reach a point where no more problems exist in the data. More likely, a point is reached where the work to fix any remaining problems isn't worth the amount of data that would be recovered. After the remaining issues are documented, the transform step is considered finished.

Now that we know how to use our iterative process, let's review each step in detail.



Before we can do anything, we have to look at our data. The first thing we want to know is whether or not the data was imported correctly. The simplest way to confirm this is to print out the first few data points and examine the first few rows for irregularities, e.g., data in the wrong columns, all missing values, column headers that don't make sense, or garbled characters.

If the data doesn't look correct, we know it wasn't imported correctly. Sometimes the beginning of the data looks fine, but if the import went wrong somewhere in

the middle of the process, the rest of the data can be affected.

Therefore, it's good practice to check the last few rows and a random sample of rows. We can also start to answer some simple questions about the data:

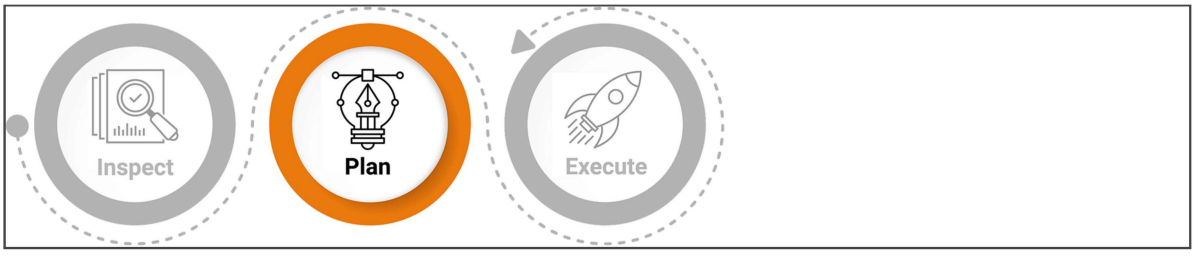
- Does it have a consistent structure (like a CSV table) or is it unstructured (like a collection of email messages)?
- How is each data point identified—is there an explicit, unique ID for each data point, or will one need to be built?

However, most usable data contains too many data points to review every single one, so we'll need to use strategies that tell us about the whole dataset.

First, count how many data points or rows exist. If the data is structured, count the number of columns and missing values in each column. If possible, count the number of unique values in each column and how frequently each unique value appears. To determine if this is possible, we'll need to investigate the data types for each column.

When investigating the data type for a column, we want to know what the data type is and what the data type should be. For example, if we see "True" and "False" as entries for a column, we expect that the data type will be a Boolean. If the data type is a string, we need to investigate further.

If a column's data type is numeric, we can summarize its data with some basic statistics, such as measures of central tendency (e.g., mean and/or median) and measures of spread (e.g., standard deviation, interquartile range, minimum/maximum). We can also investigate columns with statistical plots, like scatter plots and histograms.



After we've investigated our data and started to identify problem areas, we can make decisions about how to fix the problems. This requires articulating the problems clearly—even if that is simply expressing the problems to ourselves—and devising a plan to modify the data and fix the problem. In this step, we'll answer several questions, including:

- If a column doesn't have the right data type, is it a problem with the whole column? Or are just a handful of rows causing the issues?
- Do rows have outliers due to spurious data? Or are they valid data points?
- When values are missing, will they need to be removed, replaced, or interpolated?

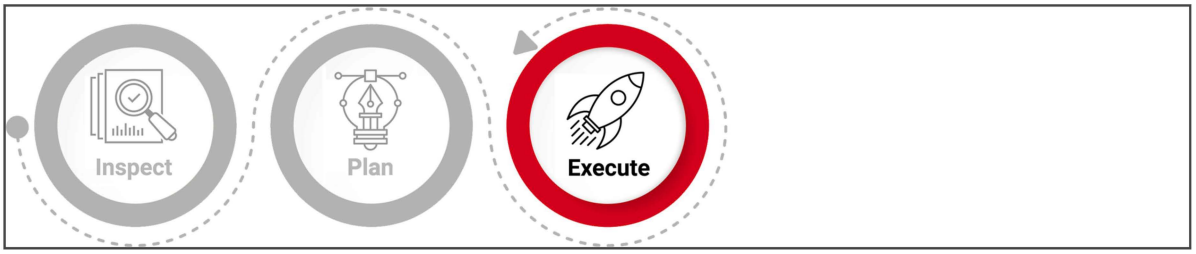
The answers to these questions will tell us how we need to modify our data. Keep in mind, there are two main ways: we can modify values and we can modify structure.

Modifying data values includes removing rows or columns, replacing values, or generating new columns from old ones. We might remove rows with missing or corrupted data, columns with only one value, or columns mostly missing data. There are many ways we might replace data. Instead of dropping missing values, we might replace them with zeros or empty strings. We might have a column that contains nonstandard values, such as percentages that are stored as whole numbers from 0 to 100 and also as fractions from 0 to 1, and we would replace them with one standard form.

Converting a column to a new data type is also a form of replacing values. We can also bin data (like rounding to the nearest hundred), replacing numeric data (e.g., income) with categorical data (e.g., income brackets). We might generate new columns by splitting an existing column into several new columns—by splitting an address column to street, city, state, and zip code columns, for example—or by calculating a new column from multiple existing columns, like calculating total price by multiplying item prices by quantities.

Modifying data structure includes pivoting the values of one column into multiple columns, aggregating rows, and merging multiple data sets. It can also include aggregating large amounts of data into summary data or summary statistics.

With clearly stated steps to fix the problem, we can make an informed decision about whether implementing the plan is worth the effort. Sometimes there are multiple viable resolutions to choose from. To decide, we weigh trade-offs and ultimately choose the best option.



Once we have a detailed list of steps to modify our dataset, it's time to implement it. We'll start writing code to fix the problem we're focusing on.

As we write, we might discover that the problem is more difficult than initially expected. This is a normal part of the process. As you implement your changes, try to take into account any unintended consequences you could introduce.

After implementing your changes, the next step is to return and **inspect** the data in a new iteration. This step is important, especially when modifying data structure, which can introduce missing data points, or inadvertently create more bad data.

Cleanup Is Messy Work

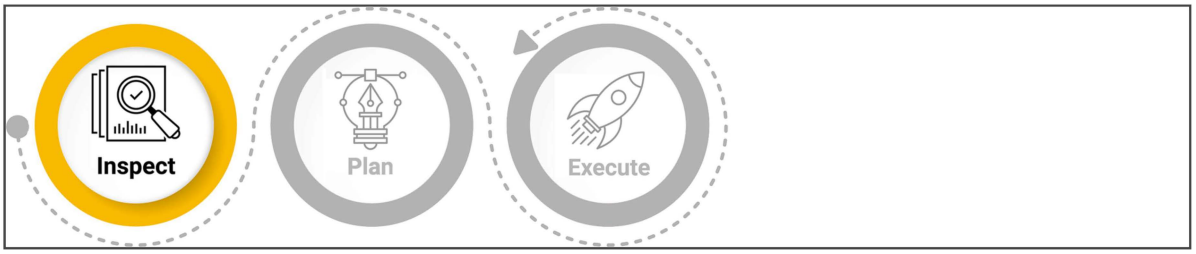
While transforming your data, you might bounce between steps in the iteration—for example, making a plan, then realizing you need to inspect more; executing a plan, then realizing a step was missed and you need to quickly rework the plan. We offer these steps as a descriptive, not prescriptive, approach. Cleaning up messy data is a messy process. The best practice is to document every step of your thought process and actions in detail.

Now let's go get our hands dirty with some messy data.

8.3.3: Investigate the Wikipedia Data

Our Wikipedia data is especially messy. As much as editors try to be consistent, each page can be edited by a different person. Besides, because the movie data comes from the sidebar, different movies can have different columns. However, after cleaning the data, the result will be a nice, organized table of data, where every row is a single movie. You'll start by investigating the data for errors.

Initial Investigation



One of the easiest ways to find glaring errors is to just pretend as if there aren't any, and try to jump straight to the finish line. Eventually, we want to clean up the Wikipedia data into tabular data with rows and columns, so let's see what happens if we create a DataFrame from our raw data.

What code should we use to turn `wiki_movies_raw` into a DataFrame?

- `wiki_movies_df = pd.DataFrame(wiki_movies_raw)`
- `wiki_movies_df = pd.read_json(wiki_movies_df)`
- `wiki_movies_df = wiki_movies_raw.to_frame()`
- `wiki_movies_df = pd.Series(wiki_movies_raw).to_frame()`

Check Answer

Finish ▸

Run `wiki_movies_df = pd.DataFrame(wiki_movies_raw)`. Then use the `head()` function to take a quick peek at the DataFrame.

```
wiki_movies_df.head()
```

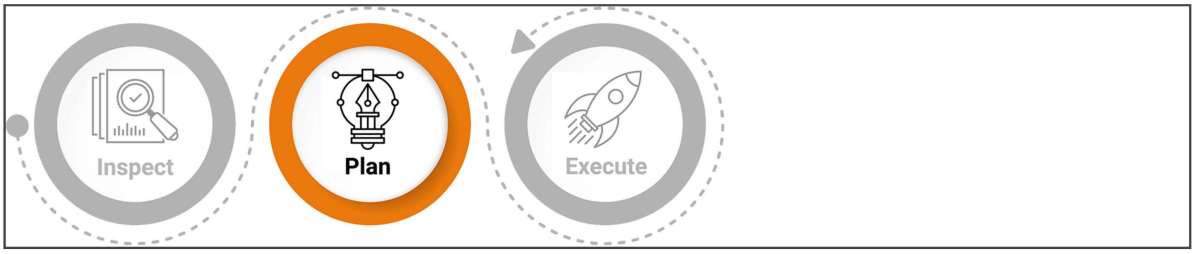
We usually use `head()` to inspect a few rows of data, but it also tells us about the shape of our DataFrame. Below the rows of data, you'll see that it says there are 5 rows of data and 193 columns. That's a lot of columns! Even if we try to use `print(wiki_movies_df.columns)`, they won't all print out. We'll have to convert `wiki_movies_df.columns` to a list to see all of the columns.

Use `wiki_movies_df.columns.tolist()` and run the cell to see all of the column names that were imported. Your output should appear as follows:

```
['url',
 'year',
 'imdb_link',
 'title',
 'Directed by',
 'Produced by',
 'Screenplay by',
 'Story by',
```

```
'Based on',
'Starring',
'Narrated by',
'Music by',
'Cinematography',
'Edited by',
'Productioncompany ',
'Distributed by',
'Release date',
'Running time',
'Country',
'Language',
```

We can identify column names that don't relate to movie data, such as "Dewey Decimal," "Headquarters," and "Number of employees." (There may be other examples that jumped out at you as well.)



Let's modify our JSON data by restricting it to only those entries that have a director and an IMDb link. We can do this with a list comprehension.

Use List Comprehensions to Filter Data

> **Rewind**
> We've used list comprehensions previously as a compact way to apply a function to every element in a list. In this module, we'll use list comprehensions to filter data.

So far, we've used list comprehensions in the form to compress code that would have been done in a `for` loop.

```
[expression for element in source_list]
```

We can also filter out results using a conditional filter expression, as shown below:

```
[expression for element in source_list if filter_expression]
```

The resulting list will only have elements where the filter expression evaluates to True. For example, if we had a list of dogs, and we only wanted a list of those weighing more than 30 pounds, the list comprehension would be:

```
[dog for dog in dogs if dog['weight'] > 30]
```

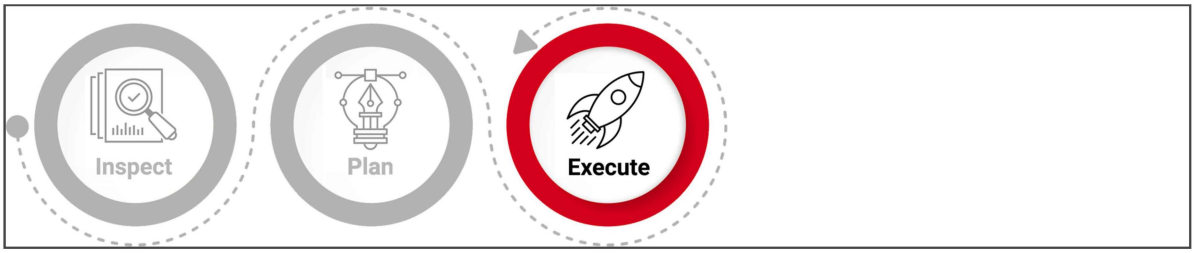
What would our results be if we used the list comprehension below?

```
[dog for dog in dogs if dog['is_good'] == True]
```

We would get back the original list of dogs as all dogs are good dogs.

To create a filter expression for only movies with a director and an IMDb link, keep in mind that there are two columns in the data for director information. We'll need to check if either "Director" or "Directed by" are keys in the current dict. If there is a director listed, we also want to check that the dict has an IMDb link. Luckily, that information is only in one column, `imdb_link`, so our filter expression will look like the following:

```
if ('Director' in movie or 'Directed by' in movie) and 'imdb_link' in mov
```

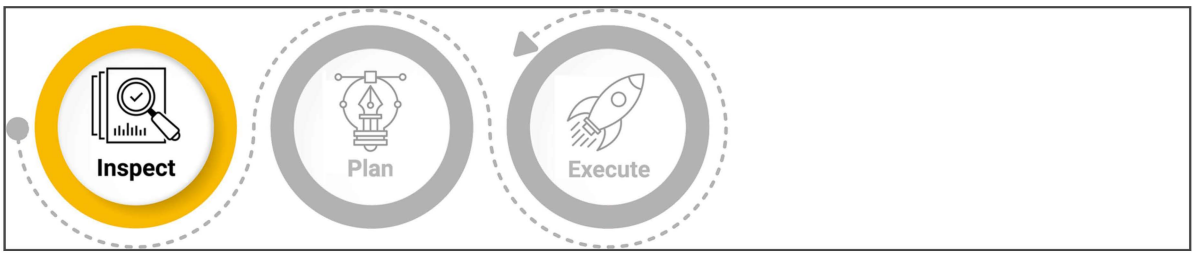


Create a list comprehension with the filter expression we created and save that to an intermediate variable `wiki_movies`. See how many movies are in `wiki_movies` with the `len()` function.

```
wiki_movies = [movie for movie in wiki_movies_raw
               if ('Director' in movie or 'Directed by' in movie)
                   and 'imdb_link' in movie]
len(wiki_movies)
```

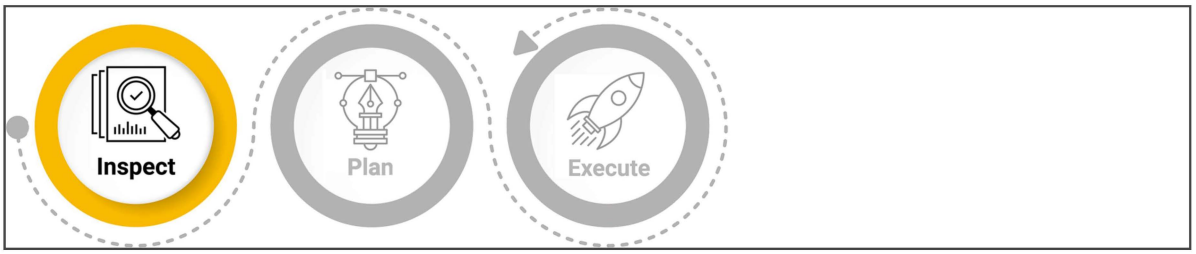
This only cuts the number of movies down to 7,080. Not too bad. Make a DataFrame from `wiki_movies`, and there should only be 78 columns.

> **Note**
> This is why it's easier to load the JSON in first and then convert it to a DataFrame. Instead of trying to identify which columns in our DataFrame don't belong, we just remove the bad data points, and the bad columns never get imported in.

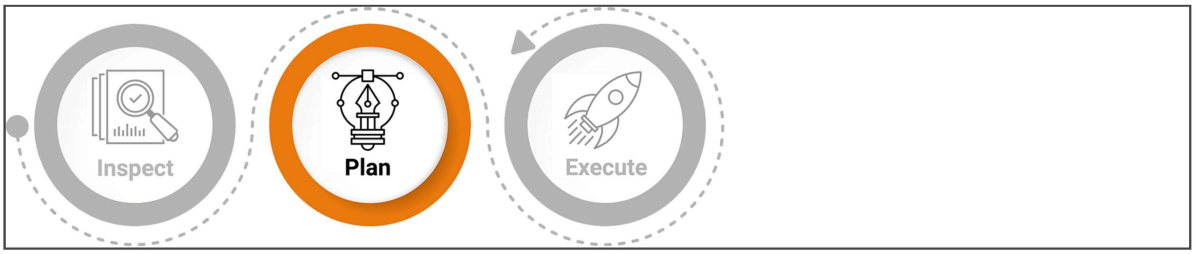


78 columns are still a lot of columns, so let's keep investigating.

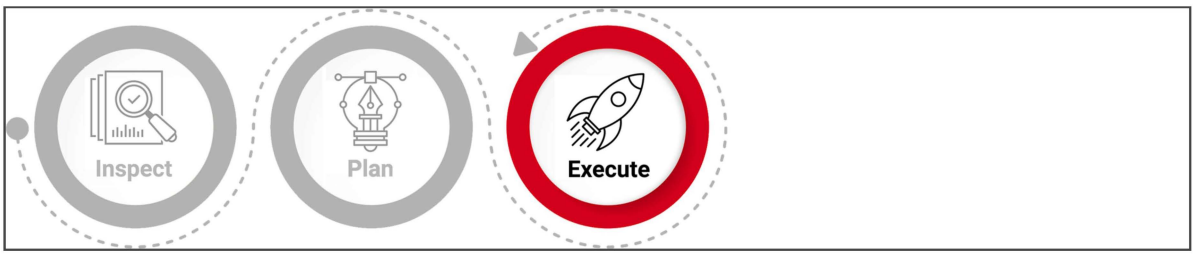
> **Important**
>
> One thing to watch out for is to make **nondestructive** edits as much as possible while designing your pipeline. That means it's better to keep your raw data in one variable, and put the cleaned data in another variable. It takes up more memory, but it makes tracking the iterative process of data cleaning easier.
>
> For example, if we had just deleted the movies from `wiki_movies_raw` that didn't have "Directed by" as a key, we'd have made a destructive edit and missed that some have "Director" as the key instead. This can cause errors to creep in until we realize our mistake, and if we made destructive edits, it would be impossible to see what caused those errors. Using nondestructive edits helps determine the origin of errors.



There sure are a lot of languages—we'll get to those shortly. For now, one of the columns that stands out is "No. of episodes."



It looks like we've got some TV shows in our data instead of movies. We'll want to get rid of those, too.



We'll add that filter to our list comprehension.

```
wiki_movies = [movie for movie in wiki_movies_raw
               if ('Director' in movie or 'Directed by' in movie)
                   and 'imdb_link' in movie
                   and 'No. of episodes' not in movie]
```

IMPORTANT

Don't worry if you didn't catch the "No. of episodes" column in the list. Cleaning data is an iterative process, and if you started with cleaning up the language data first, or some other part of the data, you would see the "No. of episodes" column soon enough. The key is to keep reworking the pipeline bit by bit.

8.3.4: Revisit Functions

Now that you've filtered out bad data, you need to clean up each movie entry so it's in a standard format. If you can make one process broad enough to handle every movie entry, you can apply that process repeatedly for every movie entry. For this task you will create a function.

To keep things organized, we are going to make a **function** that performs that cleaning process.

> **Rewind**
> Remember, functions are blocks of code within a script or algorithm that perform a specific task. There are four basic parts to a function:
>
> 1. Name
> 2. Parameters
> 3. Code block
> 4. Return value

We're going to expand on some trickier aspects of functions before we make our movie cleaning function.

First, we need to talk about **scope**. Inside the code block of a function, we can use variables that were created outside the function and initialize new variables inside

the function.

This is called the “scope” of the variables:

- Variables created outside the function are called **global** variables.
- New variables created inside the function are **local** variables.
- The hierarchy of variables is called the **scope**.

> **Important**
> The scope of local variables only works inside the function in which they are created. If we try to access a local variable outside the function in which it’s defined, Python will raise a `NameError` because it won’t be able to find the function.

Keep this trick in mind with the scope of variables: We can create a local variable with the same name as a global variable. Inside the function, the local variable takes precedence, but the value of the global variable will be unchanged. For example, consider the code below:

```
x = 'global value'

def foo():
    x = 'local value'
    print(x)

foo()
print(x)
```

The output of this code would be:

```
local value
global value
```

Outside the function, `x` has the value `global value`. Although we define a new function after assigning a value to the global variable `x`, the `x` inside the function

is a new variable whose scope is local to `foo()`. Even after calling the `foo()` function, the global variable `x` keeps its original value.

However, we have to be careful with variables we send to a function. Any data structure can be sent as a parameter to a function, including dicts, lists, and tuples, but we must be careful with mutable objects.

> **Caution**
> When passing mutable objects like a dict or list as parameters to a function, the function can change the values inside the object.

For example, consider the code below:

```
my_list = [1,2,3]
def append_four(x):
    x.append(4)
append_four(my_list)
print(my_list)
```

The output would be:

```
[1, 2, 3, 4]
```

It changed `my_list` to `[1,2,3,4]`, even outside the function. To leave the original object unchanged, we need to make a copy. For lists, we make a copy with the `list` function; for dicts, we make a copy with the `dict` function, like so:

```
new_list = list(old_list)
new_dict = dict(old_dict)
```

Lambda Functions

There's a special function we can make in Python called **lambda**, which is the most stripped-down kind we can make.

Lambda functions are written in one line and automatically return a value without using the return keyword. Lambda functions have no name and are also known as "anonymous functions."

So, how do we call a function with no name? We don't. There are functions that expect other functions to be sent to them as a parameter, and lambda functions are a way to quickly create a concise function to send as a parameter to another function. We'll return to lambda functions in a later section, but the basic syntax to a lambda function follows:

```
lambda arguments: expression
```

This function will take in an argument and will return the expression. Even though lambda functions are supposed to be anonymous, just this once we'll create a lambda function and assign a name so that we can see how they work. A lambda function that squares a value looks like the following:

```
lambda x: x * x
```

Here, `x` is the argument, and `x * x` is the expression. Let's assign this to a name so that we can use it:

```
square = lambda x: x * x
square(5)
```

The output will be:

```
25
```

If this seems strange, don't worry. The main benefit of lambda functions is that they can be used as one-time-use functions. We'll talk more about lambda functions when we have a more natural use case for them—they'll make more sense at that time.

> **Note**
> There are many esoteric topics related to functions, including defining functions within functions, recursion, and functions that create more functions. These are part of the **functional programming** paradigm. Functional programming has its place, but it also has a well-deserved reputation for being confusing and a bit niche. It's good to be aware of functional programming, but we won't rely on its concepts for this module.

Functions are powerful, but enough talk—let's put them to work.

8.3.5: Create a Function to Clean the Data, Part 1

Filtering out bad data isn't enough. You know that you need to make sure the *good* data that you have is clean enough to use. There's a lot at stake!

Now we're ready to create our function to clean our movie data.

First, write a simple function to make a copy of the movie and return it. As we work with our data, we'll iteratively add more to our code block. To start, call the function `clean_movie`, and have it take `movie` as a parameter.

```
def clean_movie(movie):
```

Because the movies are dicts and we want to make nondestructive edits, make a copy of the incoming movie.

To make a copy of `movie`, we'll use the `dict()` constructor.

> **Important**
> **Constructors** are special functions that initialize new objects. They reserve space in memory for the object and perform any initializations the object requires. Also, constructors can take parameters and initialize a new object using those parameters.

When we pass `movie` as a parameter to the `dict()` constructor, it reserves a new space in memory and copies all of the info in `movie` to that new space.

As an example, we could start our function off with this code:

```
def clean_movie(movie):
    movie_copy = dict(movie)
```

However, we have another trick that's even better.

Inside of the function, we can create a new local variable called `movie` and assign it the new copy of the parameter `movie`.

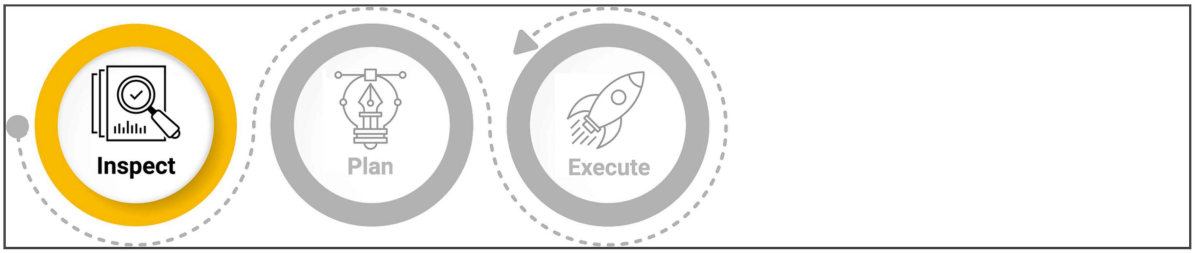
```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
```

This way, inside of the `clean_movie()` function, `movie` will refer to the local copy. Any changes we make inside `clean_movie()` will now only affect the copy, so if we make a mistake, we still have the original, untouched `movie` to reference.

To finish our skeleton of the `clean_movie` function, return the `movie` variable.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    return movie
```

This function doesn't do much right now, but we'll be adding more to it soon.



Now take a look at what's going on with those languages. The first one on the list is Arabic, so let's see which movies have a value for "Arabic."

```
wiki_movies_df[wiki_movies_df['Arabic'].notnull()]
```

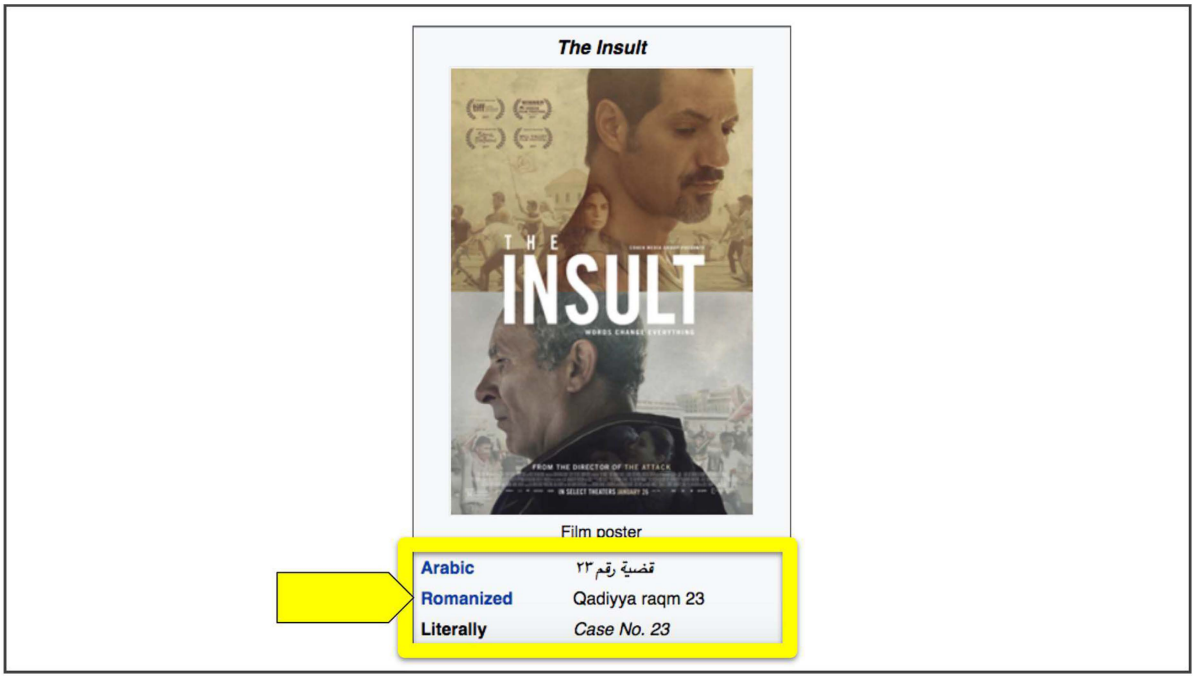
	url	year	imbd_link	title	Directed by	Produced by	Screenplay by	Story by
6834	https://en.wikipedia.org/wiki/The_Insult_(film)	2018	https://www.imbd.com/title/tt7048622/	The Insult	Ziad Doueiri	Rachid **Bouchareb,** Jean Bréhat, Julie Gayet, Antoun Sehnaoui	Nadia Turincev	Nadia Turincev
7058	https://en.wikipedia.org/wiki/Capernaum_(film)	2018	https://www.imbd.com/title/tt8267604/	Capernaum	Nadine Labaki	Michel Merkt, Khaled Mouzanar	Nadia Turincev, Jihad Hojily	Nadia Turincev

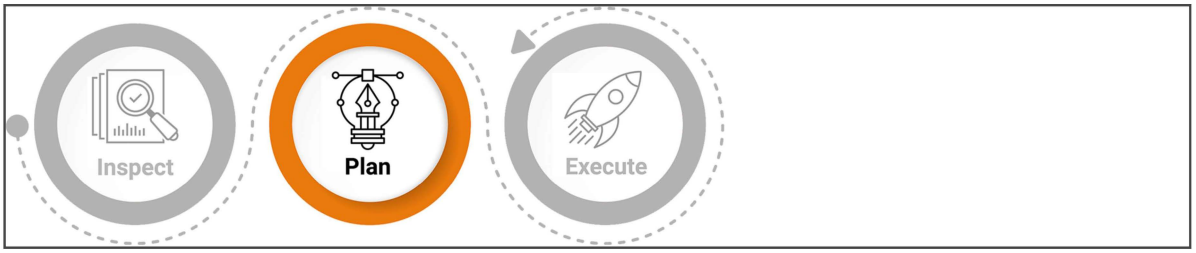
2 rows x 75 columns

The results return two movies, the first listed is *The Insult*. Visit the movie's **Wikipedia page (https://en.wikipedia.org/wiki/The_Insult_(film)%09)** for more details.

```
wiki_movies_df[wiki_movies_df['Arabic'].notnull()]['url']
```

```
6834    https://en.wikipedia.org/wiki/The_Insult_(film)
7058     https://en.wikipedia.org/wiki/Capernaum_(film)
Name: url, dtype: object
```





The different language columns are for alternate titles of the movie. Let's combine all of them into one dictionary that has all the alternate titles.

To do that, we need to go through each of the columns, one by one, and determine which are alternate titles. Some might be tricky. If you're not sure what a column name means, google it. Also, review a column's data to understand the type of content in that column.

For example, perhaps you've never heard of "McCune–Reischauer." Is it an esoteric filmmaking technique? Google it, and you'll learn it's a romanization system for Korean. Now look at the actual values contained in the column. If the values don't make sense to you either, google them, too.

> **Hint**
>
> The `value_counts()` method is a quick, easy way to see what non-null values there are in a column.

Try the following Skill Drill. If you're not sure, don't guess. Look at the data, and investigate the source if you have any questions. There are no shortcuts in this task.

> **Skill Drill**
> Go through each of the columns, one by one, and determine which columns hold alternate titles.
>
> **Hint:** You might find it easier to sort the column names first as you're going through them. The following will display columns in alphabetical order.
>
> ```
> sorted(wiki_movies_df.columns.tolist())
> ```

Question 1

Which columns hold alternate title data? (Select all that apply)

- [ ] Adaptation by
- [x] Also known as ✔
- [ ] Animation by
- [x] Arabic ✔
- [ ] Audio format
- [ ] Based on
- [ ] Box office
- [ ] Budget
- [x] Cantonese ✔
- [x] Chinese ✔
- [ ] Cinematography
- [ ] Color process
- [ ] Composer(s)
- [ ] Country
- [ ] Country of origin
- [ ] Created by
- [ ] Directed by
- [ ] Director
- [ ] Distributed by
- [ ] Distributor
- [ ] Edited by
- [ ] Editor(s)
- [ ] Executive producer(s)
- [ ] Followed by

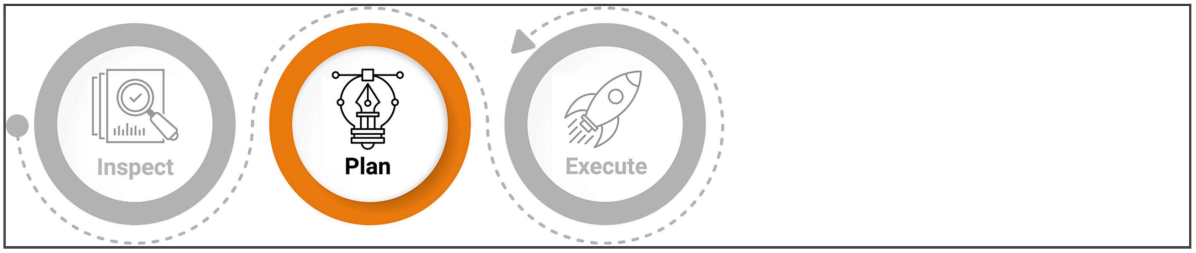
- [x] French ✔
- [ ] Genre
- [x] Hangul ✔
- [x] Hebrew ✔
- [x] Hepburn ✔
- [x] Japanese ✔
- [ ] Label
- [ ] Language
- [ ] Length
- [x] Literally ✔
- [x] Mandarin ✔
- [x] McCune–Reischauer ✔
- [ ] Music by
- [ ] Narrated by
- [ ] Original language(s)
- [ ] Original network
- [ ] Original release
- [x] Original title ✔
- [ ] Picture format
- [x] Polish ✔
- [ ] Preceded by
- [ ] Produced by
- [ ] Producer
- [ ] Producer(s)
- [ ] Production company(s)
- [ ] Production location(s)

- [ ] Productioncompanies
- [ ] Productioncompany
- [ ] Recorded
- [ ] Release date
- [ ] Released
- [x] Revised Romanization ✔
- [x] Romanized ✔
- [ ] Running time
- [x] Russian ✔
- [ ] Screen story by
- [ ] Screenplay by
- [x] Simplified ✔
- [ ] Starring
- [ ] Story by
- [ ] Suggested by
- [ ] Theme music composer
- [x] Traditional ✔
- [ ] Venue
- [ ] Voices of
- [ ] Written by
- [x] Yiddish ✔
- [ ] imdb_link
- [ ] title
- [ ] url
- [ ] year

Correct feedback:
Correct.

Retake

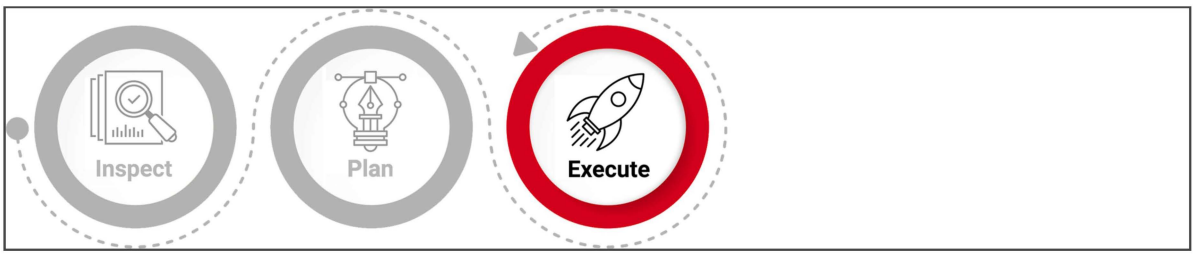
Handle the Alternative Titles



Now we can add in code to handle the alternative titles. The logic we need to implement follows:

1. Make an empty dict to hold all of the alternative titles.
2. Loop through a list of all alternative title keys:
    - Check if the current key exists in the movie object.
    - If so, remove the key-value pair and add to the alternative titles dict.
3. After looping through every key, add the alternative titles dict to the movie object.

> **Skill Drill**
> Try to implement the logic above in your `clean_movie` function on your own.
>
> **Hint:** To remove a key-value pair from a dict in Python, use the `pop()` method.



Step 1: Make an empty dict to hold all of the alternative titles.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    return movie
```

Step 2: Loop through a list of all alternative title keys.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    for key in ['Also known as','Arabic','Cantonese','Chinese','French',
                'Hangul','Hebrew','Hepburn','Japanese','Literally',
                'Mandarin','McCune-Reischauer','Original title','Polish',
                'Revised Romanization','Romanized','Russian',
                'Simplified','Traditional','Yiddish']:

    return movie
```

Step 2a: Check if the current key exists in the movie object.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    for key in ['Also known as','Arabic','Cantonese','Chinese','French',
                'Hangul','Hebrew','Hepburn','Japanese','Literally',
                'Mandarin','McCune-Reischauer','Original title','Polish',
                'Revised Romanization','Romanized','Russian'
```

```
                'Revised Romanization','Romanized','Russian',
                'Simplified','Traditional','Yiddish']:
        if key in movie:

    return movie
```

Step 2b: If so, remove the key-value pair and add to the alternative titles dictionary.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    for key in ['Also known as','Arabic','Cantonese','Chinese','French',
                'Hangul','Hebrew','Hepburn','Japanese','Literally',
                'Mandarin','McCune-Reischauer','Original title','Polish',
                'Revised Romanization','Romanized','Russian',
                'Simplified','Traditional','Yiddish']:
        if key in movie:
            alt_titles[key] = movie[key]
            movie.pop(key)

    return movie
```

Step 3: After looping through every key, add the alternative titles dict to the movie object.

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    for key in ['Also known as','Arabic','Cantonese','Chinese','French',
                'Hangul','Hebrew','Hepburn','Japanese','Literally',
                'Mandarin','McCune-Reischauer','Original title','Polish',
                'Revised Romanization','Romanized','Russian',
                'Simplified','Traditional','Yiddish']:
        if key in movie:
            alt_titles[key] = movie[key]
            movie.pop(key)
    if len(alt_titles) > 0:
```

```
        movie['alt_titles'] = alt_titles

    return movie
```

We can make a list of cleaned movies with a list comprehension:

```
clean_movies = [clean_movie(movie) for movie in wiki_movies]
```

Set `wiki_movies_df` to be the DataFrame created from `clean_movies`, and print out a list of the columns.

```
wiki_movies_df = pd.DataFrame(clean_movies)
sorted(wiki_movies_df.columns.tolist())
```

Here's the printed list:

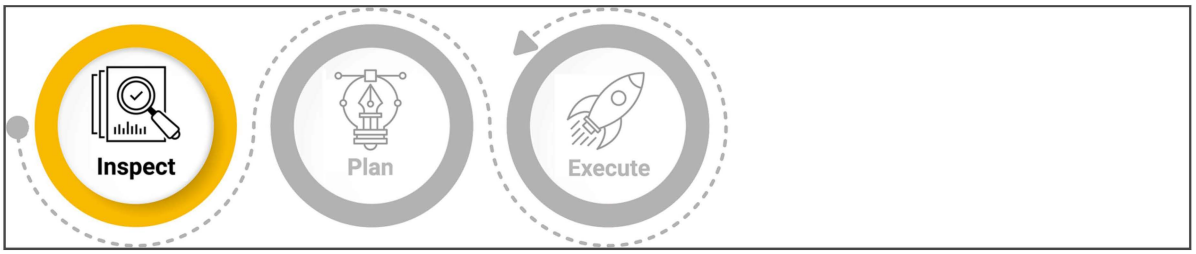
```
['Adaptation by',
 'Animation by',
 'Audio format',
 'Based on',
 'Box office',
 'Budget',
 'Cinematography',
 'Color process',
 'Composer(s)',
 'Country',
 'Country of origin',
 'Created by',
 'Directed by',
 'Director',
 'Distributed by',
 'Distributor',
 'Edited by',
 'Editor(s)',
 'Executive producer(s)',
 'Followed by',
```

We're making a lot of progress! If you've been staring at your screen for awhile, now is a great time to take a brief mental break.

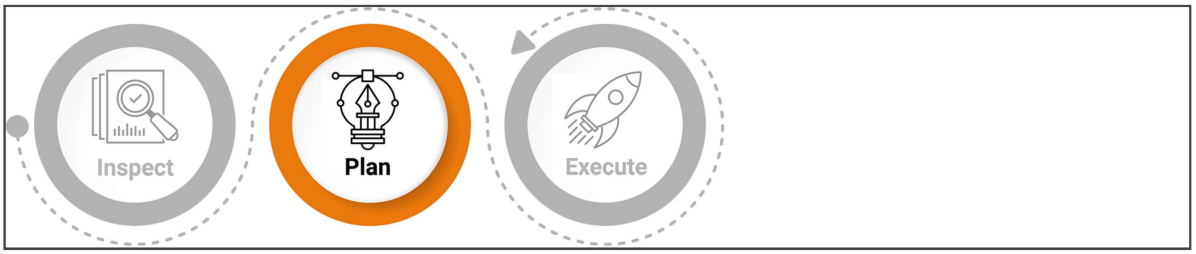
> **Add, Commit, Push**
> Remember to add, commit, and push your work!

8.3.6: Create a Function to Clean the Data, Part 2

You're honest with Britta: sometimes, it takes a long time to clean the data. But you're up to the challenge! Now you'll dig into columns with oh-so-slightly different names.



There are quite a few columns with slightly different names but the same data, such as “Directed by” and “Director.”

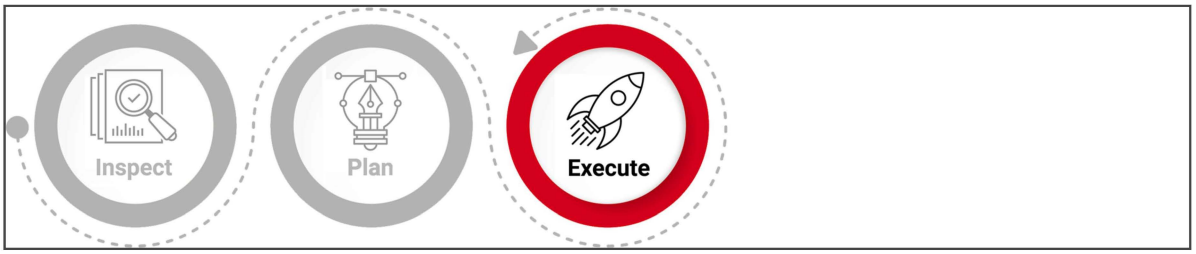


We need to consolidate columns with the same data into one column. We can use the `pop()` method to change the name of a dictionary key, because `pop()` returns the value from the removed key-value pair. We have to check if the key exists in a given movie record, so it will be helpful to make a small function inside `clean_movie()`.

> **Note**
> It's perfectly fine to define a function within another function. In fact, it's often preferable. Functions that are defined within another function live within the scope of the first equation. This means that the inner function can only be called inside the outer function. Outside the original function, it's impossible to call the inner function.

Remember, if Britta needs to go through your notebook to understand your ETL process, it'll be much easier for her to understand if you name your functions as verbs. Also, it's better to be explicit than implicit and write out full words, so we'll call our new function `change_column_name`.

> **Note**
> One of the benefits of using a good, dedicated code editor is that you can autocomplete names, usually by pressing the Tab key when you've partially written a variable, function, or keyword. Programmers used to use short, confusing names so they wouldn't have to type out repetitive code so much, but with autocomplete, we can write more descriptive names for functions and variables.

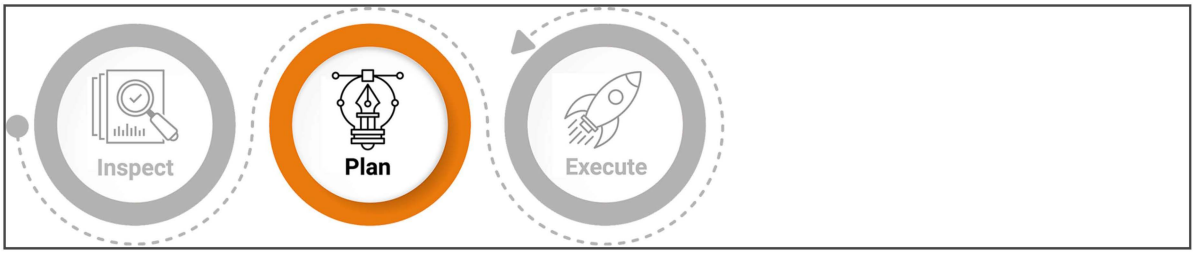


Our new function should look like the following:

```
    def change_column_name(old_name, new_name):
        if old_name in movie:
            movie[new_name] = movie.pop(old_name)
```

To change every instance where the key is “Directed by” to the new key “Director,” write the following inside `clean_movie()`:

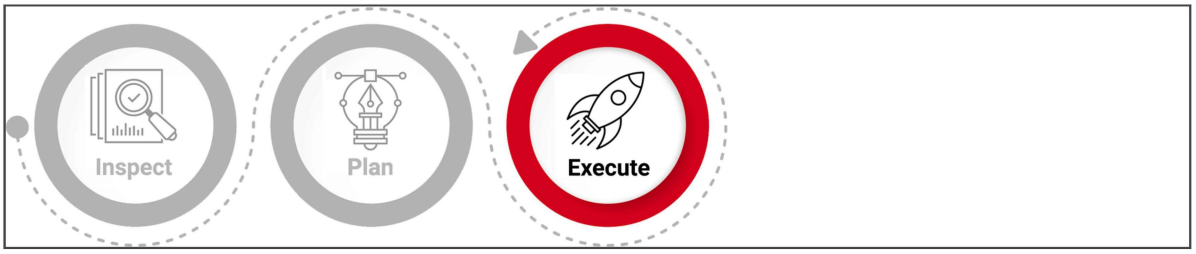
```
change_column_name('Directed by', 'Director')
```



There’s no easy way around the next step: we have to go through each column name and decide if there’s a better name for it. If you’re not sure what the column is referring to, do some research—don’t guess. Use your Google-fu to gain domain knowledge.

> **Important**
> Domain knowledge is specific expertise in the data professional’s industry or field, outside of statistics and coding. For example, a data scientist working in healthcare might need specific clinical knowledge to perform certain analyses accurately.

The most important thing to remember when consolidating the comments is to be consistent. For example, will we use “Composer” or “Composed by”? “Editor” or “Edited by”? Will our columns be capitalized? How will we handle plurals?



Below is an example of how to consolidate columns. Yours might appear slightly different.

```
    change_column_name('Adaptation by', 'Writer(s)')
    change_column_name('Country of origin', 'Country')
    change_column_name('Directed by', 'Director')
    change_column_name('Distributed by', 'Distributor')
    change_column_name('Edited by', 'Editor(s)')
    change_column_name('Length', 'Running time')
    change_column_name('Original release', 'Release date')

    change_column_name('Music by', 'Composer(s)')
    change_column_name('Produced by', 'Producer(s)')
    change_column_name('Producer', 'Producer(s)')
    change_column_name('Productioncompanies ', 'Production company(s)')
    change_column_name('Productioncompany ', 'Production company(s)')
    change_column_name('Released', 'Release Date')
    change_column_name('Release Date', 'Release date')
    change_column_name('Screen story by', 'Writer(s)')
    change_column_name('Screenplay by', 'Writer(s)')
    change_column_name('Story by', 'Writer(s)')
    change_column_name('Theme music composer', 'Composer(s)')
    change_column_name('Written by', 'Writer(s)')
```

The function `clean_movie()` is starting to look a little complicated, so we should add some commenting to make it easier to understand. The whole function should look like this:

```
def clean_movie(movie):
    movie = dict(movie) #create a non-destructive copy
    alt_titles = {}
    # combine alternate titles into one list
    for key in ['Also known as','Arabic','Cantonese','Chinese','French',
                'Hangul','Hebrew','Hepburn','Japanese','Literally',
                'Mandarin','McCune-Reischauer','Original title','Polish',
                'Revised Romanization','Romanized','Russian',
                'Simplified','Traditional','Yiddish']:
        if key in movie:
```

```
            alt_titles[key] = movie[key]
            movie.pop(key)
    if len(alt_titles) > 0:
        movie['alt_titles'] = alt_titles

    # merge column names
    def change_column_name(old_name, new_name):
        if old_name in movie:
            movie[new_name] = movie.pop(old_name)
    change_column_name('Adaptation by', 'Writer(s)')
    change_column_name('Country of origin', 'Country')
    change_column_name('Directed by', 'Director')
    change_column_name('Distributed by', 'Distributor')
    change_column_name('Edited by', 'Editor(s)')
    change_column_name('Length', 'Running time')
    change_column_name('Original release', 'Release date')
    change_column_name('Music by', 'Composer(s)')
    change_column_name('Produced by', 'Producer(s)')
    change_column_name('Producer', 'Producer(s)')
    change_column_name('Productioncompanies ', 'Production company(s)')
    change_column_name('Productioncompany ', 'Production company(s)')
    change_column_name('Released', 'Release Date')
    change_column_name('Release Date', 'Release date')
    change_column_name('Screen story by', 'Writer(s)')
    change_column_name('Screenplay by', 'Writer(s)')
    change_column_name('Story by', 'Writer(s)')
    change_column_name('Theme music composer', 'Composer(s)')
    change_column_name('Written by', 'Writer(s)')

    return movie
```

Now we can rerun our list comprehension to clean `wiki_movies` and recreate `wiki_movies_df`.

```
clean_movies = [clean_movie(movie) for movie in wiki_movies]
wiki_movies_df = pd.DataFrame(clean_movies)
```

Note

When using notebooks like Jupyter, it's easy to lose track of the order in which the code was run if you edit functions in previous cells and jump around between different cells. It's best to keep the flow of the notebook linear, if possible.

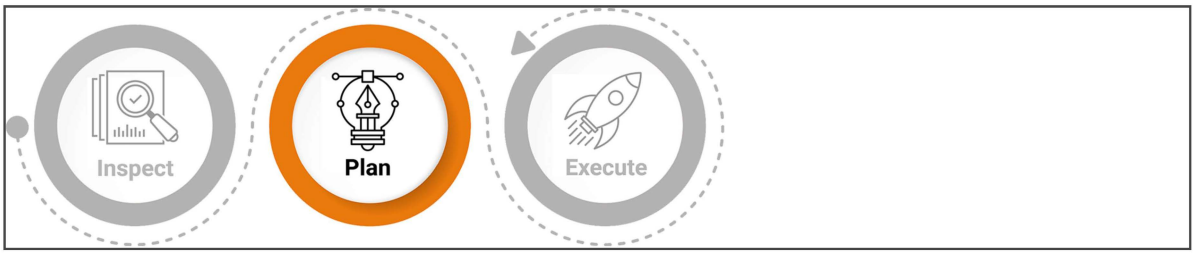
To track why certain decisions were made, show the evolution of the function through multiple cells.

Clear documentation is one of the best ways to set yourself apart from other programmers on the job. If your programming portfolio contains well-documented code and notebooks, it will also set you apart in interviews.

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

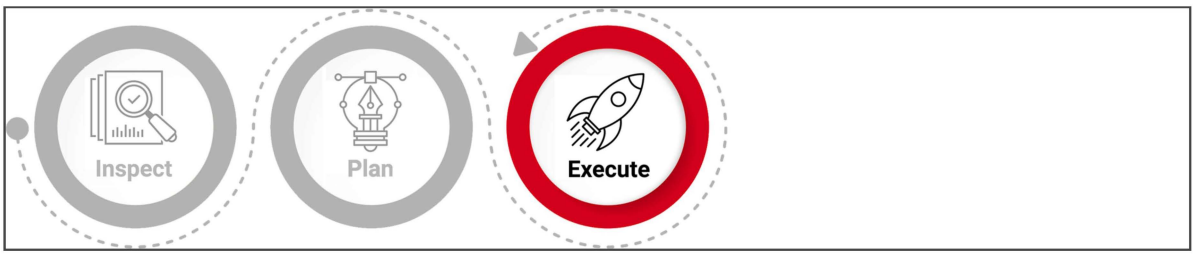
> for entering dates. You could say it's a regular expression that you can easily recognize because it follows a well-defined pattern. In the same way, "(###) ###-####" is a pattern for entering U.S. phone numbers. Regular expressions are just a more formal way of defining these kinds of patterns so 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. 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
```



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." We also want to see the new number of rows and how many rows were dropped. The whole cell should look like the following:

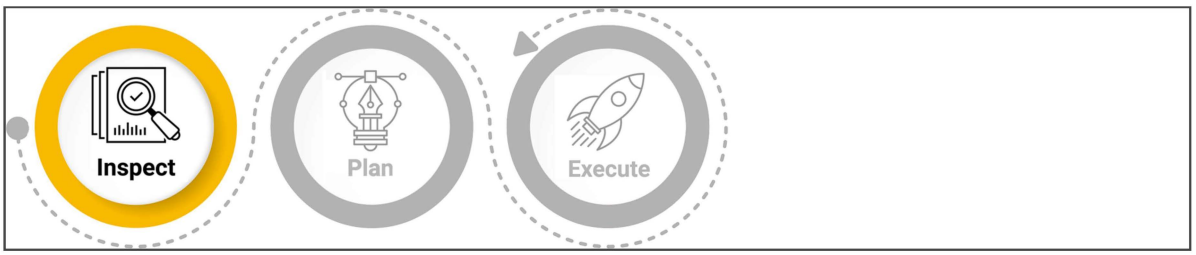
```
wiki_movies_df['imdb_id'] = wiki_movies_df['imdb_link'].str.extract(r'(tt
print(len(wiki_movies_df))
wiki_movies_df.drop_duplicates(subset='imdb_id', inplace=True)
print(len(wiki_movies_df))
wiki_movies_df.head()
```

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.



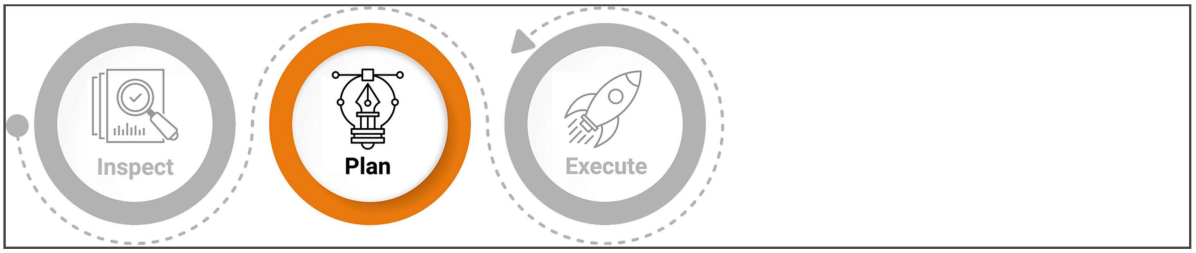
One way to get the count of null values for each column is to use a list comprehension, as shown below.

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

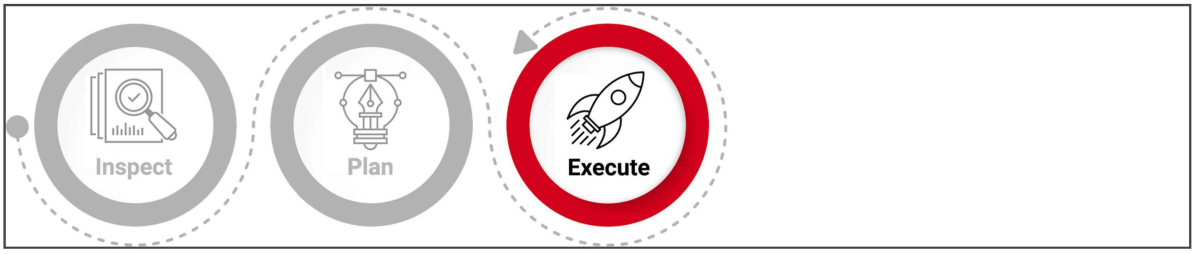
The output would be:

```
[['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],
 ['Writer(s)', 199],
```

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].is
```

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 wi
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.

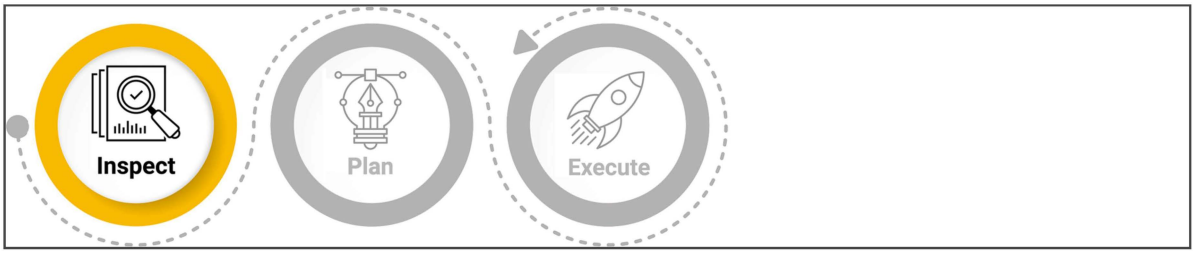
And with that, we've reduced 191 messy columns down to 21 useful, data-filled columns. That's awesome data-cleaning work!

8.3.8: Make a Plan to Convert and Parse the Data

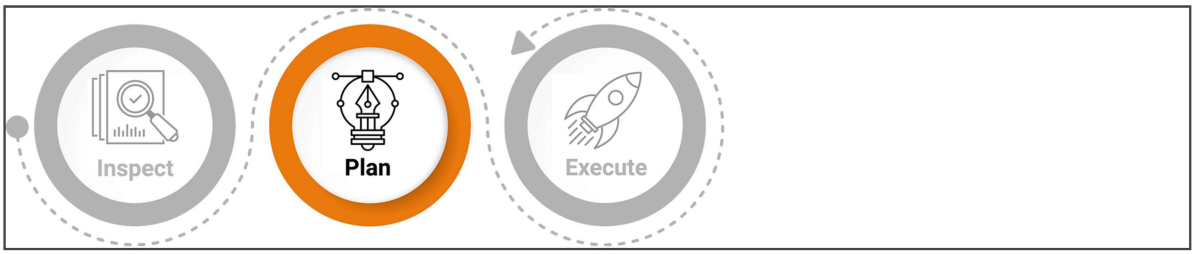
The Wikipedia data is now structured in tabular form, but Britta needs it to have the right data types once it's in the SQL table. For example, we can't do analysis on numeric data if it's stored as a string in the SQL table—it needs to be stored in numeric format. Some of the data also has numeric information written out (like the word "million"). To convert those columns to numbers, the data needs to be parsed.

We've got our DataFrame columns trimmed down to just the ones we want, but some of the columns have data stored as text when it should be a different data type, such as numeric data or datetimes. To convert those columns, we need to understand how to use regular expressions.

Regular expressions are ridiculously powerful, but they can be intimidating at first. There are even seasoned programmers who still don't feel entirely comfortable with regular expressions, but don't worry. We'll break down each step, and before you know it, you'll be rocking regular expressions like a pro.



First, we need to identify which columns need to be converted. `wiki_movies_df.dtypes` will display the data type for each column.



Looking through the data, column by column, we see that:

- **Box office** should be numeric.
- **Budget** should be numeric.
- **Release date** should be a date object.
- **Running time** should be numeric.

The box office and budget amounts aren't written in a consistent style, so we're going to need a powerful way to parse their data correctly. We've already dipped a toe into regular expressions; now it's time to dive all the way in.

We'll start on the box office data, which should give us code that we can reuse and tweak for the budget data since they're both currency. It will be helpful to only look at rows where box office data is defined, so first we'll make a data series that drops missing values with the following:

```
box_office = wiki_movies_df['Box office'].dropna()
```

> **Pause**
> As a habit, always check the number of data points that exist after you drop any data. Here we have 5,485 movies with box office data.
>
> Does that seem like a reasonable amount? (Take a minute to think about it before peeking at the next paragraph.)

> It's about 5,500 movies out of 7,000, which is a little more than three-quarters. Box office data is reported by multiple sources, and we'd expect some percentage of them to not have reliable box office numbers, or for smaller indie films to not have any box office numbers published at all. Twenty-five percent would mean the bottom quartile of movies has no box office data, which seems a little high, but for every movie missing box office data, there are a little more than three movies that do have box office data. Also, 5,500 is still a good number of movies to perform analysis on (more than 180 movies per year).

Regular expressions only work on strings, so we'll need to make sure all of the box office data is entered as a string. By using the `apply()` method, we can see which values are not strings. First, make a `is_not_a_string()` function:

```
def is_not_a_string(x):
    return type(x) != str
```

Then add the following:

```
box_office[box_office.map(is_not_a_string)]
```

The output would be:

```
34                          [US$, 4,212,828]
54      [$6,698,361 (, United States, ), [2]]
74                   [$6,488,144, (US), [1]]
126                [US$1,531,489, (domestic)]
130                         [US$, 4,803,039]
                      ...
6980               [$99.6, million, [4], [5]]
6994                   [$365.6, million, [1]]
6995                         [$53.8, million]
7015                    [$435, million, [7]]
7048                   [$529.3, million, [4]]
Name: Box office, Length: 135, dtype: object
```

Having to create a new function every time we want to use the `map()` method is cumbersome and interrupts the readability of our code. What we want is a stripped-down, one-line way of writing our functions. Also, we don't need to use it ever again outside of our `map()` call, so we don't need to give it a name. If you think we're talking about types of functions that will work here, you're right.

> **Rewind**
> Remember, this is what lambda functions are made for. Instead of creating a new function with a block of code and the `def` keyword, we can create an anonymous lambda function right inside the `map()` call.

Remember, lambda functions don't have a name (because they don't need one) and automatically return a variable. They use the following syntax:

```
lambda arguments: expression
```

So the lambda function version of `is_not_a_string()` is:

```
lambda x: type(x) != str
```

We can update our `map()` call to use the lambda function directly instead of using `is_not_a_string()`:

```
box_office[box_office.map(lambda x: type(x) != str)]
```

The output would be:

```
34                          [US$, 4,212,828]
54      [$6,698,361 (, United States, ), [2]]
74                   [$6,488,144, (US), [1]]
126               [US$1,531,489, (domestic)]
130                         [US$, 4,803,039]
```

```
                        ...
6980                [$99.6, million, [4], [5]]
6994                    [$365.6, million, [1]]
6995                          [$53.8, million]
7015                     [$435, million, [7]]
7048                    [$529.3, million, [4]]
Name: Box office, Length: 135, dtype: object
```

From the output, we can see that there are quite a few data points that are stored as lists. There is a `join()` string method that concatenates list items into one string; however, we can't just type `join(some_list)` because the `join()` method belongs to string objects. We need to make a separator string and then call the `join()` method on it. For example, the code would be:

```
some_list = ['One','Two','Three']
'Mississippi'.join(some_list)
```

The outputs would be:

```
'OneMississippiTwoMississippiThree'
```

We'll use a simple space as our joining character and apply the `join()` function only when our data points are lists. The code looks like the following:

```
box_office = box_office.apply(lambda x: ' '.join(x) if type(x) == list el
```

Looking through the data, many of the box office numbers are written either like "$123.4 million" (or billion) or "$123,456,789." We'll use regular expressions to find out just how many of each style are in our data.

There is a built-in Python module for regular expressions: `re`. We'll need to import that library, so add the line below to the first cell, with the other import statements, and rerun the cell.

```
import re
```

> **Note**
> Python best practices recommend putting all of your import statements at the beginning of your program. If you realize that you need to import another module, it's better to add it to the top rather than have import statements scattered throughout your program, especially when using Jupyter notebooks. Because cells can be run out of order, when import statements are used in other cells, it's easy for them to get lost, and for a cell to have code that depends on another module end up before the cell that imports the module.

Now let's start writing some regular expressions.

8.3.9: Write Regular Expressions

You knew this day would come: the day when you conquered regular expressions. While they may sound a bit underwhelming, regular expressions are actually quite powerful. (You could almost say the outsized impact they will allow you to have on your dataset is *irregular.*) Puns aside, you know a big reason Britta trusted you with this dataset is because you wouldn't shy away from learning how to use and apply regular expressions.

Regular expressions are just strings of characters that are used as a search pattern. They are used to test if strings are in a specific format or contain a substring in a specific format, to extract pertinent information from strings while discarding unnecessary information, and to perform complicated replacements of substrings.

Regular expressions are used in almost all general-purpose languages like Python. For example, they are also used in JavaScript, C#, and Java. Sometimes they are the only viable solution to a problem.

Each character in a regular expression serves a purpose, based on what kind of character it is. We'll break down the different character types and the purposes they serve.

Literal Characters

Literal characters are the simplest character class. A regular expression made of literal characters will match any string that contains the expression as a substring. For example, if we made a regular expression of the string "cat" and used it to search for any matches in another string—"The tomcat was placated with the catch of the day"—it would match three times: "The tomcat was placated with the catch of the day."

However, regular expressions also have special characters that make it much more powerful than just finding a substring. We've already used the special character `\d` to find any digit from 0 to 9.

> **Note**
> You might be thinking, "But `\d` is two characters!" You're not wrong. It is written with two characters. However, backslashes in strings have a long history of being used to write special characters that would be difficult to enter directly.
>
> For example, there is a character in ASCII and Unicode for creating a new line (it's aptly named "newline character"). However, adding it to a string is difficult to do, so we write `\n` inside the string, which Python converts to the newline character. Regular expressions also treat character combinations that start with a backslash as one character, so it's fine to refer to them as a single character.

Character Types: \d, \w, \s (and \D, \W, \S)

As we've seen, `\d` is a special character that matches any digit from 0 to 9. There are other special characters like `\d`. The full list is:

- `\d` will match any digit from 0 to 9.
- `\D` will match any non-digit character.
- `\w` matches a word character (a letter, digit, or underscore).
- `\W` matches any non-word character (anything other than a letter, digit, or underscore, such as spaces and punctuation).

- `\s` will match any whitespace character (including spaces, tabs, and newlines).
- `\S` will match any non-whitespace characters.

Character Sets: [ ]

If we need to be more specific than digits, alphanumeric characters, or whitespace characters, we can use the square brackets to define a character set. For example, "[ceh]at" would match "'cat", "eat", "hat", and "that", but not "rat."

We can also specify ranges of characters inside a character set. For example:

- `"[a-z]"` matches any lowercase letter.
- `"[A-Z]"` matches any uppercase character.
- `"[0-9]"` matches any digit.

We can include multiple ranges, so:

- `"[a-zA-Z]"` matches any lowercase or uppercase letter.
- `"[a-zA-Z0-9]"` matches any alphanumeric character.

But, we can also have smaller ranges, such as:

- `"[A-E]"` would match "A", "B", "C", "D", or "E".
- `"[1-3]"` would match the digits "1", "2", or "3".

We can also include character types inside a character set, so `"[a-zA-Z\d]"` and `"[a-zA-Z0-9]"` are equivalent expressions that would match any alphanumeric character.

Inside of a character set, we can specify a character that we do NOT want to include by prefacing it with a caret: `^`.

Which regular expression would match both `"million"` and `"billion"`?

- [ ] `"[mb]illion"`
- [ ] `"[bm]illion"`
- [ ] `"[million][billion]"`
- [ ] `"mb[illion]"`

Check Answer

Finish ▸

Match (Almost) Everything: .

The period, or dot (`.`), is a **wildcard** in regular expressions, which means it will match any single character whether it is a digit, a letter, whitespace, or punctuation. The only thing that a dot won't match is a line break (remember, line breaks are also stored as characters). In Python's regular expression module, there's an option to make the dot match every character, including line breaks.

Escaping: \

The dot and square brackets are examples of **metacharacters** in regular expressions. Metacharacters are like the superheroes in a regular expression because they have powers like "match everything" or "create a character set."

But what if we need metacharacters to act like ordinary literal characters? For example, if we use the regular expression "ca.," the dot will match any character, so "cat", "car", "cab", "ca!", "ca?", and "ca." would all be matches.

What if we want to specifically search for only "ca." with an actual period? The period by itself in a regular expression is being a superhero, matching everything. So when we want it to act like just a literal character, we need to give it a secret identity. We use the backslash "`\`" to do this. The backslash tells the parser to

treat the upcoming metacharacter like a literal character. So, “ca\.” will only match “ca.”, “cat”, “car”, and “cab”, and the rest of them won’t be matched.

The backslash in a regular expression is called the **escape character**. It says that the next character gets to escape its duties as a special character in the regular expression and act like a plain old literal character. We’ll meet more superpowered metacharacters, such as curly brackets, parentheses, and plus signs. If we want to match text that has curly brackets, parentheses, or plus signs, we’ll use the backslash to treat these superpowered metacharacters like literal characters.

The backslash, \, is also a superpowered character. How would you write a regular expression that matches `“cat\dog”` ?

- `“cat\dog”`
- `“cat/dog”`
- `“cat\\dog”`
- `“cat\\\dog”`

Check Answer

Finish ▸

Special Counting Characters: *, +, {}, ?

There are also special **counting characters** that specify how many times a character can show up.

The first counting character is the asterisk: *****. In regular expressions, the asterisk says the previous character can repeat any number of times, including zero. So, “ca*t” would match “cat” and “caaat” but also “fiction.” If we want to specify that the character has to show up at least once, we use the plus sign: **+**. So, “ca+t” would match “cat” and “caaat” but not “fiction.”

If we want to search for a character that shows up an exact number of times, we use the curly brackets: **{ }**. When we extracted the IMDb IDs, we wanted IDs that

had exactly seven digits, or `"\d{7}"`. We can also put two numbers in curly brackets, and that would match for any number of digits within that range. So, “ca{3}t” wouldn’t match “cat” or “fiction” but would match “caaat.” “ca{3,5}t” would match “caaat”, “caaaat”, and “caaaaat” but not “cat” or “caaaaaat.”

Finally, the question mark can be considered a counting character as well. The question mark is for optional characters, which means they can show up zero or one time. So, “ca?t” is equivalent to the regular expression pattern “ca{0,1}t”.

Alternation: |

If we want to search for a given string or a different string, we use the alternation character, or pipe: |. This essentially functions as a logical OR. For example, if we wanted to match “cat” or “mouse” or “dog,” we would make a string “cat|mouse|dog.”

String Boundaries: ^ and $

If we need to make sure that our expression matches only at the beginning or ending of the string, we use the caret (^) to represent the beginning of the string, and the dollar sign ($) to represent the end of the string. So, “^cat” would match “cat” and “catatonic,” but not “concatenate.” “cat$” would also match “cat” and “tomcat,” but not “catatonic.”

By themselves, the string boundaries represent **zero length matches**; in other words, they don’t match any actual characters themselves, just the boundaries of the string being searched.

Capture Groups: ( )

Grouping in regular expressions serves two purposes. First, groups can be used to add structure to a search pattern. For example, “1,000”, “1,000,000”, and “1,000,000,000” as strings all have a similar structure. There is a comma followed by three zeros that repeats as a group. We can match all of these with one regular expression, using parentheses to create a capture group. One regular expression that matches all three strings would be “1(,000)+.”

The second purpose for grouping is hinted at in the name "capture group." **Capture groups** are how regular expressions define what information should be extracted.

For example, when we needed to extract IMDb IDs from the links, we put the entire expression inside brackets. This can also be helpful when you need to make sure a long string matches a certain format, but you only need a substring inside of it.

For example, `"\d{3}-\d{3}-\d{4}"` would match any phone number in the form "333-333-4444," but if you wanted to extract only the digits, you would use `"(\d{3})-(\d{3})-(\d{4})"`, and only the digits would be captured. Specifically, the digits of the phone number would be captured into three groups: area code, prefix, and line number.

Non-Capturing Groups and Negative Lookahead Groups: (?:), (?!)

We can modify the behavior of a group by including a question mark after the opening parenthesis. The first modification is a **non-capturing group**, which uses a colon after the question mark. This specifies that we only want to use the grouping structure, and we do not need to capture the information.

Non-capturing groups can feel superfluous when we're using regular expressions for just matching, but they become very important when we use regular expressions for matching and replacing.

Suppose we're anonymizing a list of phone numbers of the form "333-333-4444," and we want to change the prefix to "555," like the fake phone numbers in movies. We still need to have groups in our regular expression for the area code and the four-digit line number, but we don't want to capture them—we only want to capture the prefix.

The regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"` will match numbers of the form "123-456-7890," but it will only capture the middle group, the prefix. For example:

- "212-012-9876" matches the regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"`, but only "012" is captured. So, if we used this regular expression

to replace the captured text with “555,” it would turn “212-012-9876” to “212-555-9876.”

- However, “012-3456” wouldn’t match at all, because there’s no area code. Even though the area code is in a non-capturing group, the regular expression still needs to see it before it can make a match.

Negative lookahead groups are also non-capturing groups, but they look ahead in the text and make sure a string doesn’t exist after the match.

For example, imagine we have text with phone numbers still in the form “333-333-4444,” but the text also contains ID numbers that are of the form “333-333-55555.” The regular expression we’ve been using—`"(\d{3})-(\d{3})-(\d{4})"`—will see the first 10 numbers of the ID and recognize a match and return “333-333-5555” as if it were a phone number.

What we need is a regular expression that matches the first 10 numbers, but also checks that there isn’t another digit after the phone number. We need a group that looks ahead of the rest of the regular expression, and reports back “negative” if there’s some text we don’t want to see.

That’s a negative lookahead group. Negative lookahead groups start with a question mark and an exclamation mark. So, to make sure there are no extra digits, a negative lookahead group would be `"(?!\d)"`. Our new regular expression is `"(\d{3})-(\d{3})-(\d{4})(?!\d)"`:

- “333-333-4444” will match.
- “333-333-55555” will not.

Here’s a cheat sheet for everything related to regular expressions that we’ve covered so far. The highlighted text in the Example column denotes a match.

Note: For instances of “ ”, only the whitespace between the quotation marks is a match, not the quotes themselves.

Character	Function	Example

literal characters	Directly matches characters	`"cat"` • “cat” • “dog” (no match)
`\d`	Matches a digit from 0 to 9	`"\d"` • “1” • “A” (no match) • “_” (no match) • “!” (no match) • “ ” (no match)
`\D`	Matches a non-digit	`"\D"` • “1” (no match) • “A” • “_” • “!” • “ ”
`\w`	Matches a word character (letter, digit, or underscore)	`"\w"` • “1” • “A” • “_” • “!” (no match) • “ ” (no match)
`\W`	Matches any non-word character	`"\W"` • “1” (no match) • “A” (no match) • “_” (no match) • “!” • “ ”

`\s`	Matches any whitespace character, such as spaces and tabs	`"\s"` • “1” (no match) • “A” (no match) • “_” (no match) • “!” (no match) • “ ”
`\S`	Matches any non-whitespace character	`\S` • “1” • “A” • “_” • “!” • “ ” (no match)
`[ … ]`	**Character Set** Matches any characters inside the brackets. Can specify ranges of characters as well.	`"[A-C]"` • “A” • “B” • “C” • “D” (no match) • “E” (no match)
`[^ … ]`	**Negative Character Set** Matches anything *not* inside the brackets	`"[^C-E]"` • “A” • “B” • “C” (no match) • “D” (no match) • “E” (no match)
`.`	**Wildcard**	`"."` • “1”

	Matches any character (except a newline)	• “A” • “_” • “!” • “ ”
`*`	Matches 0 or more times	`"ca*t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`+`	Matches 1 or more times	`"ca+t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`?`	Matches 0 or 1 time	`"ca?t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`{#}`	Matches a specific number of times	`"ca{2}t"` • “ct” • “cat” • “caat” • “caaat”

		• “caaaat”
`{#,}`	Matches at least a specific number of times	`"ca{2,}t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`{#,#}`	Matches within a specific range of times	`"ca{2,3}t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`|`	**Alternation** Matches either the expression before or the expression after	`"cat|dog"` • “cat” • “dog” • “bird”
`^`	Start of the string	`"^cat"` • “cat” • “catsup” • “concatenate” (no match) • “kitty-cat” (no match)
`$`	End of the string	`"cat$"` • “cat”

		• “catsup” (no match) • “concatenate” (no match) • “kitty-cat”
`\`	**Escape Character** Escapes the next character to be treated as a literal character	`"\$"` • “$”
`( … )`	**Capture Group** Identifies matches that should be extracted	`"c(at)"` • “cat” (“at” is captured) • “bat” (no match)
`(?: … )`	**Non-Capturing Group** Identifies matches that should not be extracted	`"c(?:at)"` • “cat” • “bat” (no match)
`(?! … )`	**Negative Lookahead Group** Identifies expressions that negate earlier matches	`"cat(?! burglar)"` • “cat” • “cats” • “cat burglar” (no match)

It might feel as if we've taken a deep dive into regular expressions, but there's even more they can do. It starts getting wild quickly, with different languages having slightly different implementations. It's extremely helpful to use a regular expression tester like **RegExr (https://regexr.com/)** or **RegEx101 (https://regex101.com/)** when building more complicated regular expressions.

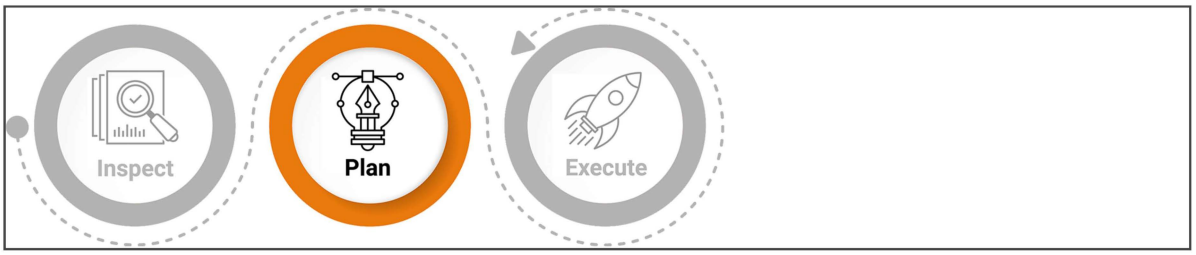
Even though there's still more to regular expressions, we now have enough for us to parse the information in our dataset, so let's get back to it.

8.3.10: Parse the Box Office Data

Now that you have added regular expressions to your analyst toolbelt, it's time to apply this tool to the box office data.

Remember, there are two main forms the box office data is written in: "$123.4 million" (or billion), and "$123,456,789." We're going to build a regular expression for each form, and then see what forms are left over.

Create the First Form



For the first form, our pattern match string will include six elements in the following order:

1. A dollar sign
2. An arbitrary (but non-zero) number of digits
3. An optional decimal point
4. An arbitrary (but possibly zero) number of more digits
5. A space (maybe more than one)
6. The word "million" or "billion"

We'll translate those rules into a regular expression, step by step.

Step 1: Start with a dollar sign.

The dollar sign is a special character in regular expressions, so we'll need to escape it.

What is the regular expression that matches a dollar sign?

- `$`
- `\$`
- `$$`
- `\\$`

Check Answer

Finish ▸

Step 2: Add an arbitrary (but non-zero) number of digits.

We'll add the `\d` character to specify digits only, and the `+` modifier to capture one or more digits. Our regular expression string now appears as `"\$\d+"`.

Step 3: Add an optional decimal point.

Remember, the decimal point is a special character, so it needs to be escaped with a backslash. Since the decimal point is optional, add a question mark modifier after it. Our regular expression string now appears as `"\$\d+\.?"`.

Step 4: Add an arbitrary (but possibly zero) number of more digits.

Once again, we'll use the `\d` character to specify digits only, but now with the `*` modifier because there may be no more digits after the decimal point. Our regular expression string now appears as `"\$\d+\.?\d*"`.

Step 5: Add a space (maybe more than one).

Now we're going to use the `\s` character to match whitespace characters. To be safe, we'll match any number of whitespace characters with the `*` modifier. Our regular expression string now appears as `"\$\d+\.?\d*\s*"`.

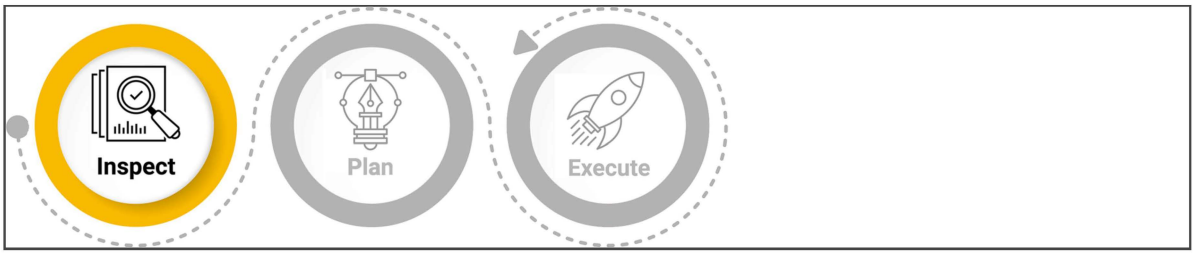
Step 6: Add the word "million" or "billion."

Since "million" and "billion" only differ by one letter, we can match it with a character set for the first letter. We specify character sets with square brackets, so we'll add `"[mb]illion"` to the end of our string. Our finished regular expression string now appears as `"\$\d+\.?\d*\s*[mb]illion"`.

Create a variable `form_one` and set it equal to the finished regular expression string. Because we need the escape characters to remain, we need to preface the string with an `r`.

```
form_one = r'\$\d+\.?\d*\s*[mb]illion'
```

> **Note**
> You might be wondering if we're going to miss any box office values that have uppercase letters. Don't worry—when we use the `contains()` method, we will specify an option to ignore case.



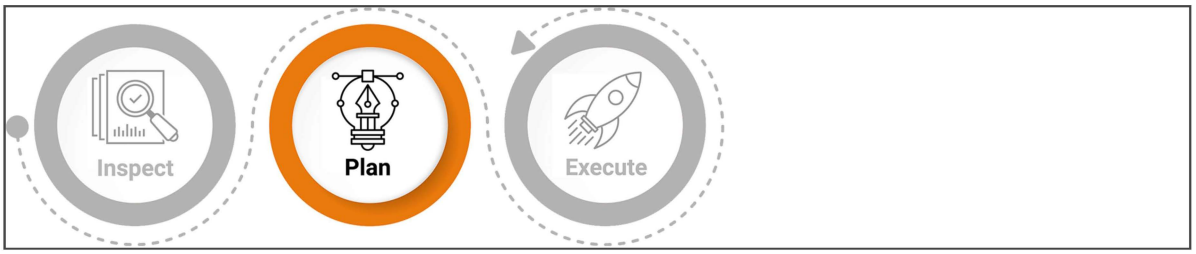
Now, to count up how many box office values match our first form. We'll use the `str.contains()` method on `box_office`. To ignore whether letters are uppercase or lowercase, add an argument called `flags`, and set it equal to `re.IGNORECASE`.

Finally, we can call the `sum()` method to count up the total number that return True. Your code should look like the following:

```
box_office.str.contains(form_one, flags=re.IGNORECASE).sum()
```

> **Finding**
> There are 3,896 box office values that match the form "$123.4 million/billion."

Create the Second Form



Next, we'll match the numbers of our second form, "$123,456,789." In words, our pattern match string will include the following elements:

1. A dollar sign
2. A group of one to three digits
3. At least one group starting with a comma and followed by exactly three digits

Step 1: Start with a dollar sign.

Once again, we need to escape the dollar sign for it to match. Our regular expression string starts like this: `"\$"`.

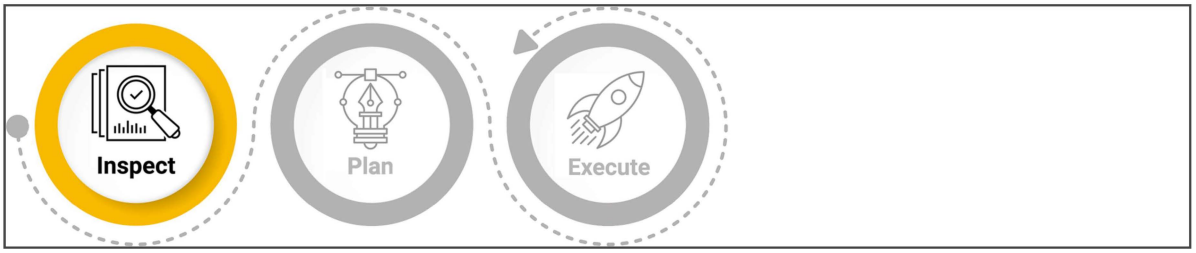
Step 2: Add a group of one to three digits.

We'll use the `\d` character for digits, but this time, we'll modify it with curly brackets to only match one through three repetitions. Our regular expression string now appears as `"\$\d{1,3}"`.

Step 3: Match at least one group starting with a comma and followed by exactly three digits.

To match a comma and exactly three digits, we'll use the string `",\d{3}"`. To match any repetition of that group, we'll put it inside parentheses, and then put a plus sign after the parentheses: `"(,\d{3})+"`. We'll add one more modification to specify that this is a non-capturing group by inserting a question mark and colon after the opening parenthesis: `"(?:,\d{3})+"`. Our finished regular expression string now appears as `"\$\d{1,3}(?:,\d{3})+"`.

Create another variable `form_two` and set it equal to the finished regular expression string. Don't forget to make it a raw string so Python keeps the escaped characters.



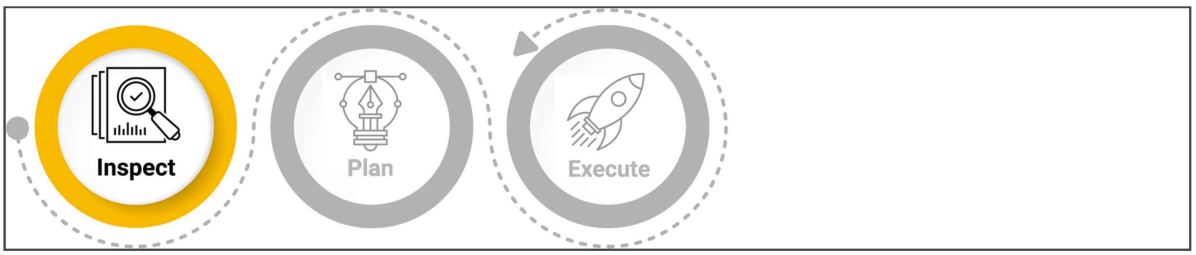
Now count up the number of box office values that match this pattern. Don't forget to put an `r` before the string and set the flags option to include `re.IGNORECASE`.

Your code should look like this:

```
form_two = r'\$\d{1,3}(?:,\d{3})+'
box_office.str.contains(form_two, flags=re.IGNORECASE).sum()
```

> **Finding**
> There are 1,544 box office values that match the form "$123,456,789."

Compare Values in Forms



Most of the box office values are described by either form. Now we want to see which values aren't described by either. To be safe, we should see if any box office values are described by both.

To make our code easier to understand, we'll create two Boolean Series called `matches_form_one` and `matches_form_two`, and then select the box office values that don't match either. First, create the two Boolean series with the following code:

```
matches_form_one = box_office.str.contains(form_one, flags=re.IGNORECASE)
matches_form_two = box_office.str.contains(form_two, flags=re.IGNORECASE)
```

Recall the Python logical keywords "not," "and," and "or." Try the following code to see which values in `box_office` don't match either form.

```
# this will throw an error!
box_office[(not matches_form_one) and (not matches_form_two)]
```

The code above will give you a `ValueError` with the explanation "The truth value of a Series is ambiguous." (Unfortunately, the meaning of that error is also ambiguous.)

Instead, Pandas has element-wise logical operators:

- The element-wise negation operator is the tilde: ~ (similar to "not")
- The element-wise logical "and" is the ampersand: &
- The element-wise logical "or" is the pipe: |

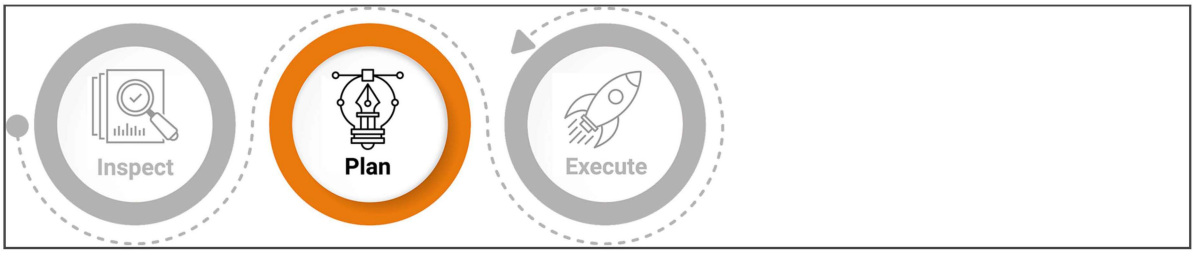
The code we want to use is as follows:

```
box_office[~matches_form_one & ~matches_form_two]
```

Your results should look like the following:

```
34                          US$ 4,212,828
79                               $335.000
110                    $4.35-4.37 million
130                         US$ 4,803,039
600                            $5000 (US)
731                          $ 11,146,270
957                             $ 50,004
1070                          35,254,617
1147    $ 407,618 (U.S.) (sub-total) [1]
1446                         $ 11,829,959
1480                           £3 million
1611                             $520.000
1865                          ¥1.1 billion
2032                                   N/A
2091                                $309
2130               US$ 171.8 million [9]
2257                    US$ 3,395,581 [1]
2263            $ 1,223,034 ( domestic )
2347                             $282.175
2638            $ 104,883 (US sub-total)
2665         926,423 admissions (France)
```

Fix Pattern Matches



We can fix our pattern matches to capture more values by addressing these issues:

1. Some values have spaces in between the dollar sign and the number.
2. Some values use a period as a thousands separator, not a comma.
3. Some values are given as a range.
4. "Million" is sometimes misspelled as "millon."

1. Some values have spaces in between the dollar sign and the number.

This is easy to fix. Just add `\s*` after the dollar signs. The new forms should look like the following:

```
form_one = r'\$\s*\d+\.?\d*\s*[mb]illion'
form_two = r'\$\s*\d{1,3}(?:,\d{3})+'
```

2. Some values use a period as a thousands separator, not a comma.

This is slightly more complicated, but doable. Simply change `form_two` to allow for either a comma or period as a thousands separator, like so:

```
form_two = r'\$\s*\d{1,3}(?:[,\.]\d{3})+'
```

The results will also match values like 1.234 billion, but we're trying to change raw numbers like $123.456.789. We don't want to capture any values like 1.234 billion, so we need to add a negative lookahead group that looks ahead for "million" or "billion" after the number and rejects the match if it finds those strings. Don't forget the space! The new form should look like this:

```
form_two = r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)'
```

3. Some values are given as a range.

To solve this problem, we'll search for any string that starts with a dollar sign and ends with a hyphen, and then replace it with just a dollar sign using the `replace()` method. The first argument in the `replace()` method is the substring that will be replaced, and the second argument in the `replace()` method is the string to replace it with. We can use regular expressions in the first argument by sending the parameter `regex=True`, as shown below.

```
box_office = box_office.str.replace(r'\$.*[---](?![a-z])', '$', regex=Tru
```

> **Caution**
> Always be wary of parsing dashes. The character you can type on standard keyboards is a hyphen, but some editors will convert them in certain situations to em dashes and en dashes. To learn more than you've ever wanted to know about dashes, see the **Wikipedia page for "Dash." (https://en.wikipedia.org/wiki/Dash)** We'll need to put all three into a character set in our replace regular expression.

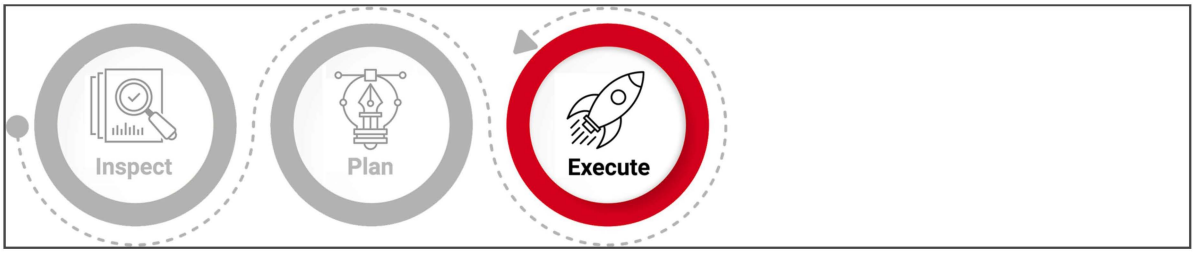
4. "Million" is sometimes misspelled as "millon."

This is easy enough to fix; we can just make the second "i" optional in our match string with a question mark as follows:

```
form_one = r'\$\s*\d+\.?\d*\s*[mb]illi?on'
```

The rest of the box office values make up such a small percentage of the dataset and would require too much time and effort to parse correctly, so we'll just ignore them.

We're finished writing our regular expressions for the box office values. The hard part is over. Save your work and take a quick break if you need one—you earned it!



Extract and Convert the Box Office Values

Now that we've got expressions to match almost all the box office values, we'll use them to extract only the parts of the strings that match. We do this with the `str.extract()` method. This method also takes in a regular expression string, but it returns a DataFrame where every column is the data that matches a capture group. We need to make a regular expression that captures data when it matches either `form_one` or `form_two`. We can do this easily with an f-string.

The f-string `f'{form_one}|{form_two}'` will create a regular expression that matches either `form_one` or `form_two`, so we just need to put the whole thing in parentheses to create a capture group. Our final string will be `f'({form_one}|{form_two})'`, and the full line of code to extract the data follows:

```
box_office.str.extract(f'({form_one}|{form_two})')
```

Now we need a function to turn the extracted values into a numeric value. We'll call it `parse_dollars`, and `parse_dollars` will take in a string and return a floating-point number. We'll start by making a skeleton function with comments explaining each step, and then fill in the steps with actual code.

```
def parse_dollars(s):
    # if s is not a string, return NaN

    # if input is of the form $###.# million

        # remove dollar sign and " million"

        # convert to float and multiply by a million
```

```
        # return value

    # if input is of the form $###.# billion

        # remove dollar sign and " billion"

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###

        # remove dollar sign and commas

        # convert to float

        # return value

    # otherwise, return NaN
```

Since we're working directly with strings, we'll use the `re` module to access the regular expression functions. We'll use `re.match(pattern, string)` to see if our string matches a pattern. To start, we'll make some small alterations to the forms we defined, splitting the million and billion matches from form one.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " million"

        # convert to float and multiply by a million
```

```
        # return value

    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flag

        # remove dollar sign and commas

        # convert to float

        # return value

    # otherwise, return NaN
    else:
        return np.nan
```

Next, we'll use `re.sub(pattern, replacement_string, string)` to remove dollar signs, spaces, commas, and letters, if necessary.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " million"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a million
```

```
        # return value

    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flag

        # remove dollar sign and commas
        s = re.sub('\$|,','', s)

        # convert to float

        # return value

    # otherwise, return NaN


    else:
        return np.nan
```

Finally, convert all the strings to floats, multiply by the right amount, and return the value.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):
```

```
        # remove dollar sign and " million"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a million
        value = float(s) * 10**6

        # return value
        return value

    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a billion
        value = float(s) * 10**9

        # return value

        return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flag

        # remove dollar sign and commas
        s = re.sub('\$|,','', s)

        # convert to float
        value = float(s)

        # return value
        return value

    # otherwise, return NaN
    else:
        return np.nan
```

Now we have everything we need to parse the box office values to numeric values.

First, we need to extract the values from `box_office` using `str.extract`. Then we'll apply `parse_dollars` to the first column in the DataFrame returned by `str.extract`, which in code looks like the following:

```
wiki_movies_df['box_office'] = box_office.str.extract(f'({form_one}|{form
```

We no longer need the Box Office column, so we'll just drop it:

```
wiki_movies_df.drop('Box office', axis=1, inplace=True)
```

8.3.11: Parse Budget Data

Luckily, you already did a lot of the heavy lifting for parsing the budget data when you parsed the box office data. You'll use the same pattern matches and see how many budget values are in a different form.

Luckily, we've already done a lot of the heavy lifting for parsing the budget data when we parsed the box office data. We'll use the same pattern matches and see how many budget values are in a different form. First, we need to preprocess the budget data, just like we did for the box office data.

Create a budget variable with the following code:

```
budget = wiki_movies_df['Budget'].dropna()
```

Convert any lists to strings:

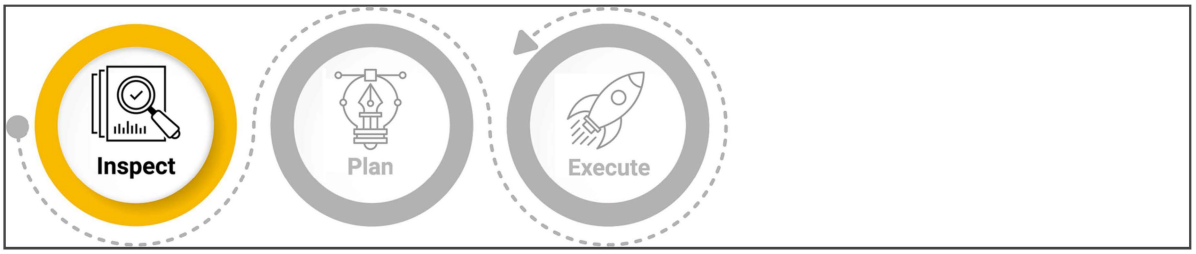
```
budget = budget.map(lambda x: ' '.join(x) if type(x) == list else x)
```

Then remove any values between a dollar sign and a hyphen (for budgets given in ranges):

```
budget = budget.str.replace(r'\$.*[---](?![a-z])', '$', regex=True)
```

Now test your skills in the following Skill Drill.

> **Skill Drill**
>
> Use the same pattern matches that you created to parse the box office data, and apply them without modifications to the budget data. Then, look at what's left.



Your code should look like this:

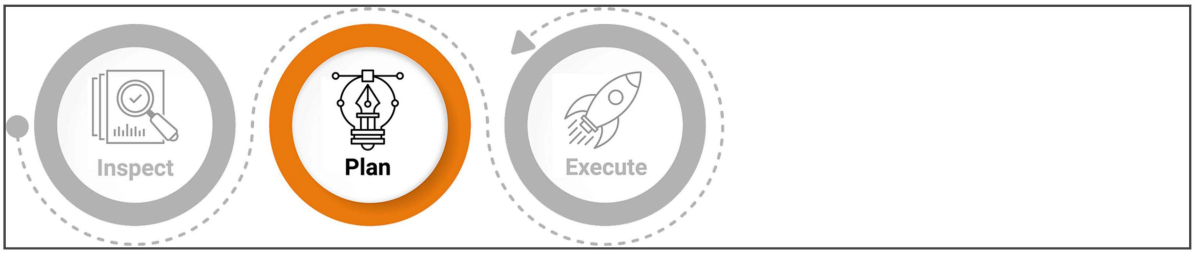
```
matches_form_one = budget.str.contains(form_one, flags=re.IGNORECASE)
matches_form_two = budget.str.contains(form_two, flags=re.IGNORECASE)
budget[~matches_form_one & ~matches_form_two]
```

The output should look like this:

```
136                          Unknown
204     60 million Norwegian Kroner
478                          Unknown
973              $34 [3] [4] million
1126                $120 [4] million
1226                         Unknown
1278                             HBO
1374                     £6,000,000
1397                      13 million
1480                   £2.8 million
1734                   CAD2,000,000
1913     PHP 85 million (estimated)
1948                    102,888,900
1953                   3,500,000 DM
1973                     £2,300,874
2281                      $14 milion
```

```
2451                    £6,350,000
3144                 € 40 million
3360            $150 [6] million
3418                    $218.32
```

Not bad! That parsed almost all of the budget data. However, there's a new issue with the budget data: citation references (the numbers in square brackets).



We can remove those fairly easily with a regular expression.

What regular expression will match a number within square brackets?

- "[0-9]"
- "[\d+]"
- "\[\d\]"
- "\[\d+\]"

Check Answer

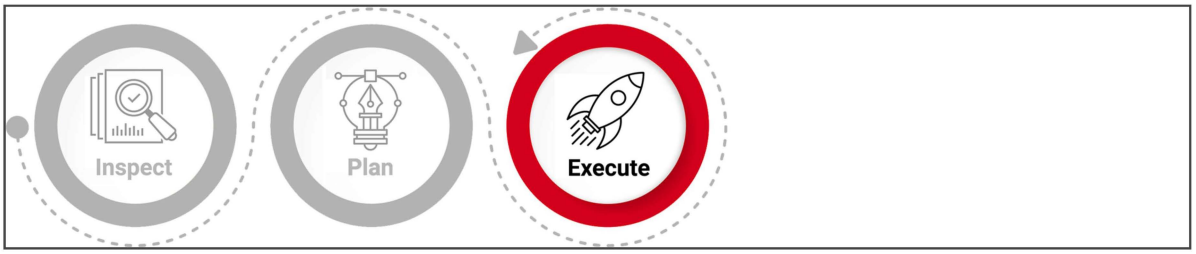
Finish ▸

Remove the citation references with the following:

```
budget = budget.str.replace(r'\[\d+\]\s*', '')
budget[~matches_form_one & ~matches_form_two]
```

There will be 30 budgets remaining.

> **Pause**
>
> Is it worth our time to try and parse what we can out of these remaining 30 budget values? A handful of them don't even have numeric values, and those that do tend to be in a different currency. Converting currencies can open up a whole can of worms about where to get conversion rates, what rates should be used, rates from what date should be used, etc.
>
> There are a handful of values that could be parsed into usable data points without worrying about currency conversion, but we have almost 4,700 other budget values to work with, so even 30 values is less than 1% of the data.
>
> Given all the time in the world, maybe it would be worth it to get every last data point into our data, but time is a valuable resource, and putting in the time to convert what we can from these remaining values won't give us enough valuable data to be worth our time.
>
> Or as they say, "The juice isn't worth the squeeze."



Everything is now ready to parse the budget values. We can copy the line of code we used to parse the box office values, changing "box_office" to "budget":

```
wiki_movies_df['budget'] = budget.str.extract(f'({form_one}|{form_two})',
```

We can also drop the original Budget column.

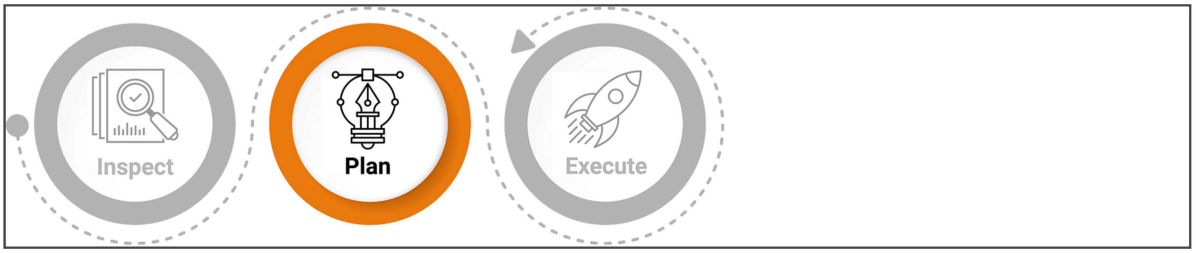
```
wiki_movies_df.drop('Budget', axis=1, inplace=True)
```

Parse Release Date

Parsing the release date will follow a similar pattern to parsing box office and budget, but with different forms.

First, make a variable that holds the non-null values of Release date in the DataFrame, converting lists to strings:

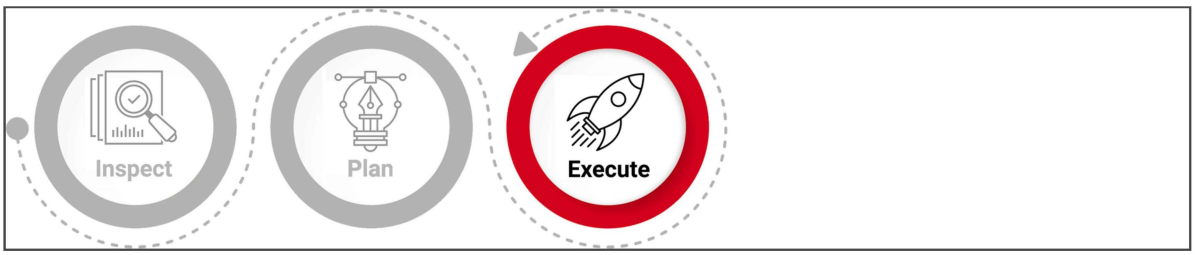
```
release_date = wiki_movies_df['Release date'].dropna().apply(lambda x: '
```



The forms we'll be parsing are:

1. Full month name, one- to two-digit day, four-digit year (i.e., January 1, 2000)
2. Four-digit year, two-digit month, two-digit day, with any separator (i.e., 2000-01-01)
3. Full month name, four-digit year (i.e., January 2000)
4. Four-digit year

> **Skill Drill**
> Try to figure out the regular expressions for each form before moving on. Test them in your Jupyter Notebook with some test strings.



One way to parse those forms is with the following:

```
date_form_one = r'(?:January|February|March|April|May|June|July|August|Se
date_form_two = r'\d{4}.[01]\d.[123]\d'
date_form_three = r'(?:January|February|March|April|May|June|July|August|
date_form_four = r'\d{4}'
```

And then we can extract the dates with:

```
release_date.str.extract(f'({date_form_one}|{date_form_two}|{date_form_th
```

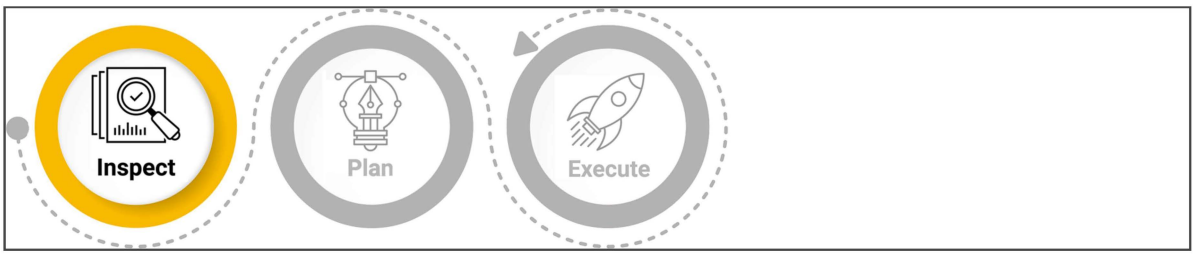
Instead of creating our own function to parse the dates, we'll use the built-in `to_datetime()` method in Pandas. Since there are different date formats, set the `infer_datetime_format` option to `True`.

```
wiki_movies_df['release_date'] = pd.to_datetime(release_date.str.extract(
```

Parse Running Time

First, make a variable that holds the non-null values of Release date in the DataFrame, converting lists to strings:

```
running_time = wiki_movies_df['Running time'].dropna().apply(lambda x: '
```



There are 6,859 entries in `running_time`, and it looks like most of the entries just look like "100 minutes." Let's see how many running times look exactly like that by using string boundaries.

```
running_time.str.contains(r'^\d*\s*minutes$', flags=re.IGNORECASE).sum()
```

The above code returns 6,528 entries. Let's get a sense of what the other 366 entries look like.

```
running_time[running_time.str.contains(r'^\d*\s*minutes$', flags=re.IGNOR
```

The output should look like this:

```
9                                                    102 min
26                                                    93 min
28                                                   32 min.
34                                                   101 min
35                                                    97 min
                                  ...
6500       114 minutes [1] 120 minutes (extended edition)
6643                                                104 mins
6709    90 minutes (theatrical) [1] 91 minutes (unrate...
7057    108 minutes (Original cut) 98 minutes (UK cut)...
7075                 Variable; 90 minutes for default path
Name: Running time, Length: 366, dtype: object
```

Let's make this more general by only marking the beginning of the string, and accepting other abbreviations of "minutes" by only searching up to the letter "m."

```
running_time.str.contains(r'^\d*\s*m', flags=re.IGNORECASE).sum()
```

That accounts for 6,877 entries. The remaining 17 follow:

```
running_time[running_time.str.contains(r'^\d*\s*m', flags=re.IGNORECASE)
```

The output should look like the following.

```
668                    UK:84 min (DVD version) US:86 min
727                       78-102 min (depending on cut)
840                     Varies (79 [3] -84 [1] minutes)
1347                                          25 : 03
1442    United States: 77 minutes Argentina: 94 minute...
1498                                          1hr 35min
1550                                             varies
1773                   Netherlands:96 min, Canada:95 min
1776                                   approx. 14 min
2272                                       1 h 43 min
2990                                           1h 48m
3919                                          4 hours
4418    US domestic version: 86 minutes Original versi...
4959    Theatrical cut: 97 minutes Unrated cut: 107 mi...
5416                   115 [1] /123 [2] /128 [3] minutes
5439                                1 hour 32 minutes
7038               Variable; 90 minutes for default path
```

We can capture some more of these by relaxing the condition that the pattern has to start at the beginning of the string, but the entries with hours and minutes listed separately will give erroneous data.

What is the new regular expression that relaxes the condition of patterns starting at the beginning of the string?

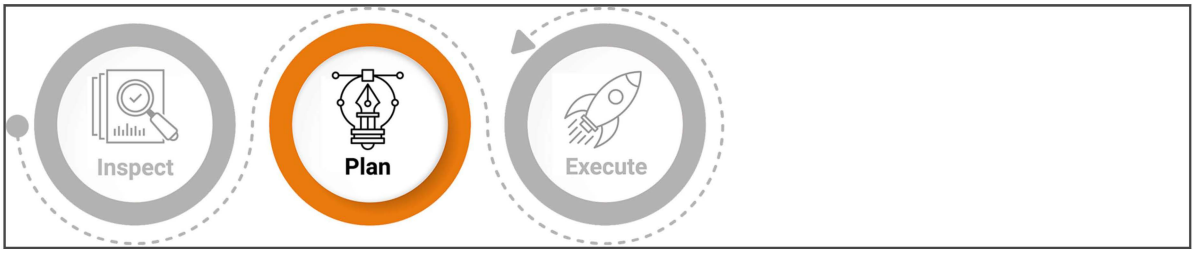
- '^\d*\s*m'
- r'\d*\s*m'
- r'^\d\s*m'
- r'^\d*\sm'

Check Answer

Finish ▸

Even though it's a very small number of entries, it's not too hard to parse, so we'll go ahead and parse those, too.

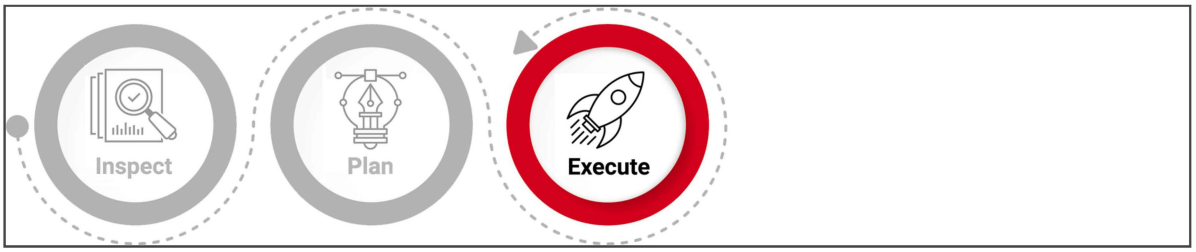
> **Pause**
>
> This is another judgment call. It's only 17 entries out of almost 7,000, so it's highly unlikely that our analysis will be affected by just ignoring these data points. In a time crunch, it would be perfectly acceptable to just move on. However, it's not very difficult to parse these new forms, and we'll have more flexible code if we do. If we decide to do another, larger scrape of Wikipedia data, it's entirely possible that a significant portion have their runtime formatted this way.



We can match all of the hour + minute patterns with one regular expression pattern. Our pattern follows:

1. Start with one digit.
2. Have an optional space after the digit and before the letter “h.”
3. Capture all the possible abbreviations of “hour(s).” To do this, we’ll make every letter in “hours” optional except the “h.”
4. Have an optional space after the “hours” marker.
5. Have an optional number of digits for minutes.

As a pattern, this looks like `"\d+\s*ho?u?r?s?\s*\d*"`.



With our new pattern, it’s time to extract values. We only want to extract digits, and we want to allow for both possible patterns. Therefore, we’ll add capture groups around the `\d` instances as well as add an alternating character. Our code will look like the following.

```
running_time_extract = running_time.str.extract(r'(\d+)\s*ho?u?r?s?\s*(\c
```

Unfortunately, this new DataFrame is all strings, we’ll need to convert them to numeric values. Because we may have captured empty strings, we’ll use the `to_numeric()` method and set the errors argument to `'coerce'`. Coercing the errors will turn the empty strings into Not a Number (NaN), then we can use `fillna()` to change all the NaNs to zeros.

```
running_time_extract = running_time_extract.apply(lambda col: pd.to_numer
```

Now we can apply a function that will convert the hour capture groups and minute capture groups to minutes if the pure minutes capture group is zero, and save the output to `wiki_movies_df`:

```
wiki_movies_df['running_time'] = running_time_extract.apply(lambda row: r
```

Finally, we can drop `Running time` from the dataset with the following code:

```
wiki_movies_df.drop('Running time', axis=1, inplace=True)
```

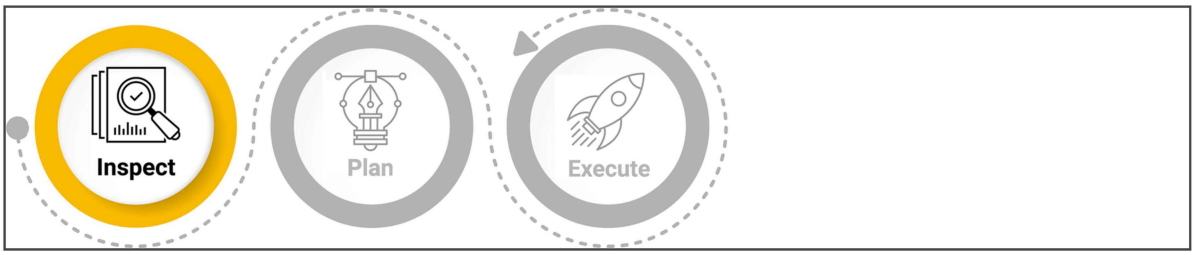
The Wikipedia dataset has been cleaned! Save your notebook and give yourself a pat on the back—you just did some hard work.

8.3.12: Clean the Kaggle Data

The Kaggle data that Britta found is much more structured, but it still requires some cleaning, including converting strings to correct data types. Therefore, your next task is to clean the Kaggle data.

As always when data cleaning, the first step is to take an initial look at the data you're working with. Let's get started.

Initial Look at the Movie Metadata



Because the Kaggle data came in as a CSV, one of the first things we want to check is that all of the columns came in as the correct data types.

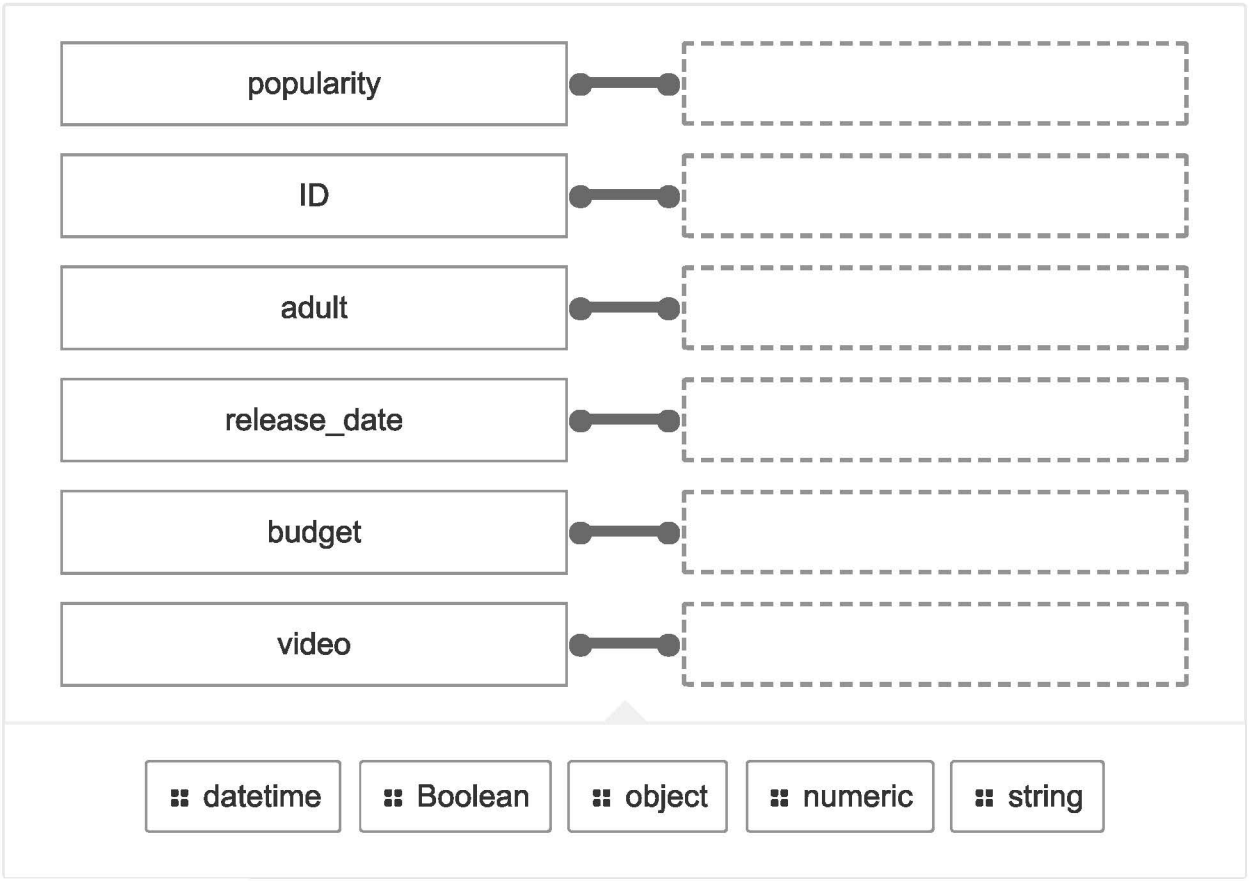
```
kaggle_metadata.dtypes
```

Here's what the output will look like:

```
adult                     object
belongs_to_collection     object
budget                    object
genres                    object
homepage                  object
id                        object
imdb_id                   object
original_language         object
original_title            object
overview                  object
popularity                object
poster_path               object
production_companies      object
production_countries      object
release_date              object
revenue                  float64
runtime                  float64
spoken_languages          object
status                    object
tagline                   object
title                     object
video                     object
vote_average             float64
vote_count               float64
dtype: object
```

Remember, the "object" data type is usually for strings. Only four columns were successfully converted to a data type—`revenue`, `runtime`, `vote_average`, and `vote_count`—but taking a look through the DataFrame, we can see some columns that should be specific data types.

Match the column name to the data type. Note that data types may be used more than once or not used at all.



Check Answer

Finish ▸

We'll just go down the list and convert the data types for each of the six columns that need to be converted.

Before we convert the "adult" and "video" columns, we want to check that all the values are either `True` or `False`.

```
kaggle_metadata['adult'].value_counts()
```

Here's what the output will look like.

```
False
True
```

```
 Avalanche Sharks tells the story of a bikini contest that turns into a h
 Rune Balot goes to a casino connected to the October corporation to try
 - Written by Ørnås
Name: adult, dtype: int64
```

Clearly, we have some bad data in here. Let's remove it.

Remove Bad Data

To remove the bad data, use the following:

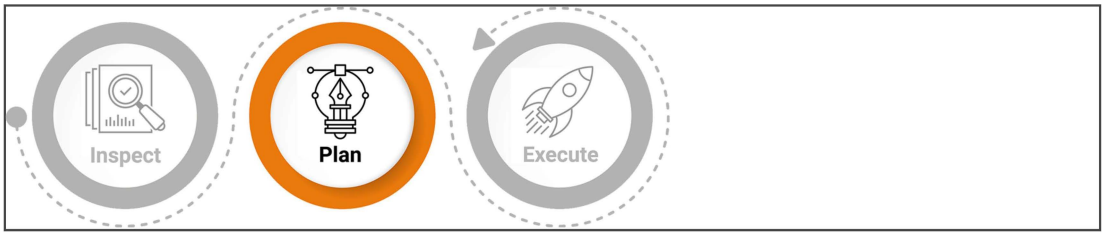
```
kaggle_metadata[~kaggle_metadata['adult'].isin(['True','False'])]
```

Take a closer look at the three movies that appear to have corrupted data:

	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id	original_language	original_title	overv
19730	- Written by Ørnås	0.065736	/ff9qCepilowshEtG2GYWwzt2bs4.jpg	[{'name': 'Carousel Productions', 'id': 11176}...	[{'iso_3166_1': 'CA', 'name': 'Canada'}, {'iso...	1997-08-20	0	104.0	[{'iso_639_1': 'en', 'name': 'English'}]	Relea
29503	Rune Balot goes to a casino connected to the ...	1.931659	/zV8bHuSL6WXoD6FWogP9j4x80bL.jpg	[{'name': 'Aniplex', 'id': 2883}, {'name': 'Go...	[{'iso_3166_1': 'US', 'name': 'United States o...	2012-09-29	0	68.0	[{'iso_639_1': 'ja', 'name': '日本語'}]	Relea
35587	Avalanche Sharks tells the story of a bikini ...	2.185485	/zaSf5OG7V8X8gqFvly88zDdRm46.jpg	[{'name': 'Odyssey Media', 'id': 17161}, {'nam...	[{'iso_3166_1': 'CA', 'name': 'Canada'}]	2014-01-01	0	82.0	[{'iso_639_1': 'en', 'name': 'English'}]	Relea

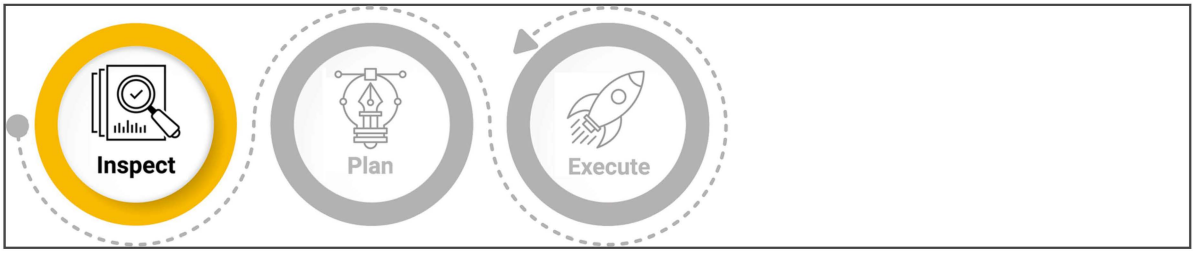
Somehow the columns got scrambled for these three movies.

> **Pause**
> How do we fix the data here?
>
> Ideally, we'd want to be able to unscramble the rows and recover the data. But since we don't know what caused the data to be scrambled, it's also possible that even if we got all the data into the right columns, the data would still be corrupt.

> The biggest concern is that none of the data in these rows looks like an `imdb_id`. Since that's missing, there's no amount of rearranging that will make these rows into good data. We're just going to have to drop them.
>
> 
>
>
> In fact, since we probably don't want to include adult movies in the hackathon dataset, we'll only keep rows where `adult` is `False`, and then drop the "adult" column.

The following code will keep rows where the adult column is `False`, and then drop the adult column.

```
kaggle_metadata = kaggle_metadata[kaggle_metadata['adult'] == 'False'].dr
```



Next, we'll look at the values of the video column:

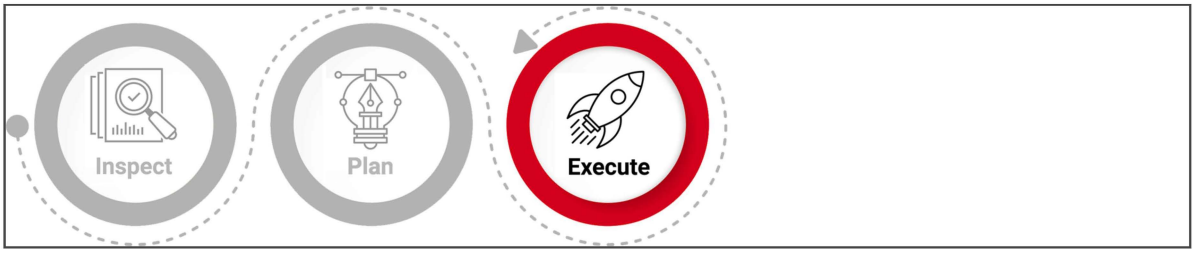
```
kaggle_metadata['video'].value_counts()
```

Here's what the output should look like.

```
False    45358
True        93
Name: video, dtype: int64
```

Great, there are only `False` and `True` values. We can convert `video` fairly easily.

Convert Data Types

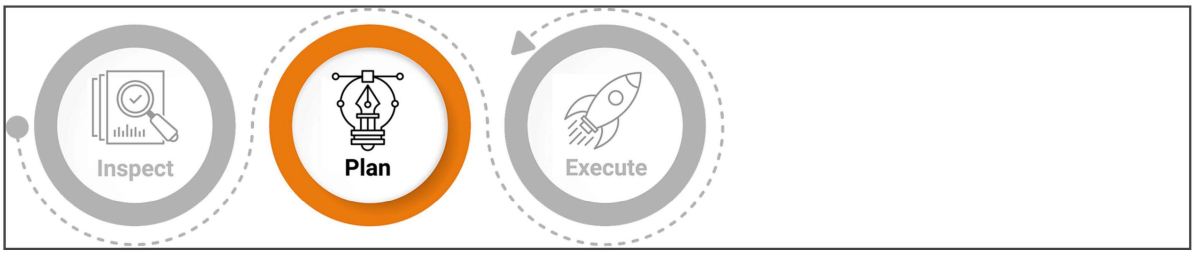


To convert, use the following code:

```
kaggle_metadata['video'] == 'True'
```

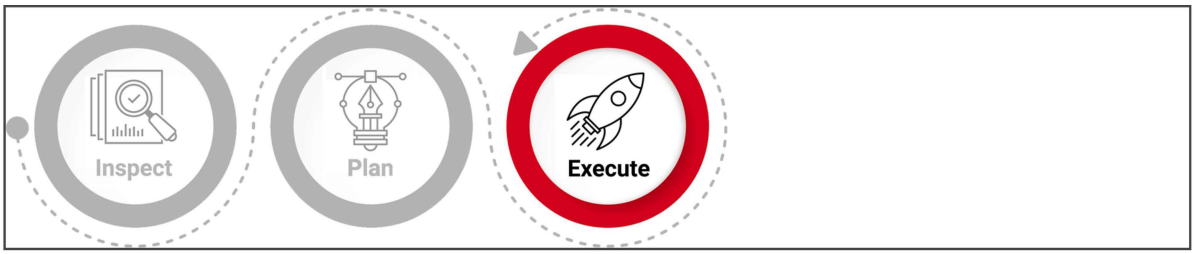
The above code creates the Boolean column we want. We just need to assign it back to `video`:

```
kaggle_metadata['video'] = kaggle_metadata['video'] == 'True'
```



For the numeric columns, we can just use the `to_numeric()` method from Pandas. We'll make sure the `errors=` argument is set to `'raise'`, so we'll know if

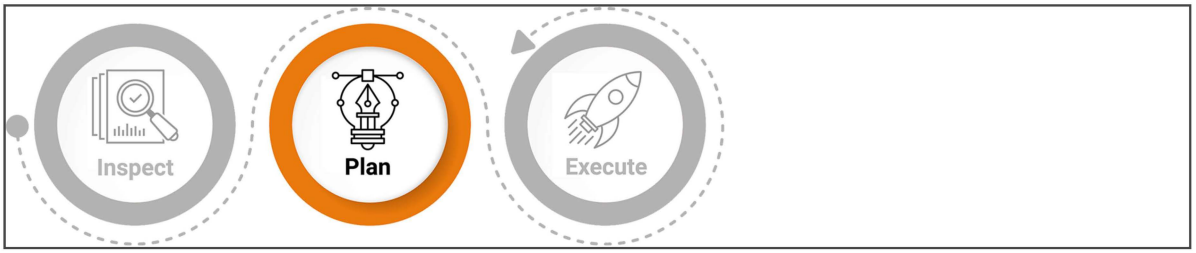
there's any data that can't be converted to numbers.



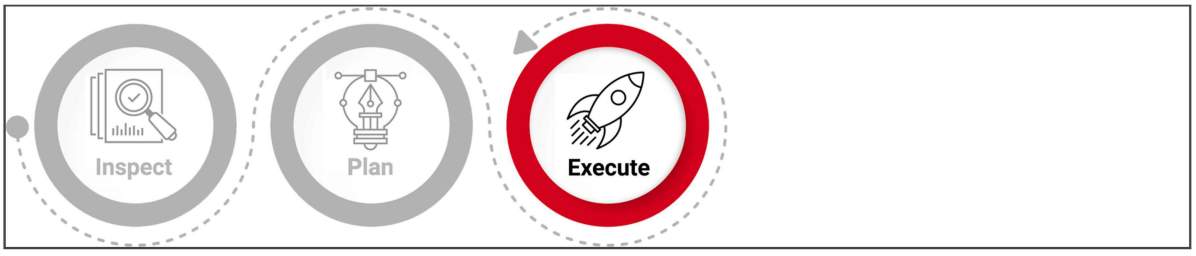

```
kaggle_metadata['budget'] = kaggle_metadata['budget'].astype(int)
kaggle_metadata['id'] = pd.to_numeric(kaggle_metadata['id'], errors='rais
kaggle_metadata['popularity'] = pd.to_numeric(kaggle_metadata['popularity
```

This code runs without errors, so everything converted fine.

Finally, we need to convert `release_date` to datetime. Luckily, Pandas has a built-in function for that as well: `to_datetime()`.



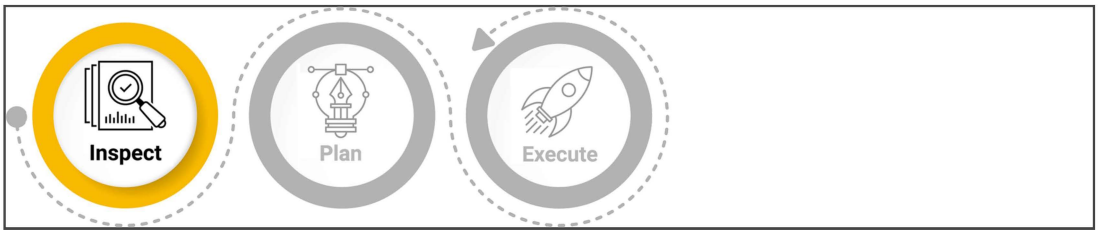
Since `release_date` is in a standard format, `to_datetime()` will convert it without any fuss.



```
kaggle_metadata['release_date'] = pd.to_datetime(kaggle_metadata['release
```

And that's it for cleaning the Kaggle metadata!

Reasonability Checks on Ratings Data



Lastly, we'll take a look at the ratings data. We'll use the `info()` method on the DataFrame. Since the ratings dataset has so many rows, we need to set the `null_counts` option to `True`.

```
ratings.info(null_counts=True)
```

The output should look like the following.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26024289 entries, 0 to 26024288
Data columns (total 4 columns):
userId       26024289 non-null int64
movieId      26024289 non-null int64
rating       26024289 non-null float64
timestamp    26024289 non-null int64
dtypes: float64(1), int64(3)
memory usage: 794.2 MB
```

For our own analysis, we won't be using the timestamp column; however, we will be storing the rating data as its own table in SQL, so we'll need to convert it to a datetime data type. From the MovieLens documentation, the timestamp is the number of seconds since midnight of January 1, 1970.

> **Important**
> Storing time values as a data type is difficult, and there are many, many standards out there for time values. Some store time values as text strings, like the ISO format "1955-11-05T12:00:00," but then calculating the difference between two time values is complicated and computationally expensive. The Unix time standard stores points of time as integers, specifically as the number of seconds that have elapsed since midnight of January 1, 1970. This is known as the Unix **epoch**. There are other epochs in use, but the Unix epoch is by far the most widespread.

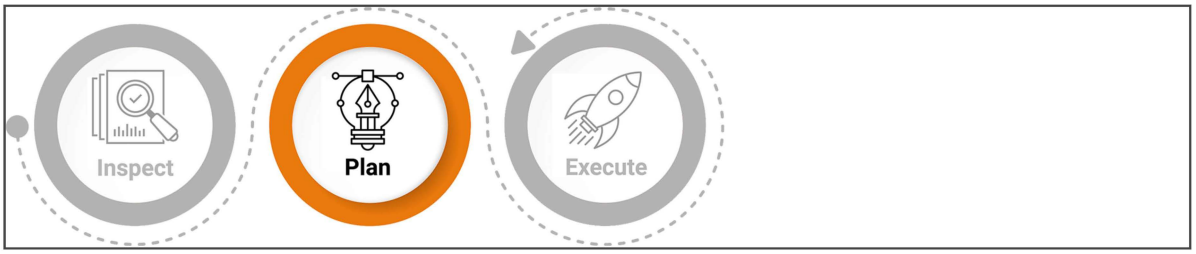
We'll specify in `to_datetime()` that the origin is `'unix'` and the time unit is seconds.

```
pd.to_datetime(ratings['timestamp'], unit='s')
```

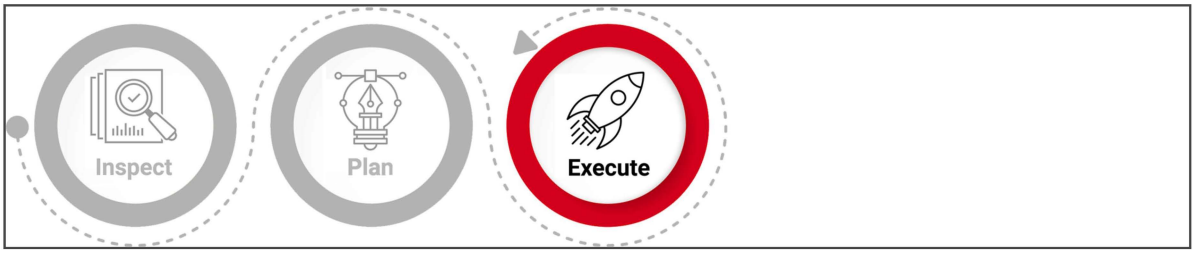
The output should look like the following.

```
0         2015-03-09 22:52:09
1         2015-03-09 23:07:15
2         2015-03-09 22:52:03
3         2015-03-09 22:52:26
4         2015-03-09 22:52:36
5         2015-03-09 23:02:28
6         2015-03-09 22:48:20
7         2015-03-09 22:53:13
8         2015-03-09 22:53:21
9         2015-03-09 23:03:48
10        2015-03-09 22:50:34
11        2015-03-09 22:49:57
12        2015-03-09 23:00:05
```

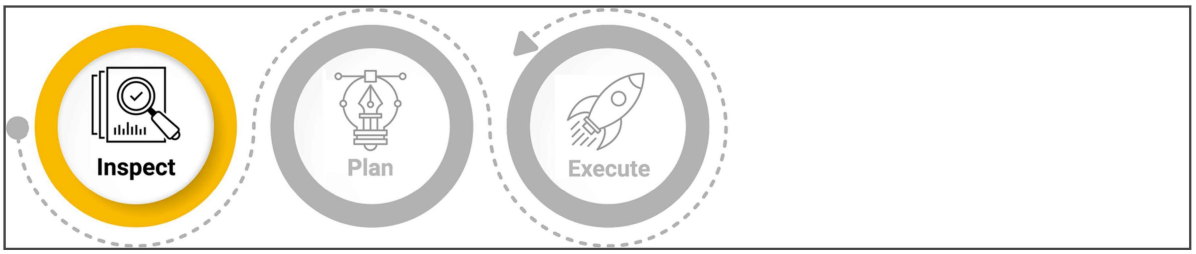
```
13          2015-03-09 22:48:33
14          2015-03-09 23:00:07
15          2015-03-09 22:51:42
16          2015-03-09 22:51:04
17          2015-03-09 23:02:19
18          2015-03-09 23:11:39
19          2015-03-09 23:02:13
```



These dates don't seem outlandish—the years are within expected bounds, and there appears to be some consistency from one entry to the next. Since the output looks reasonable, assign it to the timestamp column.




```
ratings['timestamp'] = pd.to_datetime(ratings['timestamp'], unit='s')
```



Finally, we'll look at the statistics of the actual ratings and see if there are any glaring errors. A quick, easy way to do this is to look at a histogram of the rating

distributions, and then use the `describe()` method to print out some stats on central tendency and spread.

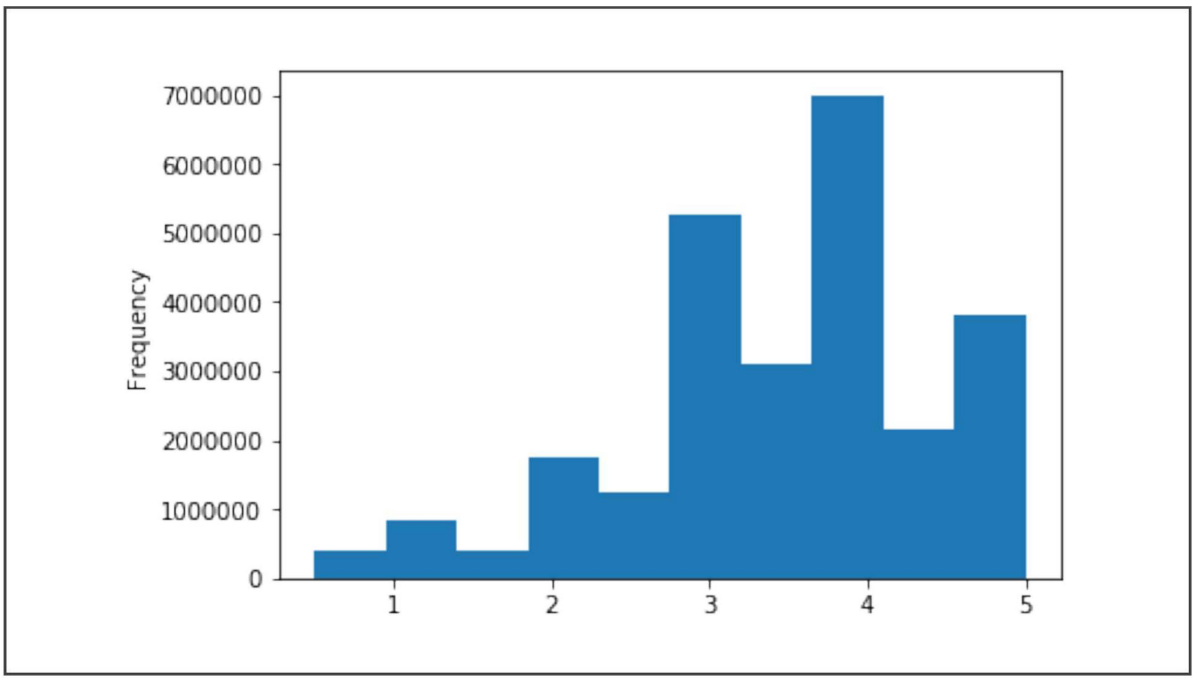
> **Note**
>
> A **histogram** is a bar chart that displays how often a data point shows up in the data. A histogram is a quick, visual way to get a sense of how a dataset is distributed.

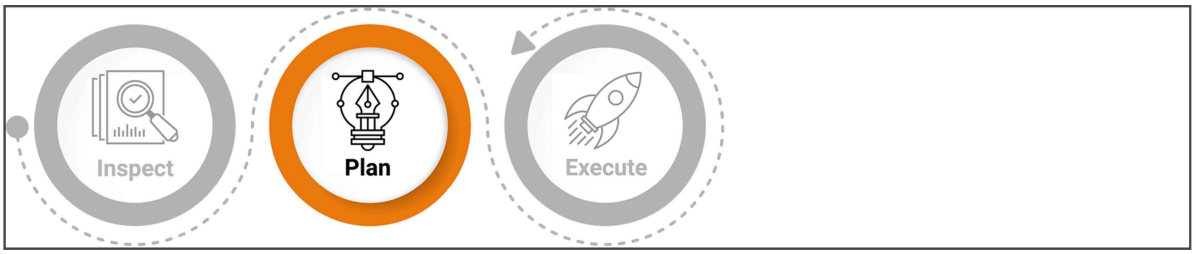
Your code should look like this:

```
ratings['rating'].plot(kind='hist')
ratings['rating'].describe()
```

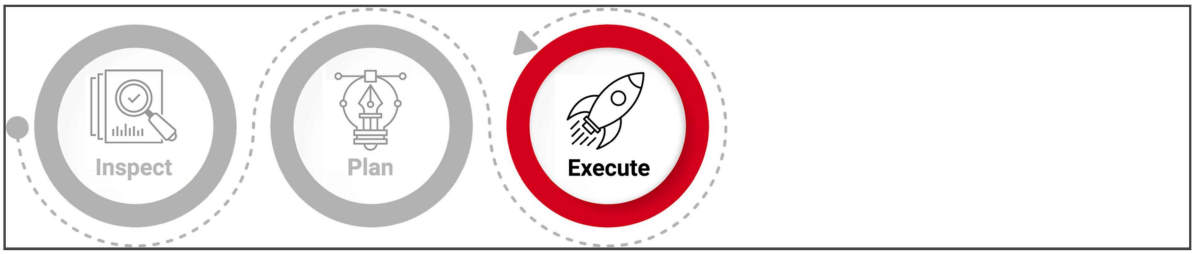
The output should look like the following.

```
count    2.602429e+07
mean     3.528090e+00
std      1.065443e+00
min      5.000000e-01
25%      3.000000e+00
50%      3.500000e+00
75%      4.000000e+00
max      5.000000e+00
Name: rating, dtype: float64
```





That seems to make sense. People are more likely to give whole number ratings than half, which explains the spikes in the histogram. The median score is 3.5, the mean is 3.53, and all the ratings are between 0 and 5.



The ratings dataset looks good to go, which means we're done with the first half of the Transform step. Let's get ready to finish it.

> **Add, Commit, Push**
> Remember to add, commit, and push your work!

8.4.1: Merge Wikipedia and Kaggle Metadata

Now that the Wikipedia data and Kaggle data are cleaned up and in tabular formats with the right data types for each column, Britta can join them together. However, after they're joined, the data still needs to be cleaned up a bit, especially where Kaggle and Wikipedia data overlap.

With all the tables cleaned up, we're ready to merge them by IMDb ID.

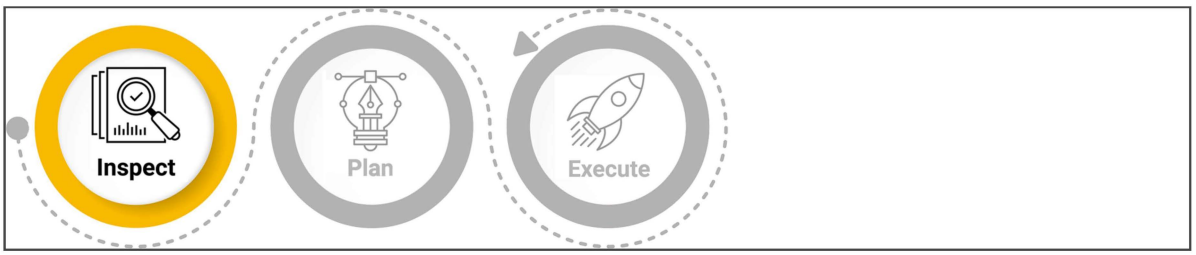
You only want movies that are in both tables, so what is the appropriate join to use?

- Inner join
- Left join
- Right join
- Outer join

Check Answer

Finish ▸

One of the things we always want to look out for after we've merged data is redundant columns.

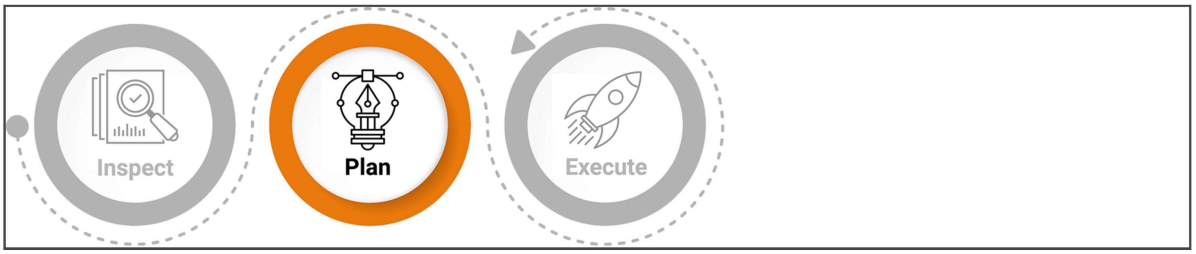


Print out a list of the columns so we can identify which ones are redundant. We'll use the `suffixes` parameter to make it easier to identify which table each column came from. Here's what your code should look like:

```
movies_df = pd.merge(wiki_movies_df, kaggle_metadata, on='imdb_id', suffi
```

There are seven pairs of columns that have redundant information. We'll look at each pair of columns and decide how to handle the data.

There are a few options when dealing with redundant data. We'll consider two. The simplest is to just drop one of the competing columns, but sometimes that means a loss of good information. Sometimes, one column will have data where the other has missing data, and vice versa. In that case, we'd want the other option: fill in the gaps using both columns.



Below is the list of competing columns. We'll fill in the resolution to each pair as we go along. We'll hold off on implementing the resolutions until we make a decision for each pair because if we did, we might inadvertently remove data that could be helpful in making a later decision.

Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	
running_time	runtime	
budget_wiki	budget_kaggle	
box_office	revenue	
release_date_wiki	release_date_kaggle	
Language	original_language	
Production company(s)	production_companies	

You may find it helpful to include a table like this in your Jupyter Notebook that documents the decisions made and the justifications for them. Unfortunately, markdown doesn't support formatting tables. One way to work around that is to just write your text down as comments in a code cell.

```
# Competing data:
# Wiki                     Movielens               Resolution
#--------------------------------------------------------------------------
# title_wiki               title_kaggle
# running_time             runtime

# budget_wiki              budget_kaggle
# box_office               revenue
# release_date_wiki        release_date_kaggle
# Language                 original_language
# Production company(s)    production_companies
```

Let's start comparing columns.

Title

First, just take a quick look at some of the titles.

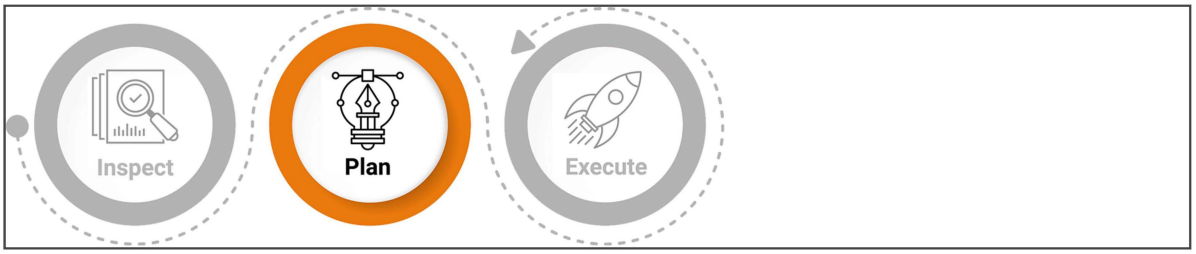
```
movies_df[['title_wiki','title_kaggle']]
```

They both seem pretty consistent, which we'd expect. Look at the rows where the titles don't match.

```
movies_df[movies_df['title_wiki'] != movies_df['title_kaggle']][['title_w
```

Both options look pretty good, but the Kaggle data looks just a little bit more consistent. Let's confirm there aren't any missing titles in the Kaggle data with the following code:

```
movies_df[(movies_df['title_kaggle'] == '') | (movies_df['title_kaggle'].
```

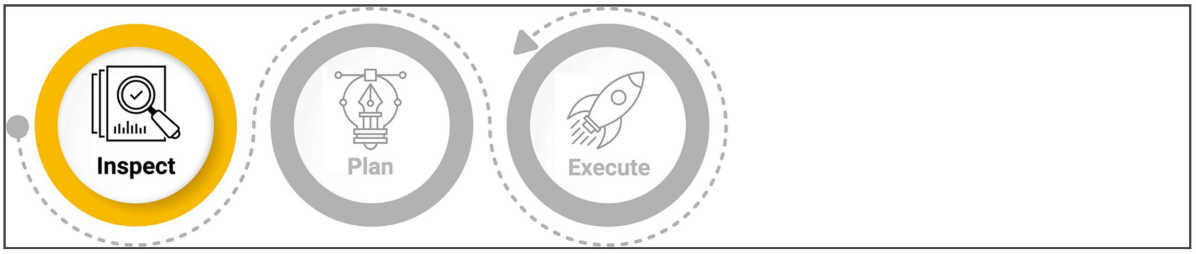


No results were returned, so we can just drop the Wikipedia titles.

Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	**Drop Wikipedia.**
running_time	runtime	

budget_wiki	budget_kaggle	
box_office	revenue	
release_date_wiki	release_date_kaggle	
Language	original_language	
Production company(s)	production_companies	

Runtime

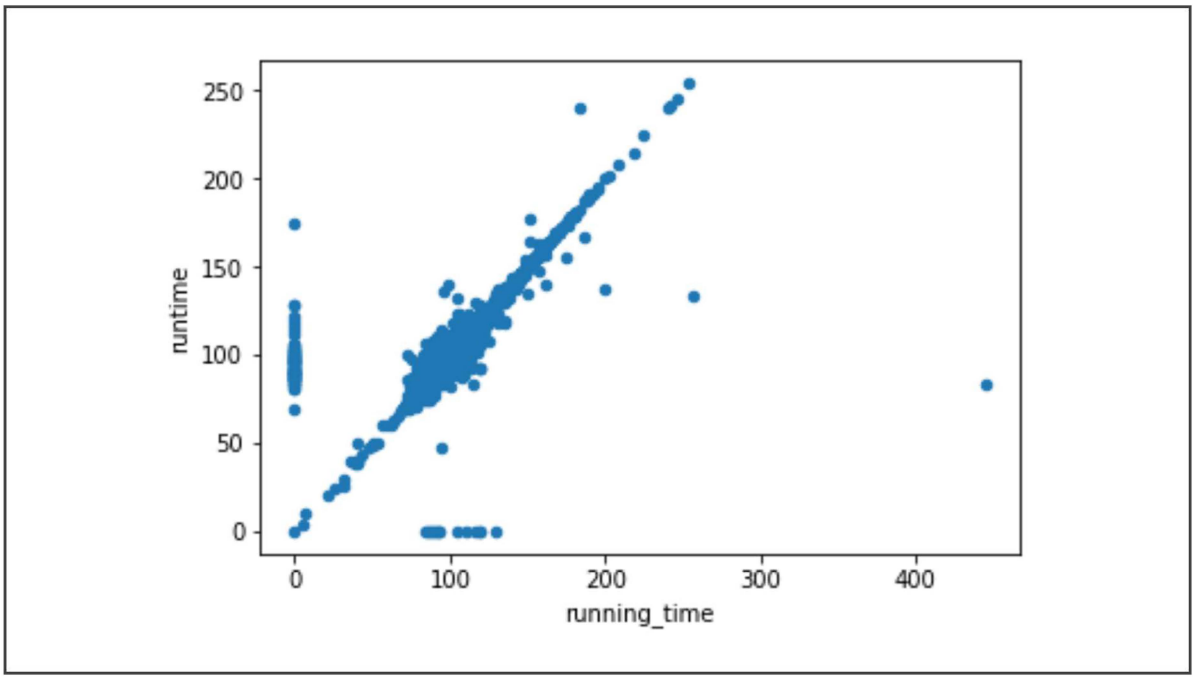


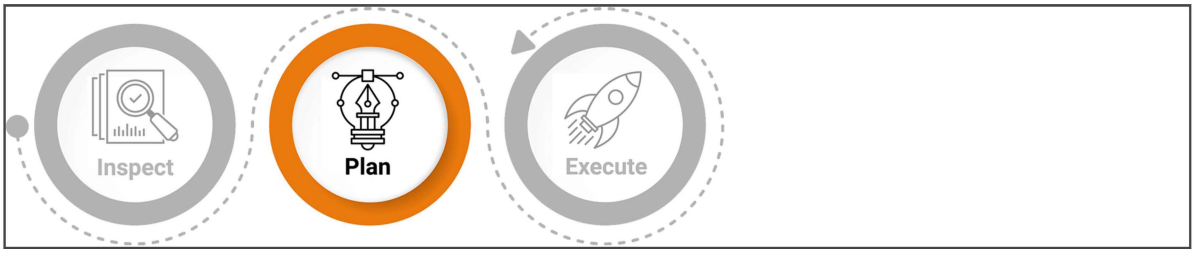
Next, look at `running_time` versus `runtime`. A scatter plot is a great way to give us a sense of how similar the columns are to each other. If the two columns were exactly the same, we'd see a scatter plot of a perfectly straight line. Any wildly different values will show up as dots far from that central line, and if one column is missing data, those values will fall on the x-axis or y-axis.

> **Caution**
> Because we're dealing with merged data, we should expect there to be missing values. Scatter plots won't show null values, so we need to fill them in with zeros when we're making our plots to get the whole picture.

The following code will fill in missing values with zero and make the scatter plot:

```
movies_df.fillna(0).plot(x='running_time', y='runtime', kind='scatter')
```



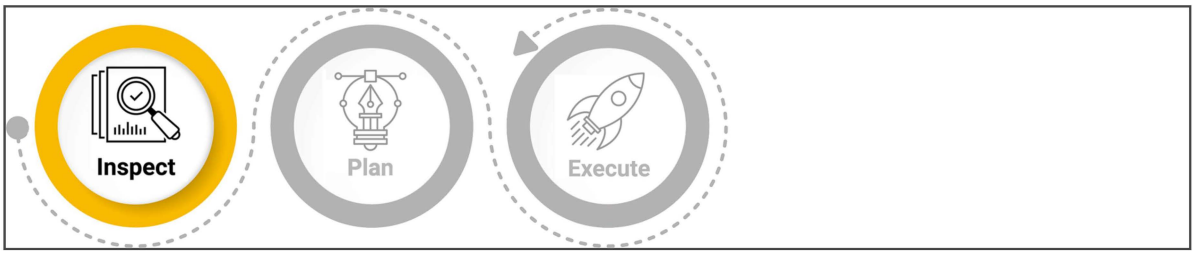


Most of the runtimes are pretty close to each other, but the Wikipedia data has some outliers, so the Kaggle data is probably a better choice here. However, we can also see from the scatter plot that there are movies where Kaggle has 0 for the runtime but Wikipedia has data, so we'll fill in the gaps with Wikipedia data.

Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	Drop Wikipedia.
running_time	runtime	**Keep Kaggle; fill in zeros with Wikipedia**

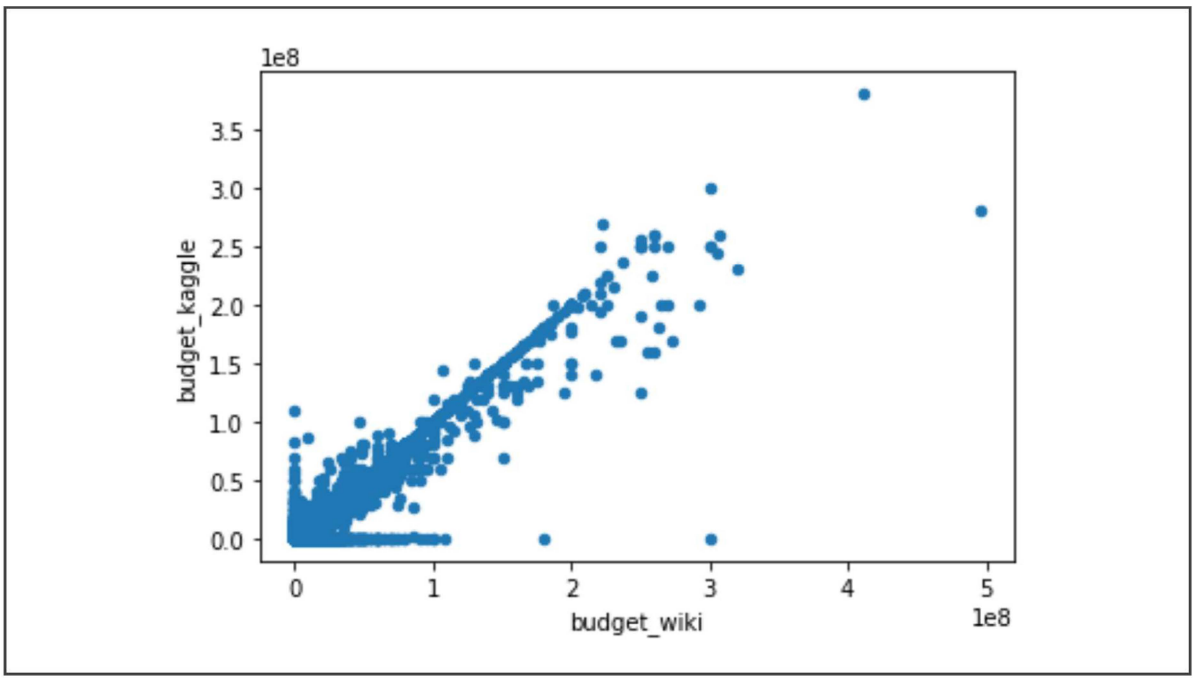
		data.
budget_wiki	budget_kaggle	
box_office	revenue	
release_date_wiki	release_date_kaggle	
Language	original_language	
Production company(s)	production_companies	

Budget



Since budget_wiki and budget_kaggle are numeric, we'll make another scatter plot to compare the values:

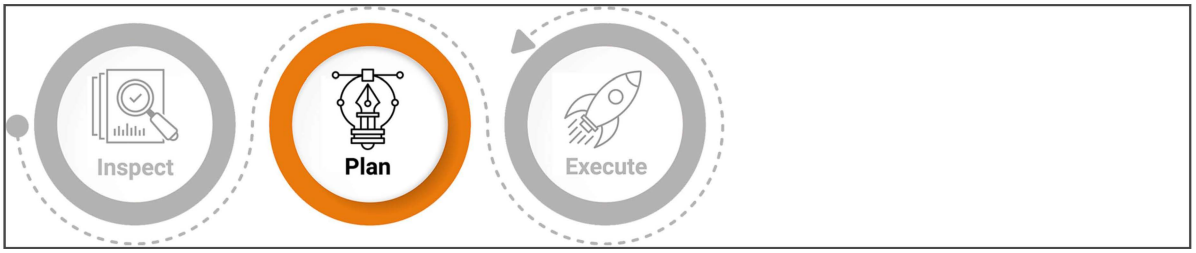
```
movies_df.fillna(0).plot(x='budget_wiki',y='budget_kaggle', kind='scatter
```



Pause

Here are some questions to consider when interpreting this scatter plot:

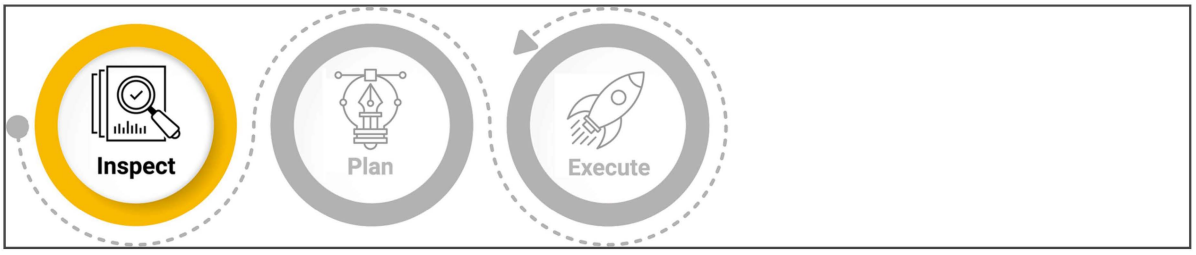
- Which dataset seems to have more outliers?
- Which dataset seems to have more missing data points?
- If we were to fill in the missing data points of one set with the other, which would be more likely to give us consistent data?
- Is it better to start with a base of consistent data and fill in missing points with possible outliers? Or is it better to start with a base of data with outliers and fill in missing points with more consistent data?



The Wikipedia data appears to have more outliers compared to the Kaggle data. However, there are quite a few movies with no data in the Kaggle column, while Wikipedia does have budget data. Therefore, we'll fill in the gaps with Wikipedia's data.

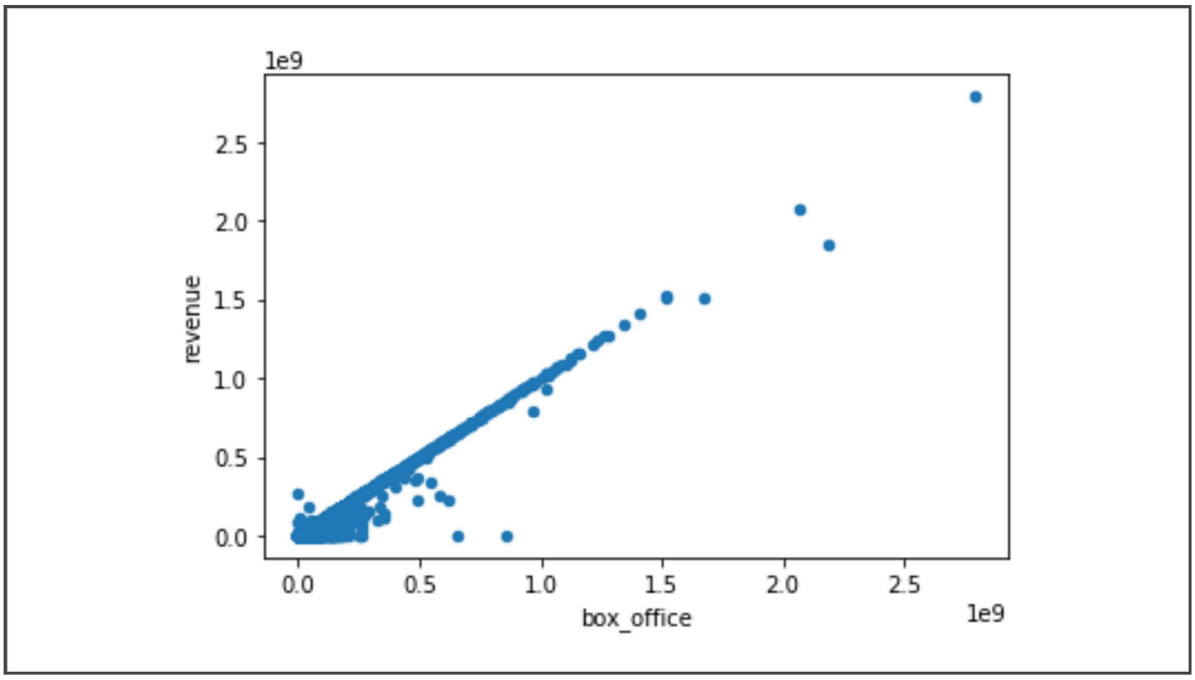
Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	Drop Wikipedia.
running_time	runtime	Keep Kaggle; fill in zeros with Wikipedia data.
budget_wiki	budget_kaggle	**Keep Kaggle; fill in zeros with Wikipedia data.**
box_office	revenue	
release_date_wiki	release_date_kaggle	
Language	original_language	
Production company(s)	production_companies	

Box Office



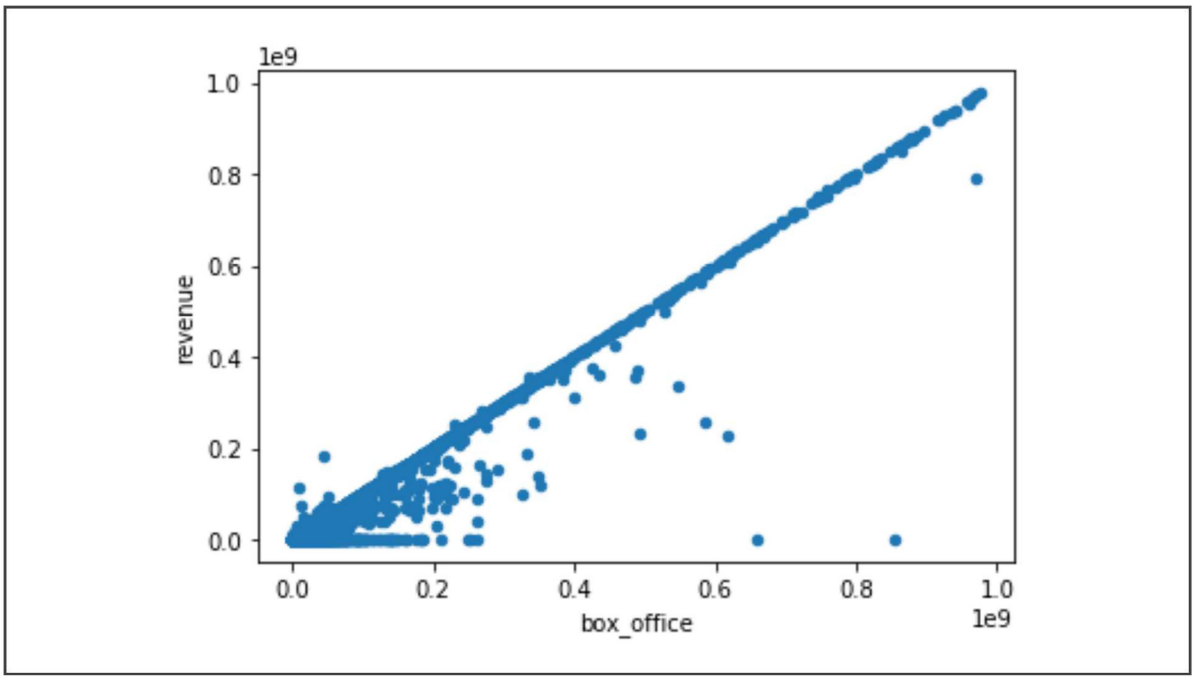
The box_office and revenue columns are numeric, so we'll make another scatter plot.

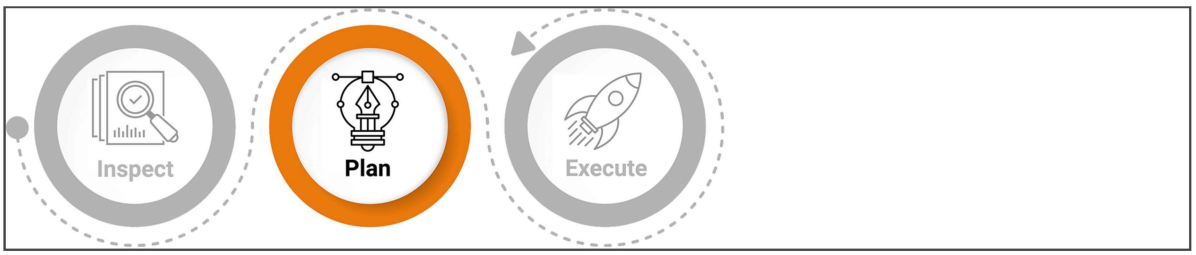
```
movies_df.fillna(0).plot(x='box_office', y='revenue', kind='scatter')
```



That looks pretty close, but we might be getting thrown off by the scale of that large data point. Let's look at the scatter plot for everything less than $1 billion in box_office.

```
movies_df.fillna(0)[movies_df['box_office'] < 10**9].plot(x='box_office',
```

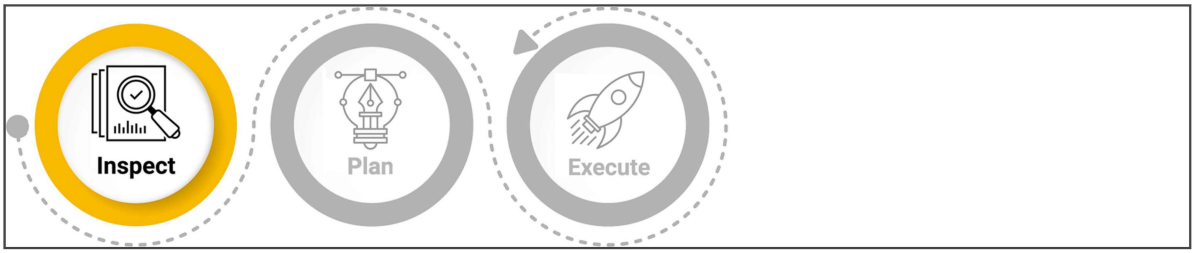




This looks similar to what we've seen for budget, so we'll make the same decision: keep the Kaggle data, but fill in the zeros with Wikipedia data.

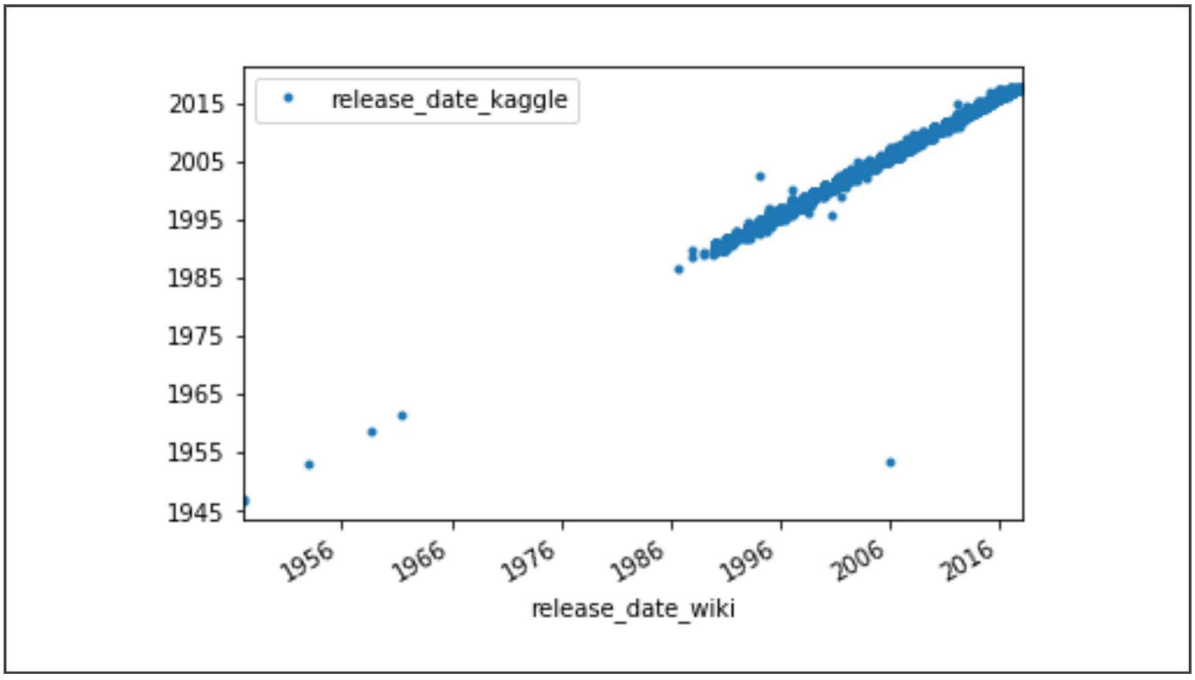
Wikipedia	**Kaggle**	**Resolution**
title_wiki	title_kaggle	Drop Wikipedia.
running_time	runtime	Keep Kaggle; fill in zeros with Wikipedia data.
budget_wiki	budget_kaggle	Keep Kaggle; fill in zeros with Wikipedia data.
box_office	revenue	**Keep Kaggle; fill in zeros with Wikipedia data.**
release_date_wiki	release_date_kaggle	
Language	original_language	
Production company(s)	production_companies	

Release Date



For release_date_wiki and release_date_kaggle, we can't directly make a scatter plot, because the scatter plot only works on numeric data. However, there's a tricky workaround that we can use. We'll use the regular line plot (which can plot date data), and change the style to only put dots by adding `style='.'` to the `plot()` method:

```
movies_df[['release_date_wiki','release_date_kaggle']].plot(x='release_da
```



We should investigate that wild outlier around 2006. We're just going to choose some rough cutoff dates to single out that one movie. We'll look for any movie

whose release date according to Wikipedia is after 1996, but whose release date according to Kaggle is before 1965. Here's what your code should look like:

```
movies_df[(movies_df['release_date_wiki'] > '1996-01-01') & (movies_df['r
```

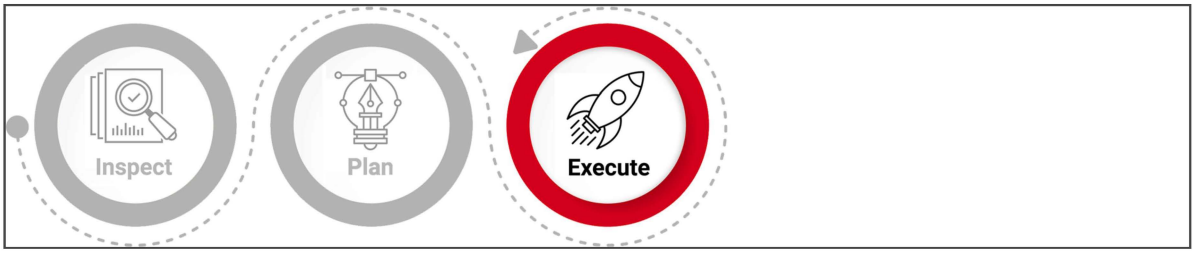
	url	year	imdb_link	title_wiki	Based on	Starring	Cinematography	Release date	Country	Languag
3607	https://en.wikipedia.org/wiki/The_Holiday	2006	https://www.imdb.com/title/tt00457939/	The Holiday	NaN	[Kate Winslet, Cameron Diaz, Jude Law, Jack Bl...	Dean Cundey	[December 8, 2006, (, 2006-12-08,)]	United States	Englis

1 rows × 45 columns

	Country	Language	...	release_date_kaggle	revenue	runtime	spoken_languages	status	tagline	title_kaggle	video	vote_average	vote_count
r -]	United States	English	...	1953-08-28	30500000.0	118.0	[{'iso_639_1': 'en', 'name': 'English'}]	Released	Pouring out of impassioned pages...brawling th...	From Here to Eternity	False	7.2	137.0

Based on the output, it looks like somehow *The Holiday* in the Wikipedia data got merged with *From Here to Eternity*. We'll have to drop that row from our DataFrame. We'll get the index of that row with the following:

```
movies_df[(movies_df['release_date_wiki'] > '1996-01-01') & (movies_df['r
```



Then we can drop that row like this:

```
movies_df = movies_df.drop(movies_df[(movies_df['release_date_wiki'] > '1
```

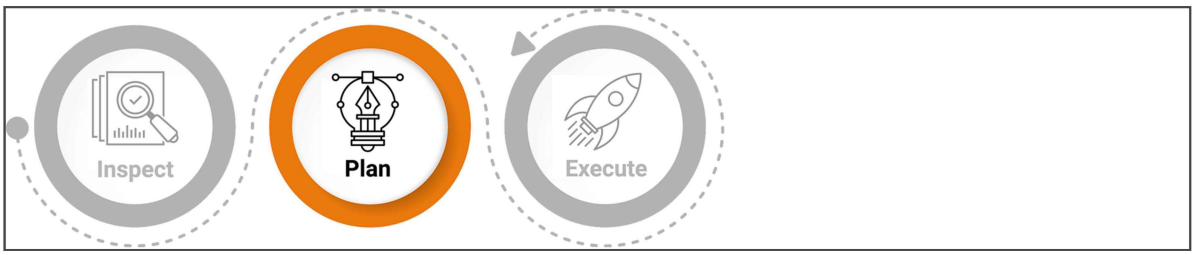
Now, see if there are any null values:

```
movies_df[movies_df['release_date_wiki'].isnull()]
```

The Wikipedia data is missing release dates for 11 movies:

	url	year	imdb_link	title_wiki	Based on	Starring	Cinematography	Release date	Country
1008	https://en.wikipedia.org/wiki/Black_Scorpion_(...	1995	https://www.imdb.com/title/tt0112519/	Black Scorpion	NaN	[Joan Severance, Bruce Abbott, Garrett Morris]	Geoff George	NaN	United States
1061	https://en.wikipedia.org/wiki/Flirt_(1995_film)	1995	https://www.imdb.com/title/tt0113080/	Flirt	NaN	[Bill Sage, Dwight Ewell, Miho Nikaido]	Michael Spiller	NaN	[United States, Germany, Japan, [1]]
1121	https://en.wikipedia.org/wiki/Let_It_Be_Me_(19...	1995	https://www.imdb.com/title/tt0113638/	Let It Be Me	NaN	NaN	Miroslav Ondrícek	NaN	NaN
1564	https://en.wikipedia.org/wiki/A_Brooklyn_State...	1997	https://www.imdb.com/title/tt0118782/	A Brooklyn State of Mind	NaN	NaN	Ken Kelsch	NaN	NaN
1633	https://en.wikipedia.org/wiki/Highball_(film)	1997	https://www.imdb.com/title/tt0119291/	Highball	NaN	[Justine Bateman, Peter Bogdanovich, Chris Eig...	Steven Bernstein	NaN	United States
1775	https://en.wikipedia.org/wiki/Velocity_Trap	1997	https://www.imdb.com/title/tt0120435/	Velocity Trap	NaN	[Olivier Gruner, Alicia Coppola, Ken Olandt]	Philip D. Schwartz	NaN	United States
2386	https://en.wikipedia.org/wiki/The_Visit_(2000_...	2000	https://www.imdb.com/title/tt0199129/	The Visit	NaN	[Hill Harper, Billy Dee Williams, Obba Babatun...	John L. Demps Jr.	NaN	NaN
2786	https://en.wikipedia.org/wiki/Stevie_(2002_film)	2002	https://www.imdb.com/title/tt0334416/	Stevie	NaN	NaN	[Dana Kupper, Gordon Quinn, Peter Gilbert]	NaN	United States
3174	https://en.wikipedia.org/wiki/Return_to_Sender...	2004	https://www.imdb.com/title/tt0396190/	Return to Sender	NaN	[Aidan Quinn, Connie Nielsen, Mark Holton]	NaN	NaN	[Denmark, USA, UK]
3651	https://en.wikipedia.org/wiki/Live_Free_or_Die...	2006	https://www.imdb.com/title/tt0432318/	Live Free or Die	NaN	[Aaron Stanford, Paul Schneider, Ebon Moss-Bac...	NaN	NaN	United States
4967	https://en.wikipedia.org/wiki/For_the_Love_of_...	2012	https://www.imdb.com/title/tt1730294/	For the Love of Money	NaN	[Yehuda Levi, Edward Furlong, James Caan, Jeff...	Andrzej Sekuła	NaN	United States

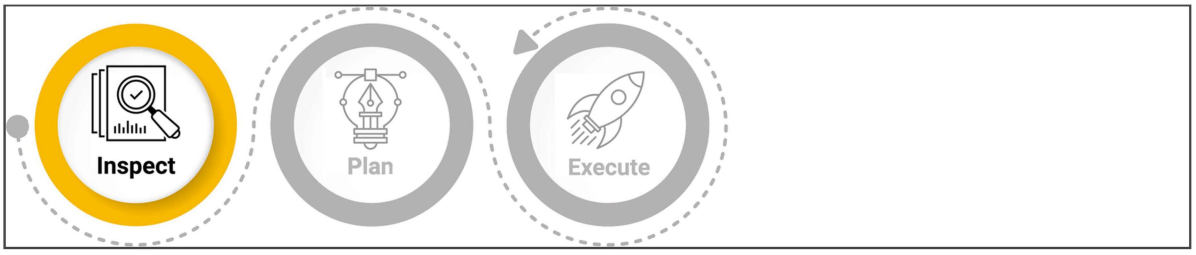
11 rows × 45 columns



But the Kaggle data isn't missing any release dates. In this case, we'll just drop the Wikipedia data.

Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	Drop Wikipedia.
running_time	runtime	Keep Kaggle; fill in zeros with Wikipedia data.
budget_wiki	budget_kaggle	Keep Kaggle; fill in zeros with Wikipedia data.
box_office	revenue	Keep Kaggle; fill in zeros with Wikipedia data.
release_date_wiki	release_date_kaggle	**Drop Wikipedia.**
Language	original_language	
Production company(s)	production_companies	

Language



For the language data, we'll compare the value counts of each. However, consider the following code:

```
movies_df['Language'].value_counts()
```

This code throws an error because some of the language data points are stored as lists.

```
TypeError: unhashable type: 'list'
```

> **Note**
> We don't need to worry about what hashing is right now, but if you're curious, **hashing** is a clever computer science trick that can be used to speed up algorithms like getting value counts. Hashing converts values, even arbitrarily long strings, to a limited space of numerical values. We'll talk about hashing more when we get to machine learning, but for now, the important part is that Python creates hash values when new objects are created if they are immutable. Since mutable objects can have their values change after being created, the values might change and not match the hash, so Python just refuses.

We need to convert the lists in `Language` to tuples so that the `value_counts()` method will work. See the following code:

```
movies_df['Language'].apply(lambda x: tuple(x) if type(x) == list else x)
```

Here's what the output will look like.

```
English                                          5479
NaN                                               134
(English, Spanish)                                 68
(English, French)                                  35
(English, Japanese)                                25
                                                  ...
(English, German, Russian, Ukrainian)               1
```

```
(English, Lao)                                          1
(English, Italian, Swedish)                             1
(English, Afrikaans, German)                            1
(English, French, Hebrew, Spanish, Arabic, Italian)     1
Name: Language, Length: 198, dtype: int64
```

For the Kaggle data, there are no lists, so we can just run `value_counts()` on it.

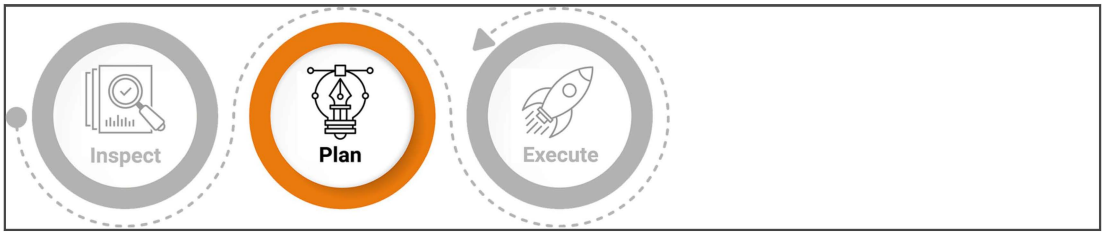
```
movies_df['original_language'].value_counts(dropna=False)
```

Here's what the output will look like.

```
en    5982
fr      16
es       9
it       8
de       6
zh       4
ja       4
pt       4
da       2
hi       2
tr       1
cn       1
ar       1
ab       1
sv       1
ko       1
he       1
Name: original_language, dtype: int64
```

> **Pause**
>
> There's a trade-off here between the Wikipedia language data and the Kaggle language data. While the Wikipedia data has more information about multiple languages, the Kaggle data is already in a consistent and usable

format. Parsing the Wikipedia data may create too many difficulties to make it worthwhile, though.



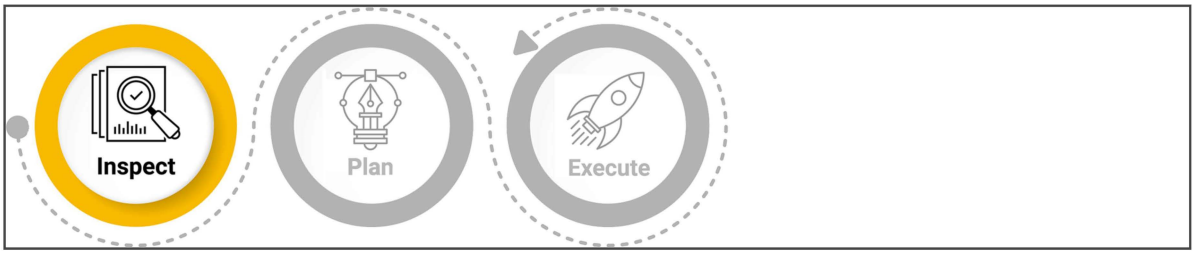
This is another judgment call; there's no clear-cut answer here. However, for better or for worse, decisions that save time are usually the ones that win, so we'll use the Kaggle data here.

Here's our updated plan:

Wikipedia	**Kaggle**	**Resolution**
title_wiki	title_kaggle	Drop Wikipedia.
running_time	runtime	Keep Kaggle; fill in zeros with Wikipedia data.
budget_wiki	budget_kaggle	Keep Kaggle; fill in zeros with Wikipedia data.
box_office	revenue	Keep Kaggle; fill in zeros with Wikipedia data.
release_date_wiki	release_date_kaggle	Drop Wikipedia.
Language	original_language	**Drop Wikipedia.**

Production company(s)	production_companies

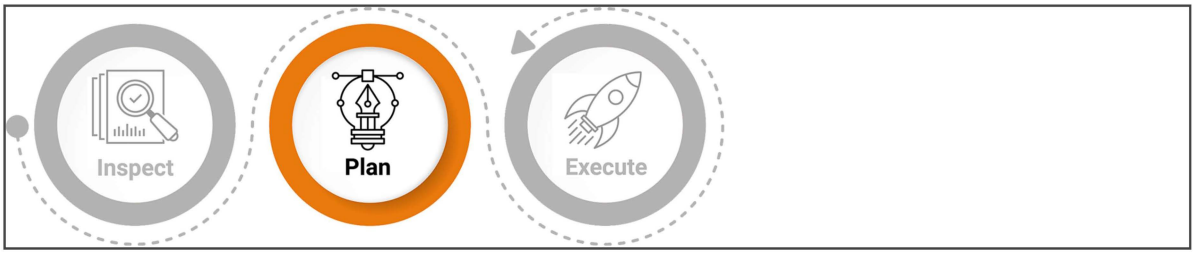
Production Companies



Again, we'll start off just taking a look at a small number of samples.

```
movies_df[['Production company(s)','production_companies']]
```

The Kaggle data is much more consistent, and it would be difficult, if not impossible, to translate the Wikipedia data into the same format.

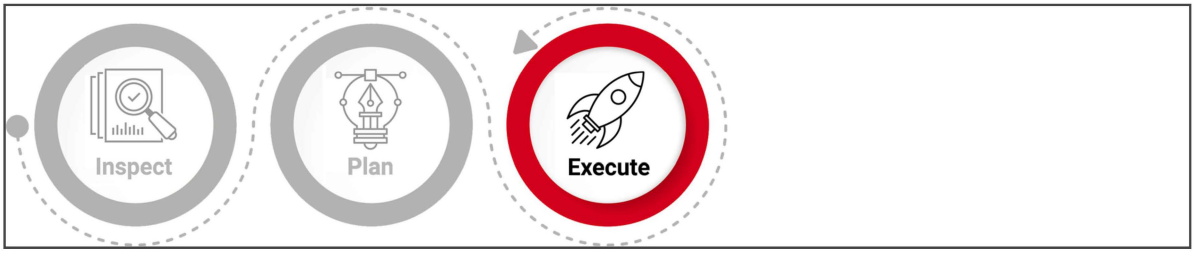


We'll drop the Wikipedia data in this case.

Wikipedia	Kaggle	Resolution
title_wiki	title_kaggle	Drop Wikipedia.

running_time	runtime	Keep Kaggle; fill in zeros with Wikipedia data.
budget_wiki	budget_kaggle	Keep Kaggle; fill in zeros with Wikipedia data.
box_office	revenue	Keep Kaggle; fill in zeros with Wikipedia data.
release_date_wiki	release_date_kaggle	Drop Wikipedia.
Language	original_language	Drop Wikipedia.
Production company(s)	production_companies	**Drop Wikipedia.**

Put It All Together



First, we'll drop the title_wiki, release_date_wiki, Language, and Production company(s) columns.

```
movies_df.drop(columns=['title_wiki','release_date_wiki','Language','Proc
```

Next, to save a little time, we'll make a function that fills in missing data for a column pair and then drops the redundant column.

```
def fill_missing_kaggle_data(df, kaggle_column, wiki_column):
    df[kaggle_column] = df.apply(
        lambda row: row[wiki_column] if row[kaggle_column] == 0 else row[
        , axis=1)
    df.drop(columns=wiki_column, inplace=True)
```

Now we can run the function for the three column pairs that we decided to fill in zeros.

```
fill_missing_kaggle_data(movies_df, 'runtime', 'running_time')
fill_missing_kaggle_data(movies_df, 'budget_kaggle', 'budget_wiki')
fill_missing_kaggle_data(movies_df, 'revenue', 'box_office')
movies_df
```

Since we've merged our data and filled in values, it's good to check that there aren't any columns with only one value, since that doesn't really provide any information. Don't forget, we need to convert lists to tuples for `value_counts()` to work.

```
for col in movies_df.columns:
    lists_to_tuples = lambda x: tuple(x) if type(x) == list else x

    value_counts = movies_df[col].apply(lists_to_tuples).value_counts(dro
    num_values = len(value_counts)
    if num_values == 1:
        print(col)
```

Running this, we see that `'video'` only has one value:

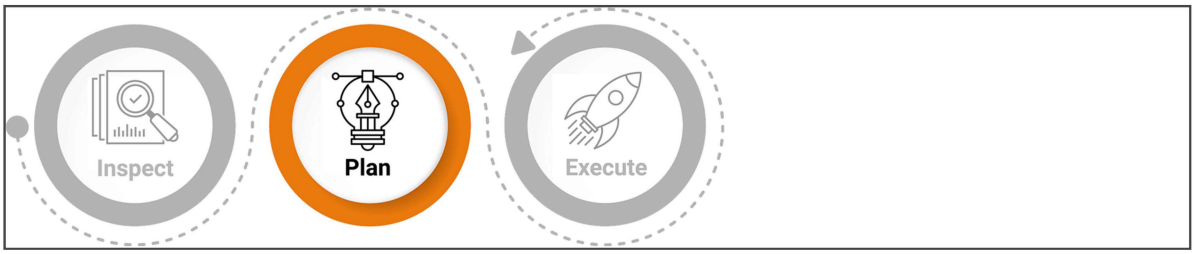
```
movies_df['video'].value_counts(dropna=False)
```

Here's what the output will look like.

```
False    6044
Name: video, dtype: int64
```

Since it's false for every row, we don't need to include this column.

> **Skill Drill**
> How could you replace the previous `for` loop with a list comprehension?



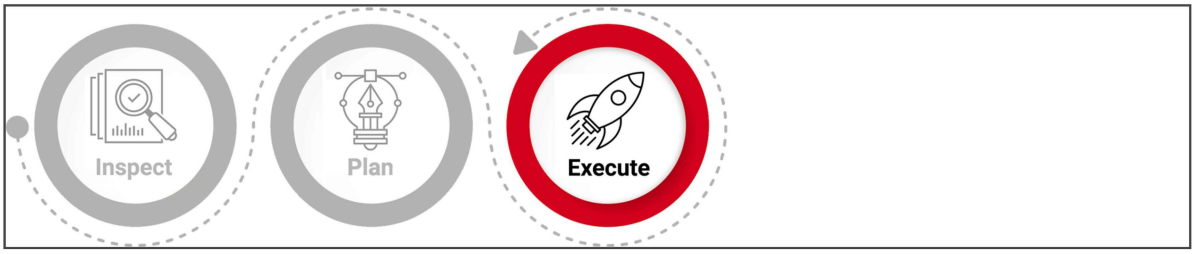
We should reorder the columns to make the dataset easier to read for the hackathon participants. Having similar columns near each other helps people looking through the data get a better sense of what information is available. One way to reorder them would be to consider the columns roughly in groups, like this:

1. Identifying information (IDs, titles, URLs, etc.)
2. Quantitative facts (runtime, budget, revenue, etc.)
3. Qualitative facts (genres, languages, country, etc.)
4. Business data (production companies, distributors, etc.)
5. People (producers, director, cast, writers, etc.)

The following code is one way to reorder the columns:

```
movies_df = movies_df[['imdb_id','id','title_kaggle','original_title','ta
                       'runtime','budget_kaggle','revenue','release_date_
                       'genres','original_language','overview','spoken_la
                       'production_companies','production_countries','Dis
                       'Producer(s)','Director','Starring','Cinematograph
```

```
                    Producer(s)', 'Director', 'Starring', 'Cinematograph
                ]]
```



Finally, we need to rename the columns to be consistent.

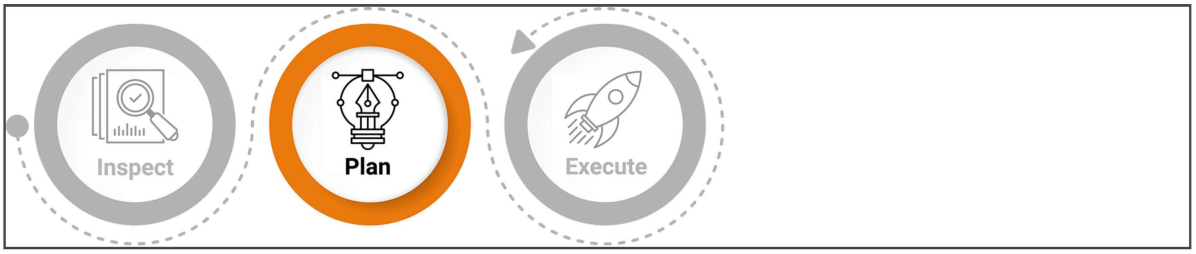
```
movies_df.rename({'id':'kaggle_id',
                  'title_kaggle':'title',
                  'url':'wikipedia_url',
                  'budget_kaggle':'budget',
                  'release_date_kaggle':'release_date',
                  'Country':'country',
                  'Distributor':'distributor',
                  'Producer(s)':'producers',
                  'Director':'director',
                  'Starring':'starring',
                  'Cinematography':'cinematography',
                  'Editor(s)':'editors',
                  'Writer(s)':'writers',
                  'Composer(s)':'composers',
                  'Based on':'based_on'
                 }, axis='columns', inplace=True)
```

Your first merge is done! We got the tough one out of the way first, and now we're almost done.

> **Add, Commit, Push**
> Remember to add, commit, and push your work!

8.4.2: Transform and Merge Rating Data

Britta wants to include the rating data with the movie data, but it's a very large dataset. She wants to reduce the ratings data to a useful summary of rating information for each movie, and then make the full dataset available to the hackathon participants if they decide they need more granular rating information.



For each movie, Britta wants to include the rating data, but the rating dataset has so much information that it's too unwieldy to use all of it. We could calculate some basic statistics like the mean and median rating for each movie, but a more useful summary is just to count how many times a movie received a given rating. This way, someone who wants to calculate statistics for the dataset would have all the information they need.

We'll include the raw ratings data if the hackathon participants want to do more in-depth analysis, such as comparing across users, but having the rating counts for each movie is easy enough to do. Plus, it will enable the hackathon participants to calculate statistics on their own without having to work with a dataset containing 26-million rows.

First, we need to use a `groupby` on the “movieId” and “rating” columns and take the count for each group.

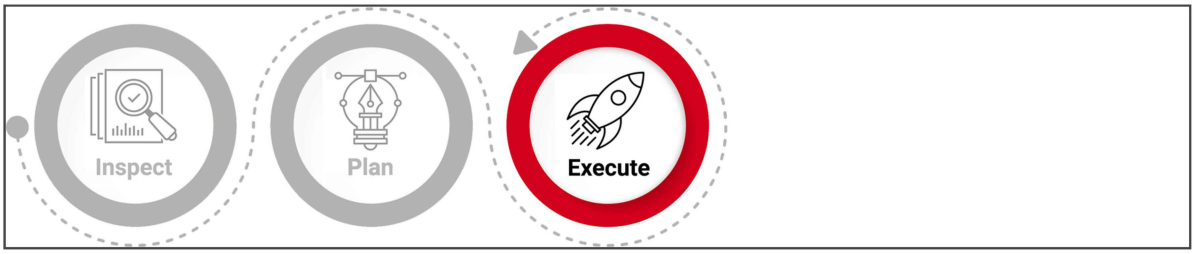
```
rating_counts = ratings.groupby(['movieId','rating'], as_index=False).cou
```

Then we’ll rename the “userId” column to “count.”

> **Note**
> The choice of renaming “userId” to “count” is arbitrary. Both “userId” and “timestamp” have the same information, so we could use either one.

Your code should look like the following:

```
rating_counts = ratings.groupby(['movieId','rating'], as_index=False).cou
                .rename({'userId':'count'}, axis=1)
```



Now the magical part. We can pivot this data so that `movieId` is the index, the columns will be all the rating values, and the rows will be the counts for each rating value.

```
rating_counts = ratings.groupby(['movieId','rating'], as_index=False).cou
                .rename({'userId':'count'}, axis=1) \
                .pivot(index='movieId',columns='rating', values='count')
```

We want to rename the columns so they're easier to understand. We'll prepend `rating_` to each column with a list comprehension:

```
rating_counts.columns = ['rating_' + str(col) for col in rating_counts.co
```

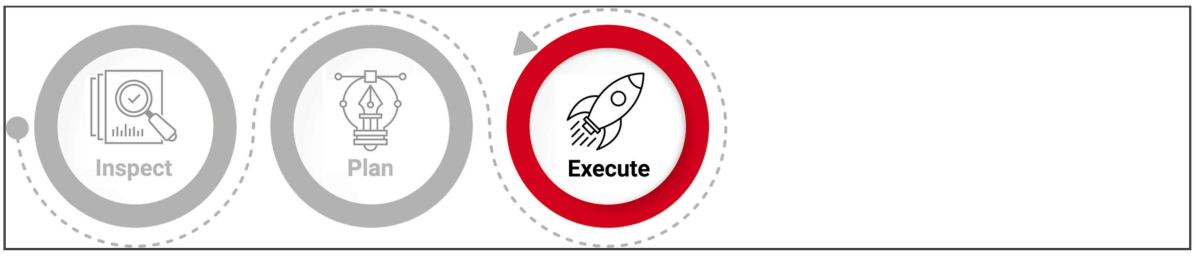
Now we can merge the rating counts into `movies_df`.

What kind of merge do we want to use to join ratings counts onto `movies_df`? (Assume `movies_df` is the left DataFrame and `rating_counts` is the right DataFrame.)

- Inner merge
- Left merge
- Right merge
- Outer merge

Check Answer

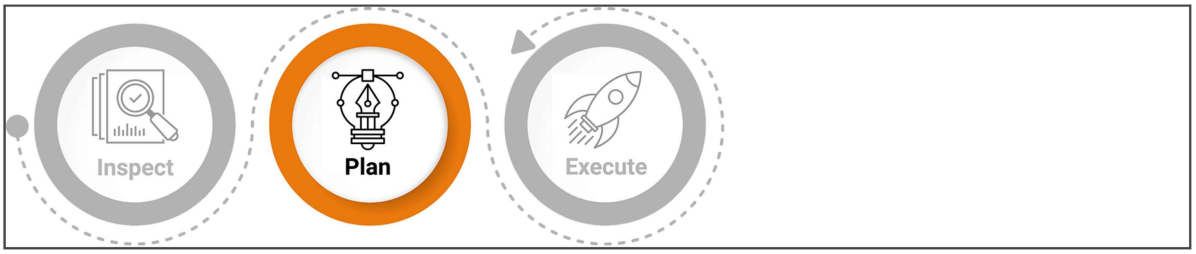
Finish ▸



This time, we need to use a left merge, since we want to keep everything in

`movies_df`:

```
movies_with_ratings_df = pd.merge(movies_df, rating_counts, left_on='kagg
```



Finally, because not every movie got a rating for each rating level, there will be missing values instead of zeros. We have to fill those in ourselves, like this:

```
movies_with_ratings_df[rating_counts.columns] = movies_with_ratings_df[ra
```

And we're done—we just finished the Transform step in ETL! Now all that's left is loading our tables into SQL.

> **Add, Commit, Push**
>
> Remember to add, commit, and push your work!

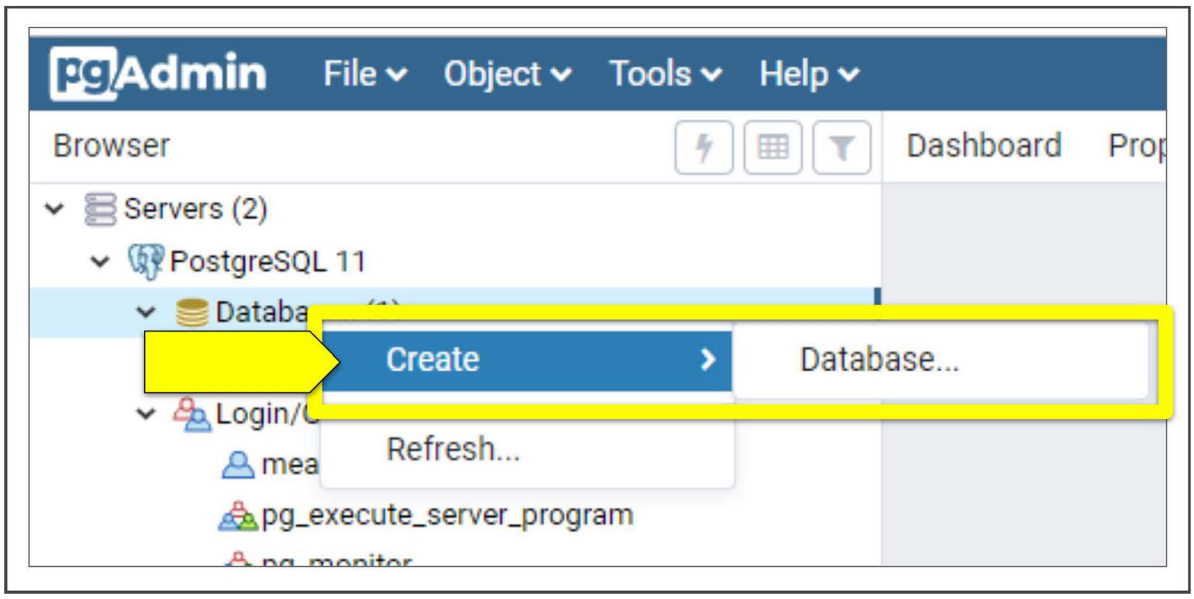
8.5.1: Connect Pandas and SQL

Amazing Prime has decided the easiest way to make the data accessible for the hackathon is to provide a SQL database to the participants. Britta needs to move the data from Pandas into a PostgreSQL database.

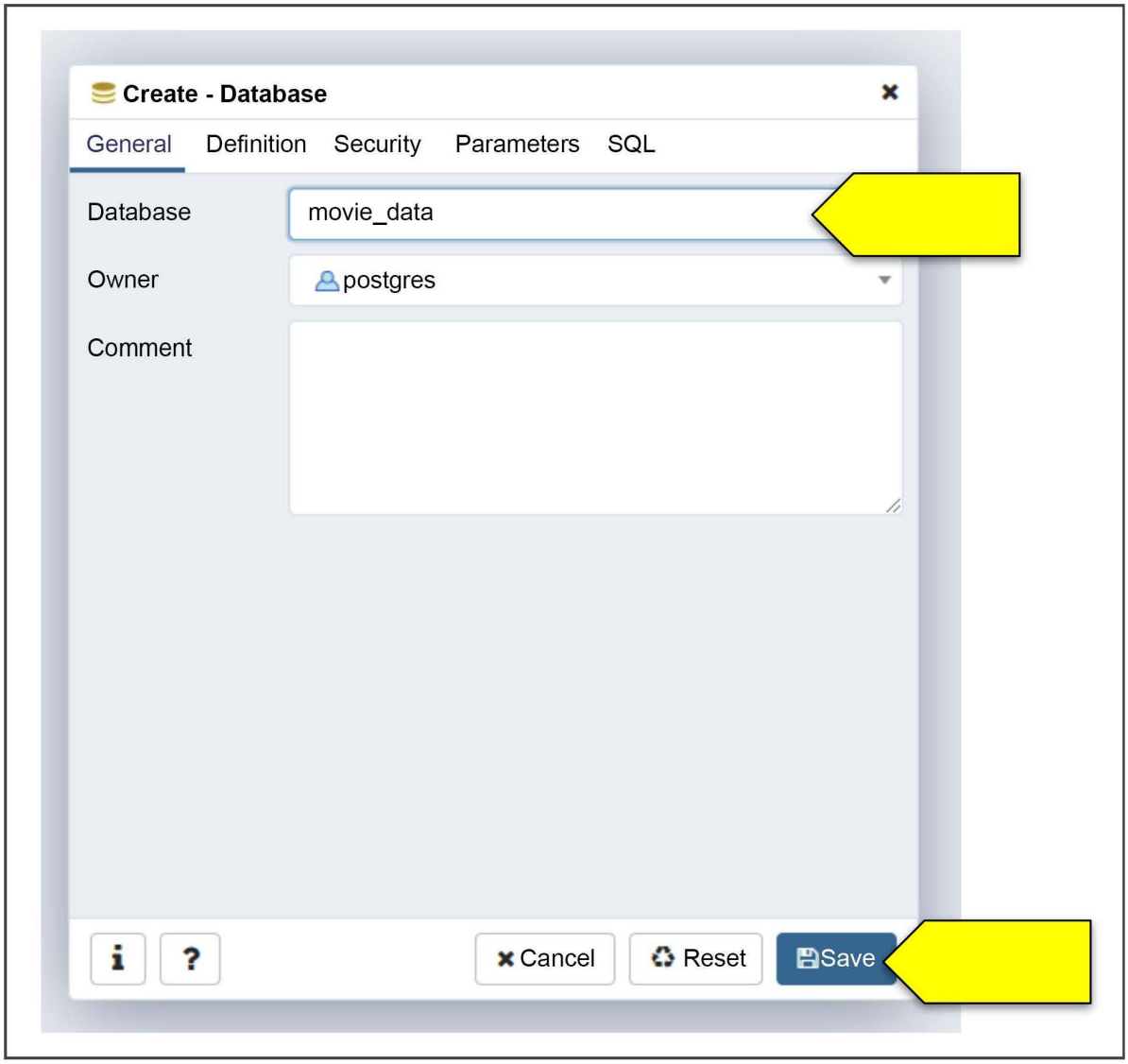
Now that we've extracted and transformed our data, it's time to load it into a SQL database. We're going to create a new database and use the built-in `to_sql()` method in Pandas to create a table for our merged movie data. We'll also import the raw ratings data into its own table.

Create a Database

Start pgAdmin and expand your local servers in the left-hand pane so you can see the Databases section. Right-click on Databases and select Create followed by Database.



Name the database “movie_data” and click Save.



Import Modules

Go back to your Jupyter Notebook. We need to import two modules. Don't forget to add these to the first cell so that all your imports are in one spot.

```
from sqlalchemy import create_engine
import psycopg2
```

You may need to install the psycopg2 module by running the following in your command line.

```
pip install psycopg2
```

Now we can create the database engine that will allow Pandas to communicate with our SQL server.

Create the Database Engine

The database engine needs to know how to connect to the database. To do that, we make a connection string. For PostgreSQL, the connection string will look like the following:

```
"postgres://[user]:[password]@[location]:[port]/[database]"
```

It looks similar to a website URL because it *is* a URL. The protocol here is "Postgres" instead of "http."

Unfortunately, this means that if we make just a simple string, we'll have to include our password, written in plaintext. Anyone who can see our code can get our database password and do anything they want with it.

> **Rewind**
> We're going to hide our database password in another file, and tell git to ignore it with `.gitignore`. Recall that we used this when we hid our API keys.

To be safe, we'll create a new file `config.py` that stores our password to a variable. We can add `config.py` to our `.gitignore` file so that git will not share our sensitive information.

Create the new file with Jupyter by going to New and selecting Text File. Rename the file `config.py`. In the text editor, add the following:

```
db_password = 'YOUR_PASSWORD_HERE'
```

In your Jupyter Notebook, add another import line:

```
from config import db_password
```

For our local server, the connection string will be as follows:

```
db_string = f"postgres://postgres:{db_password}@127.0.0.1:5432/movie_data
```

This is all the information that SQLAlchemy needs to create a database engine.SQLAlchemy handles connections to different SQL databases and manages the conversion between data types. The way it handles all the communication and conversion is by creating a database engine.

Create the database engine with the following:

```
engine = create_engine(db_string)
```

Import the Movie Data

To save the `movies_df` DataFrame to a SQL table, we only have to specify the name of the table and the engine in the `to_sql()` method.

```
movies_df.to_sql(name='movies', con=engine)
```

In pgAdmin, confirm that the table imported correctly. Follow these steps:

1. Right-click the "movies" table name and select Properties.
2. Click the Columns tab to make sure all columns have an appropriate data type.
3. Close the Properties window, and then right-click "movies" again.
4. Select "View/Edit Data" followed by "First 100 Rows."
5. Right-click "movies" and select Query Tool.
6. Inside the Query Editor, run the query `select count(*) from movies` to make sure all the rows were imported.

Nice work! Now it's time to import the ratings data.

Import the Ratings Data

The ratings data is too large to import in one statement, so it has to be divided into "chunks" of data. To do so, we'll need to reimport the CSV using the `chunksize=` parameter in `read_csv()`. This creates an iterable object, so we can make a `for` loop and append the chunks of data to the new rows to the target SQL table.

Caution

> The `to_sql()` method also has a `chunksize=` parameter, but that won't help us with memory concerns. The `chunksize=` parameter in `to_sql()` creates smaller transactions sent to SQL to prevent the SQL instance from getting locked up with a large transaction.

The simplest way to do this is with two lines:

```
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur
    data.to_sql(name='ratings', con=engine, if_exists='append')
```

This can take quite a long time to run (more than an hour). It's a really good idea to print out some information about how it's running.

Let's add functionality to this code to print out:

- How many rows have been imported
- How much time has elapsed

Step 1: Print Number of Imported Rows

Below is the previous block of code, with comments added for refactoring:

```
# create a variable for the number of rows imported
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    # print out the range of rows that are being imported

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the chunksize

    # print that the rows have finished importing
```

create a variable for the number of rows imported

We'll call the new variable `rows_imported` and give it the value 0 to start.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    # print out the range of rows that are being imported


    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'

    # print that the rows have finished importing
```

print out the range of rows that are being imported

When printing out monitoring information, it's generally a good practice to print out when a process is beginning and when a process has ended successfully, because if there's any problem, we have a better sense of which process caused the problem by seeing what part never finished successfully.

On top of this, it's good practice to keep both outputs on the same line, because it's easier to monitor which step is currently being performed. To do this, we use the `end=` parameter in the `print` function. Setting the end to an empty string will prevent the output from going to the next line.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}
```

```
    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'

    # print that the rows have finished importing
```

increment the number of rows imported by the size of 'data'

This is a great time to use the compound operator `+=` to add the length of the data read in to `rows_imported`.

> **Rewind**
>
> Remember, compound operators are shortcuts to perform a simple arithmetic operation on a variable and reassign the new value to the variable.
>
> For example, `foo += 1` is equivalent to `foo = foo + 1`.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}

    data.to_sql(name='ratings', con=engine, if_exists='append')


    # increment the number of rows imported by the size of 'data'
    rows_imported += len(data)

    # print that the rows have finished importing
```

print that the rows have finished importing

Finally, we can print that everything was imported successfully. We don't need to specify an `end=` parameter in the `print` function since we do want a new line printed now.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur


    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'
    rows_imported += len(data)

    # print that the rows have finished importing
    print('Done.')
```

Now that we're done refactoring, we can delete our comments.

```
rows_imported = 0
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    print(f'importing rows {rows_imported} to {rows_imported + len(data)}
    data.to_sql(name='ratings', con=engine, if_exists='append')

    rows_imported += len(data)

    print(f'Done.')
```

Step 2: Print Elapsed Time

This is an optional step, but it's a good idea when running a long process. We're going to print the total amount of time elapsed at every step. This is useful to estimate how long the process is going to take.

We'll use the built-in `time` module in Python. `time.time()` returns the current time whenever it is called. Subtracting two time values gives the difference in seconds. By setting a variable at the beginning to the time at the start, inside the loop we can easily calculate elapsed time and print it out.

First, we'll add the following to our import cell and rerun it:

```
import time
```

Add two new comments: one before the `for` loop, and one inside the `for` loop, right before the last `final print()` statement. The first comment is to get the start time from `time.time()`, and the second comment is to add the elapsed time to the final printout.

```
rows_imported = 0
# get the start_time from time.time()
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur

    print(f'importing rows {rows_imported} to {rows_imported + len(data)}
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done.')
```

get the start_time from time.time()

The `start_time = time.time()` method will initialize the `start_time` with the current time.

```
rows_imported = 0
# get the start_time from time.time()
```

```
start_time = time.time()
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done.')
```

The elapsed time is simply `time.time() - start_time`, which can be added directly into the f-string.

```
rows_imported = 0
# get the start_time from time.time()
start_time = time.time()
for data in pd.read_csv(f'{file_dir}the-movies-dataset/ratings.csv', chur
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done. {time.time() - start_time} total seconds elapsed')
```

Now we're ready to run this block of code and load the rating data into PostgreSQL. It may take some time to load, so you might want to use this time as a coffee break.

> **Add, Commit, Push**
> Remember to add, commit, and push your work!

Once the cell finishes running, confirm the table imported correctly using pgAdmin. Verify the columns have the correct data type, inspect the first 100 rows, and check the row count.

If everything looks good, you are done. You just extracted really messy and almost unusable data, combed through it carefully to transform it, and then loaded it into a SQL database. Now the hackathon has a reliable, clean dataset just begging to be analyzed. Britta will definitely appreciate the hard work you put in.

> **Note**
>
> Congrats on performing your first ETL process. By the way, ETL isn't the only way way to create a data pipeline (even though it's the most common). There is also the **Extract, Load, and Transform (ELT)** paradigm.
>
> With ELT, data is stored as unstructured data in a data lake and transformed when analyses are performed. This requires very powerful analytical tools to perform the transformation tasks quickly, where ETL frontloads the transformation to make analyses easier to perform.

Module 8 Challenge

Submit Assignment

Due Mar 8 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Amazing Prime loves the dataset and wants to keep it updated on a daily basis. Britta needs to create an automated pipeline that takes in new data, performs the appropriate transformations, and loads the data into existing tables.

In this challenge, you will write a Python script that performs all three ETL steps on the Wikipedia and Kaggle data. To complete this task, you can use the code you created in your Jupyter Notebook, but don't just copy and paste! You'll need to leave out any code that performs exploratory data analysis, and you may need to add code to handle potentially unforeseen errors due to changes in the underlying data.

Background

While ETL can absolutely be used for a one-time transfer of data, it becomes really powerful when it can be automated as a repeated, ongoing process. Since this process will be running without supervision, it won't be necessary to perform the exploratory data analysis steps. However, if new incoming data contains errors, the ETL process may halt or produce corrupted data. Adding `try-except` blocks will make the automated ETL script more robust to errors.

Objectives

The goals of this challenge are for you to:

- Create an automated ETL pipeline.
- Extract data from multiple sources.
- Clean and transform the data automatically using Pandas and regular expressions.
- Load new data into PostgreSQL.

Instructions

For this task, assume that the updated data will stay in the same formats: Wikipedia data in JSON format and Kaggle metadata and rating data in CSV formats. Follow these steps:

1. Create a function that takes in three arguments:
    - Wikipedia data
    - Kaggle metadata
    - MovieLens rating data (from Kaggle)
2. Use the code from your Jupyter Notebook so that the function performs all of the transformation steps. Remove any exploratory data analysis and redundant code.
3. Add the load steps from the Jupyter Notebook to the function. You'll need to remove the existing data from SQL, but keep the empty tables.
4. Check that the function works correctly on the current Wikipedia and Kaggle data.
5. Document any assumptions that are being made. Use `try-except` blocks to account for unforeseen problems that may arise with new data.

Submission

1. Save your code into a `challenge.py` script in your repo folder.
2. Add, commit, and push this new file to your repository.

3. Submit the link to your repository through Canvas.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 8 Rubric

Criteria	Ratings					Pts
Create a Function	**20.0 pts** **Mastery** Create a function that successfully takes in three arguments: Wikipedia data, Kaggle metadata, and MovieLens rating data.	**15.0 pts** **Approaching Mastery** Create a function that successfully takes in three arguments: Wikipedia data, Kaggle metadata, and MovieLens rating data, with one or two minor errors.	**10.0 pts** **Progressing** Create a function that successfully takes in two of these three arguments: Wikipedia data, Kaggle metadata, and MovieLens rating data, with one or two minor errors.	**5.0 pts** **Emerging** Create a function that successfully takes in one of these three arguments: Wikipedia data, Kaggle metadata, and MovieLens rating data.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty.	20.0 pts
Ensure function performs transformation steps	**25.0 pts** **Mastery** Function performs all transformation steps, with no errors.	**20.0 pts** **Approaching Mastery** Function performs most transformation steps, with one or two minor errors.	**15.0 pts** **Progressing** Function performs some transformation steps, with more than two minor errors.	**10.0 pts** **Emerging** Function performs one or two transformation steps, with more than two errors.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty.	25.0 pts
Add load steps to function	**25.0 pts** **Mastery** Function performs all load steps with no errors. Additionally, existing data is removed from SQL and tables are not removed from SQL.	**20.0 pts** **Approaching Mastery** Function performs all load steps with one or two minor errors. Additionally, existing data is removed from SQL and tables are not removed from SQL.	**15.0 pts** **Progressing** Function performs all load steps with one or two minor errors. Additionally, existing data is removed from SQL OR tables are not removed from SQL.	**10.0 pts** **Emerging** Function performs all load steps, with significant errors.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty.	25.0 pts

Criteria	Ratings					Pts
Document assumptions	**30.0 pts** **Mastery** Presents a cohesive written analysis that documents 5+ valid assumptions.	**22.5 pts** **Approaching Mastery** Presents a cohesive written analysis that documents 3-4 valid assumptions.	**15.0 pts** **Progressing** Presents a developing written analysis that documents 2-3 valid assumptions.	**7.5 pts** **Emerging** Presents a limited written analysis that documents 1-2 valid assumptions.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty.	30.0 pts
						Total Points: 100.0

Module 8 Career Connection

Will your network hire you?

Introduction:

Did you know that 80% of jobs never get posted and are only found through networking? Increase your chances of finding the right opportunity by expanding your network. Contact your Career Director for access to your Regional Guide for a collection of MeetUps, networking groups, and recruiters in your market.

Career Services Next Step: Link Milestone 3 (https://courses.bootcampspot.com/courses/138/pages/milestone-3-build-your-visibility)

Milestone 3: Build Your Visibility

"80% of jobs never get posted and are only found through networking."
– The Muse (https://www.themuse.com/advice/6-insider-job-search-facts-thatll-make-you-rethink-how-youre-applying)

Key Takeaways

- By the end of this Milestone, you will be able to develop a plan for networking and outreach.
- Review this guide, and mark it complete.
- For more insight on networking, view the "Expand Your Network" workshop below.

Employer Competitive candidates stand out in their search by making themselves visible and selling their strengths. They're clear on their search **goals**, the **value** they're able to add to their target jobs, and the importance of **increasing their visibility** to help them reach those goals.

For a guided experience on networking, see the video above.

Getting Started

1. Review the guide below for best practices on in-person and online networking.
2. If you'd like additional support with networking, see the resources we've included at the end of this guide.
3. Once you're done, mark this milestone complete. Reminder: This Milestone **does not** require a submission to a Profile Coach.

About Networking

Successful networking involves building relationships, and it's important to note, that building a strong network doesn't happen overnight. It takes time and consistent effort, but it will be worth it!

Networking IS....

- A chance to learn more about an industry and what employers are looking for
- A chance to gain visibility and make new connections
- A chance to gain confidence in your ability to describe your interests, skills, values (which will help you in a real interview)

Networking IS NOT...

- Asking for a job
- An interview for employment
- A guarantee of employment or employability
- Just a business card swap at a meeting or conference
- Lots of connections on LinkedIn

In-Person Networking

(1) Identify Your Current Network
You may have more people in your network than you realize. Use the categories below to help you think about who's in your network and how you might reach out to them.

  - Family
  - Friends
  - Former Colleagues/ Supervisors
  - Professors, Trainers, Etc.

- Other Connections

(2) Expand your Network

Consider the following approaches to expand your network:

- **Reach out to every person on your networking list above**, and send them your materials with a specific ask. "Asks" can include a quick chat on the phone for advice or a lunch date to talk about your target industry as well as recommendations for who you should connect with next.
- If you're currently employed, **ask your boss for projects that require you to interact with new departments or individuals**. For example, you can propose that you help the company enhance its website, and in doing so, you'll interact with other developers and/or the marketing department.
- **Find volunteer opportunities**. Get involved in an organization or group that interests you, and offer to contribute some of your new tech skills. You may meet people who can be helpful.
- **Create business cards** that include your target role, links to Github, LinkedIn, and a QB code to scan for your resume.
- Continue to use **LinkedIn weekly to connect with employees and decision makers**. Look for people who might have secondary connections to you. Send personal messages about your passions and common interests, and request informational interviews.
- **Here is a great reminder of all the many places where you can network. (http://www.jobmonkey.com/best-places-to-network/)**

(3) Attend Networking Events

It's always helpful to set a goal when attending networking events (e.g. "I will have 3 meaningful conversations that may lead to potential follow-up," or "I will not leave until I have entered into at least 5 conversations."). Establishing a goal allows you to set a measurable standard of success for the event, which can help change your experience of networking into a positive one.

TIP: Always bring business cards with you to events. On the back of the cards you receive, take notes about the person you're speaking with so that you can follow

up in a personal way.

Here are some additional tips that will help you differentiate yourself at an event:

1. **Master the use of tech language** – The better your vocabulary (especially as it relates to your industry), the more impressed people will be. Being confident, articulate, and knowledgeable will help you create a strong first impression.
2. **Eye Contact** – Always maintain eye contact when you're speaking with someone. Looking away can make you appear less confident. Also, remember to smile.
3. **Leave personal space** – Don't stand too close to anyone. Keep a reasonable distance.
4. **Acknowledge your understanding** – When someone else is talking, acknowledge that you heard them with non-verbal body language such as nodding.
5. **Wait your turn** – Successful professionals are also good listeners. Allow your new connection the opportunity to complete their thoughts before offering a response.
6. **Watch body language** – Mirror the body language of the person with whom you are interacting. If they sit down, you should sit down too— they may be ready for a longer conversation. Try not to cross or fold your arms, as that may create the appearance that you are guarded. Overall, be mindful of both you and your new connection's body language.
7. **Be curious** – Open the conversation with questions. Focus on the other person's interests first, and show genuine interest.

Here are some conversation starters that might help you as well. (https://www.themuse.com/advice/30-brilliant-networking-conversation-starters)

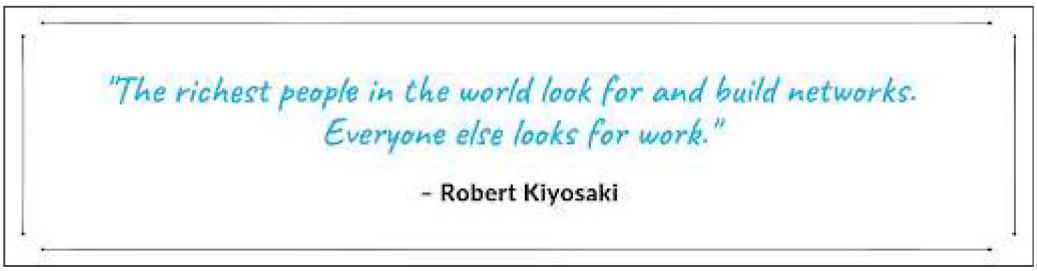
> *"Networking is more about farming than it is about hunting. It's about cultivating relationships."*
>
> – Ivan Misner

(4) Follow-Up

Networking only works if you follow up! After meeting new contacts, follow up with a personal message soon after you've met.

When reaching out, whether via email or phone, here are a few tips:

1. **Remind them how they know you**. Always begin by referencing a common person, event, educational experience, work experience, organization, or award that creates a common bond.
2. **Be clear on what you bring to the table**. Express interest in the person's work, and add value instead of asking for something. Sharing interesting articles, making introductions to helpful contacts, supporting the contact's endeavors, and engaging with their LinkedIn posts are great ways to add value.
3. **Be flexible with scheduling**. Make it easy and convenient for the contact to say yes to connecting again!
4. **Do your homework!** Research your new connections to help you better foster a relationship with them. LinkedIn and general internet searches provide instant access to information on your targeted connections.
5. **Don't give up, and don't take it personally**. Some people hesitate to reach out again for fear of being ignored, rejected, or of being a pest. It's okay if someone doesn't take you up on your offer. If you are reaching out to people regularly, you'll get more accepted invitations than passes.
6. **Breathe, and stay calm**. It's perfectly normal to be nervous about calling people. Networking is a skill that requires practice. It may help to practice your calls with friends or family. It may also help to remember that you're not calling to ask for favors — you are asking to learn from someone. Most people love sharing their expertise!



(5) Request Informational Interviews & Seek Mentors

Informational interviews are your opportunity to explore whether a your goals or current opportunities really are the right match for you. They're also great ways to expand your network through introductions from the connection you're interviewing.

Before the Interview

DO

- Research the individual — use LinkedIn, Google them, or personal connections to prepare.
- Prepare a list of questions (at least 4).
- Review a list of conversation starters for informational interviews, and have a few ready to go.
- Be ready to deliver your elevator pitch.

DO NOT

- Plan to "wing it" — while these are not job interviews, preparation is needed.
- Script every second of the interview — you need to build a relationship as well.
- Assume this person is going to lead the conversation or listen to you talk the entire time.

During the Interview

DO

- Smile, be aware of appropriate eye contact, and lean forward.
- Ask questions to demonstrate interest and active listening.

- Find a personal connection through interests, passions, or hobbies.
- Listen for ways you may be able to help or volunteer for them.
- Use varied tones and volumes to demonstrate your passion and enthusiasm.
- Describe work you have done that might be interesting.

DO NOT

- Complain about previous employers or peers.
- Dominate conversation — be sure to let them talk.
- Answer questions with one word answers — be concise, but be thorough too.
- Look at your phone during the conversation.

After the Interview

DO

- Jot down notes to remember the conversation.
- Write a thank you email.
- Follow up about once a month with updates and check ins.

DO NOT

- Follow up too frequently (more than about once per month).
- Text a thank you — this should be a more formal thank you.

NOTE: Informational interviews should lead to more interviews, volunteer or open-source projects, or ideas about new directions to take. For more on informational interviews, check out this **article called 5 Tips for Non-Awkward Informational Interviews (https://www.themuse.com/advice/5-tips-for-nonawkward-informational-interviews)** .

Online Networking

SOCIAL MEDIA

- Make sure your profiles look polished on platforms like LinkedIn, Angel.co, and any others. Examples of excellent Data profiles can be found here:
  - **https://www.linkedin.com/in/robinhchoi/ (https://www.linkedin.com/in/robinhchoi/)**
  - **https://www.linkedin.com/in/carlosmarin2/ (https://www.linkedin.com/in/carlosmarin2/)**
  - **https://www.linkedin.com/in/dylansather/ (https://www.linkedin.com/in/dylansather/)**
  - **https://www.linkedin.com/in/leonardo-apolonio/ (https://www.linkedin.com/in/leonardo-apolonio/)**
- On LinkedIn, Facebook, and other platforms, follow companies, thought leaders, and professionals in the industry. Learn how to do this here: **http://bit.ly/2Ec5ikA (http://bit.ly/2Ec5ikA)** . Engage with these companies and individuals through likes and comments on their posts.
- Look for alumni groups for your current or past organizations.
- **Use these templates to help you draft your outreach messages. (http://bit.ly/2vRZj24)**

Additional Resources

- **10 Simple Ways to Improve Your Networking Skills (https://www.youtube.com/watch?v=E5xTbn6OnAA)**

Minimize Video

- **How to Find Your Next Job Over Coffee (https://blog.udacity.com/2014/11/informational-interviews-how-to-find.html)**
- **Visit the 'Build Your Visibility' section of your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents)** – for outreach templates, potential jobs, and additional networking resources.

Please check the box below to verify that you have completed this milestone.

- ○ Yes, I have completed this milestone.

Check Answer

Finish ▸

Module 8 Survey

Due No due date **Questions** 14 **Time Limit** None

Instructions

Congratulations on completing the module. Please take a moment to reflect back over the last week and provide feedback. We value this feedback as we continually enhance our program.

Take the Survey