Real World Data: Working with APIs & JSON

- As is often the case, when you must work with web APIs to acquire data, it often comes in the format of JSON files
- JSON stands for JavaScript object notation and is a language independent data format to store, transform,and interchange data.
- all popular programming languages know how to handle JSON data, therefore we can interchange data between these languages.
- JSON is the standard format to transfer data through APIs
- Not tabular in nature, but often complex, nested data structures
- Files often have different versions which contain different orientations of the data
- orientation is very important when loading JSON files into pandas

```
import pandas as pd
import requests
import json
pd.options.display.max_columns = 30
```

Importing data from JSON files:

Importing the JSON data (way no. 1):

```
with open('data/blockbusters.json') as file:
    # loading JSON file into a python object saved as data
    data = json.load(file)
```

```
# data
```

```
type(data)
```

```
list
```

```
len(data)

print(f'There are {len(data)} films in this JSON file.')
```

```
There are 18 films in this JSON file.
```

```
data[10]
```

```
{'title': 'Black Panther',
 'id': 284054,
 'revenue': 1346739107,
 'genres': [{'id': 28, 'name': 'Action'},
```

```
  {'id': 12, 'name': 'Adventure'},
  {'id': 14, 'name': 'Fantasy'},
  {'id': 878, 'name': 'Science Fiction'}],
 'belongs_to_collection': {'id': 529892,
  'name': 'Black Panther Collection',
  'poster_path': '/9ZSPIsxI3TZDgfg0Jzk0RZl4INg.jpg',
  'backdrop_path': '/1Jj7FrjjbewB6Q6dl6YXhL3kuvL.jpg'},
 'runtime': 134}
```

Creating Pandas df out of the JSON file:

```
df = pd.DataFrame(data)
df.head(3)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162
2	Star Wars: The Force Awakens	140607	2068223624	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 10, 'name': 'Star Wars Collection', 'po...	136

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 6 columns):
 #   Column                 Non-Null Count  Dtype 
---  ------                 --------------  ----- 
 0   title                  18 non-null     object
 1   id                     18 non-null     int64 
 2   revenue                18 non-null     int64 
 3   genres                 18 non-null     object
 4   belongs_to_collection  15 non-null     object
 5   runtime                18 non-null     int64 
dtypes: int64(3), object(3)
memory usage: 992.0+ bytes
```

Importing the JSON data (way no. 2):

```
df = pd.read_json('data/blockbusters.json')
df.head(3)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162

	title	id	revenue	genres	belongs_to_collection	runtime
2	Star Wars: The Force Awakens	140607	2068223624	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 10, 'name': 'Star Wars Collection', 'po...	136

Data still has a lot of nested structures:

```
df.belongs_to_collection[0]
```

```
{'id': 86311,
 'name': 'The Avengers Collection',
 'poster_path': '/yFSIUVTCvgYrpalUktulvk3Gi5Y.jpg',
 'backdrop_path': '/zuW6fOiusv4X9nnW3paHGfXcSll.jpg'}
```

```
df.genres[0]
```

```
[{'id': 12, 'name': 'Adventure'},
 {'id': 878, 'name': 'Science Fiction'},
 {'id': 28, 'name': 'Action'}]
```

Flattening JSON files:

```
# This flattens the belongs_to_collection column
pd.json_normalize(data = data, sep = "_").head(3)
```

	title	id	revenue	genres	runtime	belongs_to_collection_id	belongs_to_collection_name	b
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	181	86311.0	The Avengers Collection	/y
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	162	87096.0	Avatar Collection	/nslJV
2	Star Wars: The Force Awakens	140607	2068223624	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	136	10.0	Star Wars Collection	/iTQH

The above did not flatten the genres column. It is still nested:

```
pd.json_normalize(data = data, sep = "_").genres[0]
```

```
[{'id': 12, 'name': 'Adventure'},
 {'id': 878, 'name': 'Science Fiction'},
 {'id': 28, 'name': 'Action'}]
```

Creating an entirely new DF from the nested original data:

`pd.json_normalize(data, record_path = genres)`

- this indicates we want to dig deeper into the still-nested "genres" column

- here, each row stands for a different combination of film and genre, like using `pd.stack()` when unpacking
- `meta = [list of columns from original to include]`
- `record_prefix = prefix` - to use for columns that would throw errors, such as duplicate `id` columns, the genre id and the original movie id

```
pd.json_normalize(data = data, record_path='genres',
                  meta = ['title', 'id'],
                  record_prefix = 'genre_').head(3)
```

	genre_id	genre_name	title	id
0	12	Adventure	Avengers: Endgame	299534
1	878	Science Fiction	Avengers: Endgame	299534
2	28	Action	Avengers: Endgame	299534

JSON Orientation and Formats:

- 3 major orientations: records, columns, and split
- The one above is a records orientation, `blockbusters.json`
- records - every element of the list is one record, one movie, one row
- columns - data is organized in columns, `blockbusters2.json`
- split - column labels for all rows, then index labels for all rows, then the data - `blockbusters3.json` - can minimize memory usage but causes issues with Pandas
- Records orientation is best for Pandas, and should always be used unless there are extreme memory restrictions.

Columns Orientation:

- a collection of dictionaries, each being one of the columns
- cannot use `pd.json_normalize()` on this orientation
- this is why the records orientation is best for working in Pandas

Using `with open`

```
with open('data/blockbusters2.json') as file:
    data2 = json.load(file)

# data2
```

```
pd.DataFrame(data2).head(2)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181

	title	id	revenue	genres	belongs_to_collection	runtime
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162

Using `pd.read_json()`

```
df2 = pd.read_json('data/blockbusters2.json')
df2.head(3)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162
2	Star Wars: The Force Awakens	140607	2068223624	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 10, 'name': 'Star Wars Collection', 'po...	136

```
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 18 entries, 0 to 17
Data columns (total 6 columns):
 #   Column                 Non-Null Count  Dtype 
---  ------                 --------------  ----- 
 0   title                  18 non-null     object
 1   id                     18 non-null     int64 
 2   revenue                18 non-null     int64 
 3   genres                 18 non-null     object
 4   belongs_to_collection  15 non-null     object
 5   runtime                18 non-null     int64 
dtypes: int64(3), object(3)
memory usage: 1008.0+ bytes
```

Split Orientation:

- dictionary with three elements: column labels, row labels, and data
- cannot be used with `pd.DataFrame()`
- if using `pd.read_json()` must specify orientation
- also does not work with `pd.json_normalize()`

```
with open('data/blockbusters3.json') as file:
    data3 = json.load(file)

# data3
```

```
df3 = pd.read_json('data/blockbusters3.json', orient = "split")
df3.head(2)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
1	Avatar	19995	2787965087	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162

```
api_key = "31eb33083cb6b5f3c9e5c3b980247eb9"
```

```
movie_id = 140607
movie_api = 'https://api.themoviedb.org/3/movie{}?'
```

Real World Data: Web APIs

- **API** = Application programming interface - defines how an application works with other programs
- Web APIs are the most common ways to get dta from a web application without knowing the database details
- Other APIs = Twitter, YouTube, etc.
- Users can pull data from Web APIs with HTTP requests with specific queries and get data in JSON, CSV, or other file formats
- Details are usually defined in an API's documention, which must be followed to write the requests.
- Users can often cut and paste the formate for their HTTP requests, but this can also be automated with Python using the **`requests`** library

HTTP Request Example:

URL :
`https://api.themoviedb.org/3/discover/movie?api_key=api_key&language=en-US&sort_by=pc`

- this includes the API address for the movies database, version 3
- indicates that you want to search movies
- and then pass API key
- required parameters = API key

Examples from the API website

API Documentation - Typically going to work with the Discover Module and Movies Module

Try It Out (build queries)

```
import pandas as pd
import requests
import json
pd.options.display.max_columns = 30
```

Movies Module: pulling data from the database

- building the URL

```
movie_id = 140607
# {} = placeholder for movie id
movie_api = 'https://api.themoviedb.org/3/movie/{}?'
api_key = "api_key=31eb33083cb6b5f3c9e5c3b980247eb9"
# .format() will put our data fragments into the {}
url = movie_api.format(movie_id) + api_key
url
```

```
'https://api.themoviedb.org/3/movie/140607?api_key=31eb33083cb6b5f3c9e5c3b980247eb9'
```

Automating the Process:

```
# This will return a response object, which we will save as "response"

response = requests.get(url)
print(response)

# Returns the JSON-encoded content of the response
data = response.json()

# Response code 200 means there was no problem getting the data
```

```
<Response [200]>
```

Converting to Pandas:

Since this response is only one row of a dataframe, it will not work to convert to dataframe. But it can be converted into a series

```
# pd.Series(data)
```

```
# Converting to dataframe after converting to series
# Must transform, or else columns come out as rows

df = pd.Series(data).to_frame().T
```

```
df
```

	adult	backdrop_path	belongs_to_collection	budget	genres	
0	False	/8BTsTfln4jlQrLXUBquXJ0ASQy9.jpg	{'id': 10, 'name': 'Star Wars Collection', 'po...	245000000	[{'id': 12, 'name': 'Adventure'}, {'id': 28, '...	http://www.starwars.com/wa

Passing to `pd.json_normalize()`

- using `record_path =` to get the nested data unpacked

```
pd.json_normalize(data = data, sep = '_', record_path =  'genres', meta = 'title')
```

	id	name	title
0	12	Adventure	Star Wars: The Force Awakens
1	28	Action	Star Wars: The Force Awakens
2	878	Science Fiction	Star Wars: The Force Awakens
3	14	Fantasy	Star Wars: The Force Awakens

```
pd.json_normalize(data = data, sep = '_', record_path =  'production_companies', meta =
```

	id	logo_path	name	origin_country	title
0	1	/o86DbpburjxrqAzEDhXZcyE8pDb.png	Lucasfilm Ltd.	US	Star Wars: The Force Awakens
1	11461	/p9FoEt5shEKRWRKVIlvFaEmRnun.png	Bad Robot	US	Star Wars: The Force Awakens

---

Discover Module: pulling data from the database

- Discover module returns less detailed results about the films than the movies module

```
discover_api = 'https://api.themoviedb.org/3/discover/movie?'

# querying for films released between Jan 1 and Feb 29 2020
# gte means >= and lte means <=
query = "&primary_release_date.gte=2020-01-01&primary_release_date.lte=2020-02-29"

# combining all parts of the URL
url = discover_api + api_key + query

# getting the response to the request and saving as JSON
data = requests.get(url).json()

# The results column contains the actual results, the others
# are meta data about the search, such as how many results,
# how many pages of results, etc.
pd.DataFrame(data['results']).head(2)

# By default, this returns the first page of the data.
```

	adult	backdrop_path	genre_ids	id	original_language	original_title	overview	popul
0	False	/z8sNNjEXEpZNQCHCuo3QH8kK00t.jpg	[10749]	665142	ko	□□ □□ 3	Seok-yeong has been living with his father eve...	105
1	False	/stmYfCUGd8Iy6kAMBr6AmWqx8Bq.jpg	[28, 878, 35, 10751]	454626	en	Sonic the Hedgehog	Powered with incredible speed, Sonic The Hedge...	86

```
# To get other pages, add to the query:

query = "&primary_release_date.gte=2020-01-01&primary_release_date.lte=2020-02-29&page=

url = discover_api + api_key + query
```

```
data = requests.get(url).json()

pd.DataFrame(data['results']).head(2)
```

	adult	backdrop_path	genre_ids	id	original_language	original_title	overview	popul
0	False	/6mKAKhj8POVGqV1GsroS5mGIUe9.jpg	[14, 28, 12]	666750	en	Dragonheart: Vengeance	Lukas, a young farmer whose family is killed b...	23
1	False	/aiQlCxiWNcOsJruYxdPuhb6WtWu.jpg	[9648, 18, 27, 12, 16]	658558	ja	□□□ □□□□	After an airplane crash during a school trip, ...	23

Importing and Saving Movies Datasets:

- with the discover module, a user can search with a variety of conditions
- then the movie ids can be extracted and used to do a more detailed search using the movies module

```
# list of movie ids desired
movie_ids = [0, 299534, 19995, 140607, 299536, 597, 135397,
             420818, 24428, 168259, 99861, 284054, 12445,
             181808, 330457, 351286, 109445, 321612, 260513]

basic_url = 'https://api.themoviedb.org/3/movie/{}?{}'
```

Automating and scaling:

```
json_list = []

# for-loop performing same operations as above for multiple
for movie in movie_ids:
    url = basic_url.format(movie, api_key)
    response = requests.get(url)
    # if status code is not good, skip that movie id
    if response.status_code != 200:
        continue
    else:
        data = response.json()
        json_list.append(data)

df = pd.DataFrame(json_list)
```

```
# if data is not available or id is wrong:
requests.get(basic_url.format(0, api_key)).status_code
```

```
df.head(3)
```

	adult	backdrop_path	belongs_to_collection	budget	genres	
0	False	/7RyHsO4yDXtBv1zUU3mTpHeQ0d5.jpg	{'id': 86311, 'name': 'The Avengers Collection...	356000000	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	https://www.marvel.co
1	False	/7ABsaBkO1jA2psC8Hy4IDhkID4h.jpg	{'id': 87096, 'name': 'Avatar Collection', 'po...	237000000	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	https://www.avatar
2	False	/8BTsTfln4jlQrLXUBquXJ0ASQy9.jpg	{'id': 10, 'name': 'Star Wars Collection', 'po...	245000000	[{'id': 12, 'name': 'Adventure'}, {'id': 28, '...	http://www.starwars.c

Narrowing data and sorting by revenue:

```
# Extracting only these specified columns
# Must have [:,[]] or throws error
# This will get all the data in these columns

df = df.loc[:, ["title", "id", "revenue", "genres", "belongs_to_collection", "runtime"]]
```

```
df.head(2)
```

	title	id	revenue	genres	belongs_to_collection	runtime
1	Avatar	19995	2920357254	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162
0	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181

Exporting and saving as JSON file:

```
df.to_json("movies.json", orient = 'records')
```

Reimporting the exported data:

```
with open('movies.json') as file:
    data = json.load(file)

pd.json_normalize(data).head(3)
```

	title	id	revenue	genres	runtime	belongs_to_collection.id	belongs_to_collection.name	bel

	title	id	revenue	genres	runtime	belongs_to_collection.id	belongs_to_collection.name	bel
0	Avatar	19995	2920357254	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	162	87096.0	Avatar Collection	/uO2yU
1	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	181	86311.0	The Avengers Collection	/yFS
2	Titanic	597	2187463944	[{'id': 18, 'name': 'Drama'}, {'id': 10749, 'n...	194	NaN	NaN	

Normalizing and creating a separate genre df:

```
# data = data, record_path = 'genres', meta = 'title'
pd.json_normalize(data, "genres", 'title').head(3)
```

	id	name	title
0	28	Action	Avatar
1	12	Adventure	Avatar
2	14	Fantasy	Avatar

Real World: Importing and Saving Movies Dataset

```
df.head(3)
```

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avatar	19995	2920357254	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162
1	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
2	Titanic	597	2187463944	[{'id': 18, 'name': 'Drama'}, {'id': 10749, 'n...	None	194

```
# Not the best method, not very efficient
df.to_csv("movies_raw.csv", index = False)
```

```
df = pd.read_csv('movies_raw.csv')
```

What happens when exporting to CSV:

```
df.head(3)
```

	title	id	revenue	genres	belongs_to_collection	runtime

	title	id	revenue	genres	belongs_to_collection	runtime
0	Avatar	19995	2920357254	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	{'id': 87096, 'name': 'Avatar Collection', 'po...	162
1	Avengers: Endgame	299534	2797800564	[{'id': 12, 'name': 'Adventure'}, {'id': 878, ...	{'id': 86311, 'name': 'The Avengers Collection...	181
2	Titanic	597	2187463944	[{'id': 18, 'name': 'Drama'}, {'id': 10749, 'n...	NaN	194

```
print("The nested date was converted to: ", type(df.genres[0]), "\n")
```

```
The nested date was converted to:  <class 'str'>
```

JSON Data Cleaning:

- handle and clean data that has been saved to a csv file
- some text and numerical data that does not make sense
- removing duplicates
- dropping irrelevant columns
- dealing with stringified data, returning to lists, dicts, etc
- flattening nested data

```
import pandas as pd
import requests
import json
pd.options.display.max_columns = 30
```

---

1. **Load** and **inspect**

the messy dataset **movies_metadata.csv**. Identify columns with nested / stringified json data.

```
movies = pd.read_csv('movies_metadata.csv', low_memory=False)
```

```
movies.head(2)
```

	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id	original_lang
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...	http://toystory.disney.com/toy-story	862	tt0114709	
1	False		NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	NaN	8844	tt0113497	

Issues upon import:

- mixed data types in columns like budget
- columns that should be numeric but are listed as object due to mixed data types
- release_date should be datetime
- popularity should be numeric
- some columns have a lot of missing values

```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45466 entries, 0 to 45465
Data columns (total 24 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   adult                  45466 non-null  object 
 1   belongs_to_collection  4494 non-null   object 
 2   budget                 45466 non-null  object 
 3   genres                 45466 non-null  object 
 4   homepage               7782 non-null   object 
 5   id                     45466 non-null  object 
 6   imdb_id                45449 non-null  object 
 7   original_language      45455 non-null  object 
 8   original_title         45466 non-null  object 
 9   overview               44512 non-null  object 
 10  popularity             45461 non-null  object 
 11  poster_path            45080 non-null  object 
 12  production_companies   45463 non-null  object 
 13  production_countries   45463 non-null  object 
 14  release_date           45379 non-null  object 
 15  revenue                45460 non-null  float64
 16  runtime                45203 non-null  float64
 17  spoken_languages       45460 non-null  object 
 18  status                 45379 non-null  object 
 19  tagline                20412 non-null  object 
 20  title                  45460 non-null  object 
 21  video                  45460 non-null  object 
 22  vote_average           45460 non-null  float64
 23  vote_count             45460 non-null  float64
dtypes: float64(4), object(20)
memory usage: 8.3+ MB
```

Messy nested data:

```
print(type(movies.genres[0]))
movies.genres[0] # nested, stringified JSON data
```

```
<class 'str'>
```

```
"[{'id': 16, 'name': 'Animation'}, {'id': 35, 'name': 'Comedy'}, {'id': 10751, 'name': 'Family'}]"
```

```
print(type(movies.belongs_to_collection[0]))
movies.belongs_to_collection[0]
```

```
<class 'str'>
```

```
"{'id': 10194, 'name': 'Toy Story Collection', 'poster_path':
'/7G9915LfUQ2lVfwMEEhDsn3kT4B.jpg', 'backdrop_path':
'/9FBwqcd9IRruEDUrTdcaafOMKUq.jpg'}"
```

```
print(type(movies.production_companies[0]))
movies.production_companies[0]
```

```
<class 'str'>
```

```
"[{'name': 'Pixar Animation Studios', 'id': 3}]"
```

```
print(type(movies.production_countries[0]))
movies.production_countries[0]
```

```
<class 'str'>
```

```
"[{'iso_3166_1': 'US', 'name': 'United States of America'}]"
```

```
print(type(movies.spoken_languages[0]))
movies.spoken_languages[0]
```

```
<class 'str'>
```

```
"[{'iso_639_1': 'en', 'name': 'English'}]"
```

---

2. **Drop** the irrelevant columns

'adult', 'imdb_id', 'original_title', 'video' and 'homepage'.

```
movies.drop(columns = ['adult', 'imdb_id', 'original_title', 'video', 'homepage'], inpl
```

---

3. Handle stringified JSON columns

Evaluate Python Expressions in the stringified columns ["belongs_to_collection", "genres", "production_countries", "production_companies", "spoken_languages"] and **remove quotes** ("") where possible.

```
import ast # abstract syntax trees
```

```
json_col = ["belongs_to_collection", "genres",
            "production_countries", "production_companies",
            "spoken_languages"]
```

De-Stringing:

- in the cells above, showing the string data, there are single quotes around strings and double quotes around the entire container that has been stringified
- this can be used to automate cleaning
- see the step by step below with examples

`json.loads()` - destringifies the JSON data into a Python object

```
# Example JSON data
# (must have single quotes then double for json.loads to work)

json_01 = '{"dog": 3, "cat": 5}'

print("json.loads() has decoded the data to: ", type(json.loads(json_01)))

json.loads(json_01)
```

```
json.loads() has decoded the data to:  <class 'dict'>
```

```
{'dog': 3, 'cat': 5}
```

Replace quote types in JSON data:

```
# replace the quote types

json_01.replace("'", '"')
```

```
'{"dog": 3, "cat": 5}'
```

```
json.loads(json_01.replace("'", '"'))
```

```
{'dog': 3, 'cat': 5}
```

Lambda function to fix quotes:

```
print("The data has been converted to type: ", type(movies.genres.apply(lambda x: json.

movies.genres.apply(lambda x: json.loads(x.replace("'", '"')))[0]
```

```
The data has been converted to type:  <class 'list'>
```

```
[{'id': 16, 'name': 'Animation'},
 {'id': 35, 'name': 'Comedy'},
 {'id': 10751, 'name': 'Family'}]
```

Better way: `ast.literal_eval()`

- evaluates a Python string and tries to find the appropriate data type in the string

- the cleaner and safer way to reformat and un-stringify the data

```
# json.loads could not work with this data
# the quotes are swapped
# but ast.literal_eval() can handle it

json_02 = "{'dog': 3, 'cat': 5}"

print(type(ast.literal_eval(json_02)))

(ast.literal_eval(json_02))
```

```
<class 'dict'>
```

```
{'dog': 3, 'cat': 5}
```

```
# Applying ast.literal_eval() to entire column

movies.genres = movies.genres.apply(ast.literal_eval)
```

```
# Applying to all JSON data columns:
# Will not work as long as there is mixed data

# movies.loc[:, json_col].apply(ast.literal_eval, axis = 0)
```

De-Stringing CSV-Imported JSON Data:

- dealing with columns that have mixed data types and do not allow `ast.literal_eval()` to work properly
- it requires all data to be strings

```
import numpy as np
```

```
# Turn column data into strings so that ast.literal_eval() will work
# If it is a string, convert it, if not turn to NaN

movies.belongs_to_collection = movies.belongs_to_collection.apply(lambda x: ast.literal
                                    if isinstance(x, str) else np.nan)
```

```
type(movies.belongs_to_collection[0])
```

```
dict
```

Data conversion on JSON columns:

```
movies.spoken_languages = movies.spoken_languages.apply(lambda x: ast.literal_eval(x)
                                    if isinstance(x, str) else np.nan)

movies.production_companies = movies.production_companies.apply(lambda x: ast.literal_e
                                    if isinstance(x, str) else np.nan)

movies.production_countries = movies.production_countries.apply(lambda x: ast.literal_e
                                    if isinstance(x, str) else np.nan)
```

Function to convert JSON data to all applicable columns:

```
# Re-import data to apply the function:

movies = pd.read_csv('movies_metadata.csv', low_memory=False)

movies.drop(columns = ['adult', 'imdb_id', 'original_title', 'video', 'homepage'], inpl
```

```
def convert_json(df, json_column_list):

    def convert_column(col):
        col = col.apply(lambda x:
                        ast.literal_eval(x)
                        if isinstance(x, str)
                        else np.nan)
        return col
    print("Converting these columns: ")

    for col in json_column_list:
        current_type = type(df[col][0])
        df[col] = convert_column(df[col])
        print("\t -", col, '⇨', current_type, "to", type(df[col][0]))

    return df
```

```
movies = convert_json(movies, json_col)
```

```
Converting these columns:
     - belongs_to_collection ⇨ <class 'str'> to <class 'dict'>
     - genres ⇨ <class 'str'> to <class 'list'>
     - production_countries ⇨ <class 'str'> to <class 'list'>
     - production_companies ⇨ <class 'str'> to <class 'list'>
     - spoken_languages ⇨ <class 'str'> to <class 'list'>
```

---

4. Flattening Collection:

Extract only the **collection name** from the column "belongs_to_collection" and **overwrite** "belongs_to_collection".

For example: The value in the first row (Toy Story) should be 'Toy Story Collection'.

```
# We need to extract the value from name
movies.belongs_to_collection[0]
```

```
{'id': 10194,
 'name': 'Toy Story Collection',
 'poster_path': '/7G9915LfUQ2lVfwMEEhDsn3kT4B.jpg',
 'backdrop_path': '/9FBwqcd9IRruEDUrTdcaafOMKUq.jpg'}
```

```
movies.belongs_to_collection = movies.belongs_to_collection.apply(lambda x: x['name'] i
```

```
# The collection field has been updated with only the name of the collection
movies.belongs_to_collection[0]
```

```
'Toy Story Collection'
```

```
movies.belongs_to_collection.value_counts().head(5)
```

```
The Bowery Boys                 29
Totò Collection                 27
James Bond Collection           26
Zatôichi: The Blind Swordsman   26
The Carry On Collection         25
Name: belongs_to_collection, dtype: int64
```

---

5. Flattening Genres:

Extract all **genre names** from the column "genres" and **overwrite** "genres". If a movie has more than one genre, **seperate genres by a pipe** "|".
For example: The value in the first row (Toy Story) should be 'Animation|Comedy|Family'.

```
movies.genres[0]
```

```
[{'id': 16, 'name': 'Animation'},
 {'id': 35, 'name': 'Comedy'},
 {'id': 10751, 'name': 'Family'}]
```

```
# Replace the genres fields with a group of its genres separated by |

movies.genres = movies.genres.apply(lambda x: "|".join(i['name'] for i in x))

# Replace empty fields with NaN
movies.genres.replace('', np.nan, inplace = True)
```

```
movies.genres[0]
```

```
'Animation|Comedy|Family'
```

```
movies.genres.value_counts().sort_values(ascending = False).head()
```

```
Drama              5000
Comedy             3621
Documentary        2723
Drama|Romance      1301
Comedy|Drama       1135
Name: genres, dtype: int64
```

---

6. Flattening Spoken Languages (same as collections):

Extract all **spoken language names** from the column "spoken_languages" and **overwrite** "spoken_languages". If a movie has more than one spoken language, **seperate spoken languages by a pipe** "|".
For example: The value in the first row (Toy Story) should be 'English'.

```
movies.spoken_languages[0]
```

```
[{'iso_639_1': 'en', 'name': 'English'}]
```

```
movies.spoken_languages = movies.spoken_languages.apply(lambda x: "|".join(i['name'] fo

# replacing empty fields with NaN
movies.spoken_languages.replace("", np.nan, inplace = True)
```

```
movies.spoken_languages.value_counts(dropna = False).head(5)
```

```
English     22395
NaN          3958
Français     1853
□□□          1289
Italiano     1218
Name: spoken_languages, dtype: int64
```

---

7. Flattening Production Companies:

Extract all **production countries names** from the column "production_countries" and **overwrite** "production_countries". If a movie has more than one production country, **seperate production countries by a pipe** "|".
For example: The value in the first row (Toy Story) should be 'United States of America'.

```
movies.production_companies = movies.production_companies.apply(lambda x: "|".join(i['r

# replacing empty fields with NaN
movies.production_companies.replace("", np.nan, inplace = True)
```

```
movies.production_companies.value_counts(dropna = False).head(5)
```

```
NaN                                          11881
Metro-Goldwyn-Mayer (MGM)                      742
Warner Bros.                                   540
Paramount Pictures                             505
Twentieth Century Fox Film Corporation         439
Name: production_companies, dtype: int64
```

---

8. Flattening Production Countries:

Extract all **production companies names** from the column "production_companies" and **overwrite** "production_companies". If a movie has more than one production company, **seperate production companies by a pipe** "|".
For example: The value in the first row (Toy Story) should be 'Pixar Animation Studios'

```
movies.production_countries = movies.production_countries.apply(lambda x: "|".join(i['r

# replacing empty fields with NaN
movies.production_countries.replace("", np.nan, inplace = True)
```

```
movies.production_countries.value_counts(dropna = False).head(5)
```

```
United States of America    17851
NaN                          6288
United Kingdom               2238
France                       1654
Japan                        1356
Name: production_countries, dtype: int64
```

Comparing original data to cleaned so far:

```
movies.isna().sum()
```

```
belongs_to_collection    40975
budget                       0
genres                    2442
id                           0
original_language           11
overview                   954
popularity                   5
poster_path                386
production_companies     11881
production_countries      6288
release_date                87
revenue                      6
```

```
runtime                   263
spoken_languages         3958
status                     87
tagline                25054
title                      6
vote_average               6
vote_count                 6
dtype: int64
```

```
pd.read_csv("movies_metadata.csv", low_memory = False).isna().sum()
```

```
adult                      0
belongs_to_collection    40972
budget                     0
genres                     0
homepage                37684
id                         0
imdb_id                   17
original_language         11
original_title             0
overview                 954
popularity                 5
poster_path              386
production_companies       3
production_countries       3
release_date              87
revenue                    6
runtime                  263
spoken_languages           6
status                    87
tagline                25054
title                      6
video                      6
vote_average               6
vote_count                 6
dtype: int64
```

---

10. Cleaning Numerical Columns

Convert the datatype in the columns "**budget**", "**id**" and "**popularity**" **to numeric**. Set invalid values as NaN.

- No missing values in the budget column
- Some data type issues
- Revenue has few missing values
- Revenue data type is good.

```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45466 entries, 0 to 45465
Data columns (total 19 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   belongs_to_collection  4491 non-null   object 
 1   budget                 45466 non-null  object 
 2   genres                 43024 non-null  object 
 3   id                     45466 non-null  object 
 4   original_language      45455 non-null  object 
 5   overview               44512 non-null  object 
 6   popularity             45461 non-null  object 
 7   poster_path            45080 non-null  object 
 8   production_companies   33585 non-null  object 
 9   production_countries   39178 non-null  object 
 10  release_date           45379 non-null  object 
 11  revenue                45460 non-null  float64
 12  runtime                45203 non-null  float64
 13  spoken_languages       41508 non-null  object 
 14  status                 45379 non-null  object 
 15  tagline                20412 non-null  object 
 16  title                  45460 non-null  object 
 17  vote_average           45460 non-null  float64
 18  vote_count             45460 non-null  float64
dtypes: float64(4), object(15)
memory usage: 6.6+ MB
```

Cleaning up budget column:

- There are weird values like links as strings
- Must use coerce in order to be able to convert to numeric
- You can also use ignore and raise
- coerce sets invalid elements to NaN

```
movies.budget = pd.to_numeric(movies.budget, errors = "coerce")
```

```
movies.budget.value_counts().head(5)
```

```
0.0           36573
5000000.0       286
10000000.0      259
20000000.0      243
```

```
2000000.0       242
Name: budget, dtype: int64
```

---

11. Replacing zeroes with Nan:

- because zeroes affect calculations a great deal and offer nothing for accuracy, replacing them is better than leaving them

```
movies.budget = movies.budget.replace(0, np.nan)
```

---

12. Setting budget to millions scale:

- This will improve readability

```
movies.budget = movies.budget.div(1000000)
```

Fixing revenue column:

```
movies.revenue.value_counts(dropna = False)
```

```
0.0           38052
12000000.0       20
10000000.0       19
11000000.0       19
2000000.0        18
                ...
36565280.0        1
439564.0          1
35610100.0        1
10217873.0        1
1413000.0         1
Name: revenue, Length: 6864, dtype: int64
```

Following same steps as with budget for same reasons:

```
movies.revenue = movies.revenue.replace(0, np.nan)
movies.revenue = movies.revenue.div(1000000)
```

Renaming budget and revenue with millions:

```
movies = movies.rename(columns = {'revenue':'revenue_mil_usd', 'budget':'budget_mil_usd
```

```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45466 entries, 0 to 45465
Data columns (total 19 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   belongs_to_collection  4491 non-null   object 
 1   budget_mil_usd         8890 non-null   float64
 2   genres                 43024 non-null  object 
 3   id                     45466 non-null  object 
 4   original_language      45455 non-null  object 
 5   overview               44512 non-null  object 
 6   popularity             45461 non-null  object 
 7   poster_path            45080 non-null  object 
 8   production_companies   33585 non-null  object 
 9   production_countries   39178 non-null  object 
 10  release_date           45379 non-null  object 
 11  revenue_mil_usd        7408 non-null   float64
 12  runtime                45203 non-null  float64
 13  spoken_languages       41508 non-null  object 
 14  status                 45379 non-null  object 
 15  tagline                20412 non-null  object 
 16  title                  45460 non-null  object 
 17  vote_average           45460 non-null  float64
 18  vote_count             45460 non-null  float64
dtypes: float64(5), object(14)
memory usage: 6.6+ MB
```

13. Fixing zeros in runtimes:

- Having zeros in the runtime field is meaningless and can through off calculations.
- Changing zeroes to NaN will minimize that effect.

```
movies.runtime = movies.runtime.replace(0, np.nan)
```

Fixing the rest of the numeric columns:

Fixing the id and popularity columns to numeric:

- similar to the odd values in fields under budget, the id column needs to be cleaned as well
- all data in popularity should be numerical floats. There are some odd string values. So it will have to be coerced as well

```
movies.id = pd.to_numeric(movies.id, errors = 'coerce')
movies.popularity = pd.to_numeric(movies.popularity, errors = 'coerce')
```

```
movies.id.value_counts(dropna = False).head(5)
```

```
NaN         3
141971.0    3
11115.0     2
25541.0     2
15028.0     2
Name: id, dtype: int64
```

```
movies.popularity.value_counts(dropna = False).head(5)
```

```
0.000000    66
0.000001    56
0.000308    43
0.000220    40
0.000844    38
Name: popularity, dtype: int64
```

Fixing vote counts and average votes:

```
movies.vote_count.value_counts(dropna = False).head(5)
```

```
1.0    3264
2.0    3132
0.0    2899
3.0    2787
4.0    2480
Name: vote_count, dtype: int64
```

```
movies.vote_average.value_counts(dropna = False).head(5)
```

```
0.0    2998
6.0    2468
5.0    2001
7.0    1886
6.5    1722
Name: vote_average, dtype: int64
```

Examining vote average for movies where the vote count is 0:

- for films where there are no vote counts it does not make sense for the vote average to be zero
- A film could be good and just not have any votes but still not be a zero
- Replacing with NaN

```
movies.loc[movies.vote_count == 0, 'vote_average'] = np.nan
```

---

14. Cleaning `release_date` / creating datetime:

- **Convert** the datatype in the column "**release_date**" **to datetime**.
- Set invalid values as NaN.

```
movies.release_date = pd.to_datetime(movies.release_date, errors = 'coerce')
```

Release date value counts:

- It seems that January 1 is often a date used when the year of release was known, but not the entire date
- It is important to keep this in mind when working with the data

```
movies.release_date.value_counts(dropna = False).head(8)
```

```
2008-01-01    136
2009-01-01    121
2007-01-01    118
2005-01-01    111
2006-01-01    101
2002-01-01     96
2004-01-01     90
NaT            90
Name: release_date, dtype: int64
```

Cleaning Text / String Columns

15.Cleaning Text / String Columns

- **Analyze** the text columns "overview" and "tagline" and other text columns
- identify **missing data that is not represented by NaN** (e.g. "No Data").
- **Replace as NaN**

```
movies.original_language.value_counts(dropna = False).head(5)
```

```
en    32269
fr     2438
it     1529
ja     1350
de     1080
Name: original_language, dtype: int64
```

```
movies.title.value_counts(dropna = False).head(5)
```

```
Cinderella              11
Alice in Wonderland      9
Hamlet                   9
Les Misérables           8
Beauty and the Beast     8
Name: title, dtype: int64
```

Cleaning overview:

```
movies.overview[0]
```

```
"Led by Woody, Andy's toys live happily in his room until Andy's birthday brings Buzz Lightyear onto the scene. Afraid of losing his place in Andy's heart, Woody plots against Buzz. But when circumstances separate Buzz and Woody from their owner, the duo eventually learns to put aside their differences."
```

```
movies.overview.value_counts(dropna = False).head(10)
```

```
NaN
954
No overview found.
133
No Overview
7

5
No movie overview available.
3
A few funny little novels about different aspects of life.
3
Recovering from a nail gun shot to the head and 13 months of coma, doctor Pekka Valinta starts to unravel the mystery of his past, still suffering from total amnesia.
3
King Lear, old and tired, divides his kingdom among his daughters, giving great importance to their protestations of love for him. When Cordelia, youngest and most honest, refuses to idly flatter the old man in return for favor, he banishes her and turns for support to his remaining daughters. But Goneril and Regan have no love for him and instead plot to take all his power from him. In a parallel, Lear's loyal courtier Gloucester favors his illegitimate son Edmund after being told lies about his faithful son Edgar. Madness and tragedy befall both ill-starred fathers.      3
Adaptation of the Jane Austen novel.
3
Released
3
Name: overview, dtype: int64
```

```
movies.overview.replace(['No overview found.',
                         'No Overview',
                         'No movie overview available.',
                         'Released',
                         'No overview yet.',
                         " "],
                         np.nan, inplace = True)
```

Cleaning tagline:

```
movies.tagline.value_counts(dropna = False).head(10)
```

```
NaN                                   25054
Based on a true story.                    7
Trust no one.                             4
Be careful what you wish for.             4
-                                         4
Classic Albums                            3
Some doors should never be opened.        3
A Love Story                              3
Drama                                     3
Know Your Enemy                           3
Name: tagline, dtype: int64
```

```
movies.tagline.replace(["-"], np.nan, inplace = True)
```

---

16. Removing Duplicates

- each movie id should only appear once
- rows that are exact copies of a pre-existing row
- `df.duplicated(keep = False)` - check for each row whether there is a duplicate of rows
- `df.drop_duplicates(inplace = True)` = if there are duplicates, then , will be removed

```
# View duplicates
movies[movies.duplicated(keep = False)].sort_values(by = 'id').head(5)
```

	belongs_to_collection	budget_mil_usd	genres	id	original_language	overview	populari
7345	NaN	NaN	Crime|Drama|Thriller	5511.0	fr	Hitman Jef Costello is a perfectionist who alw...	9.09128

	belongs_to_collection	budget_mil_usd	genres	id	original_language	overview	populari
9165	NaN	NaN	Crime|Drama|Thriller	5511.0	fr	Hitman Jef Costello is a perfectionist who alw...	9.09128
24844	NaN	NaN	Comedy|Drama	11115.0	en	As an ex-gambler teaches a hot-shot college ki...	6.88036
14012	NaN	NaN	Comedy|Drama	11115.0	en	As an ex-gambler teaches a hot-shot college ki...	6.88036
22151	NaN	NaN	Action|Horror|Science Fiction	18440.0	en	When a comet strikes Earth and kicks up a clou...	1.43608

```
# Drop duplicate rows
movies.drop_duplicates(inplace = True)
```

Films with duplicate movie id:

```
# Find all rows with duplicate id values
movies[movies.duplicated(subset = 'id', keep = False)].sort_values(by = 'id').head(5)
```

	belongs_to_collection	budget_mil_usd	genres	id	original_language
33826	NaN	30.0	Comedy|Crime|Drama|Romance|Thriller	4912.0	en
5865	NaN	30.0	Comedy|Crime|Drama|Romance|Thriller	4912.0	en
4114	Pokémon Collection	16.0	Adventure|Fantasy|Animation|Action|Family	10991.0	ja
44821	Pokémon Collection	16.0	Adventure|Fantasy|Animation|Action|Family	10991.0	ja

	belongs_to_collection	budget_mil_usd	genres	id	original_language
44826	Pokémon Collection	NaN	Adventure|Fantasy|Animation|Science Fiction|Fa...	12600.0	ja

```
# Dropping duplicates with same movie id
# Keeping the first instance

movies.drop_duplicates(subset = 'id', inplace = True)
```

---

17. Missing Values & Removing Observations:

- Drop all rows/movies with unknown id or title.
- options for other missing data:

```
    - do nothing and leave as they are
    - completely remove rows or columns with missing data over a certain amount
    - replacing missing values with appropriate alternatives (usually used just for m
```

df.isna().sum

- get sum of missing values in each column

```
movies.isna().sum()
```

```
belongs_to_collection    40946
budget_mil_usd           36554
genres                    2442
id                           1
original_language           11
overview                  1105
popularity                   4
poster_path                386
production_companies     11872
production_countries      6283
release_date                88
revenue_mil_usd          38036
runtime                   1819
spoken_languages          3954
status                      85
tagline                  25037
title                        4
vote_average              2900
```

```
vote_count                    4
dtype: int64
```

Removing any rows with blank title or id:

```
# drops all rows that have at least one missing value in the subset column(s)
movies.dropna(subset = ['id', 'title'], inplace = True)
```

Converting id column to int:

- as long as there is any missing value in a numerical column, the type will remain float
- since the missing values have now been dealt with, the column can be converted to int

```
movies.id = movies.id.astype('int')
```

---

18. Remove rows with too many missing values:

Keep only those rows/movies in the df with **10 or more non-NaN** values.

df.notna().sum()

- Check the number of values are valid
- total columns is 18

```
# how many rows have the specified number of valid values out of 18
movies.notna().sum(axis = 1).value_counts().sort_values(ascending = False)
```

```
15    12522
16    11454
14     5424
17     4265
18     3859
13     3040
12     1891
19     1132
11     1020
10      511
9       184
8       104
7        20
6         4
dtype: int64
```

```
# rows that only have 6 / 18 valid values
movies[movies.notna().sum(axis = 1) == 6]
```

	belongs_to_collection	budget_mil_usd	genres	id	original_language	overview	popularity	poster_path	p
18038	NaN	NaN	NaN	344741	pl	NaN	0.000000	NaN	
20166	NaN	NaN	NaN	139909	fi	NaN	0.000127	NaN	
41718	NaN	NaN	NaN	216550	de	NaN	0.000000	NaN	
44978	NaN	NaN	NaN	398295	pt	NaN	0.000331	NaN	

Remove films with fewer than 10 non-missing values:

- `df.dropna(threah = 10)` - removes all rows with fewer than 10 valid values

```
movies.dropna(thresh = 10, inplace = True)
```

```
print('The database still has ', len(movies), 'films in it.')
```

```
The database still has  45118 films in it.
```

---

19. Removing status column:

- **Keep** only those rows/movies in the df with **status** "**Released**". Then **drop** the column "status".

```
movies.status.value_counts()
```

```
Released           44691
Rumored              226
Post Production       98
In Production         20
Planned               15
Canceled               2
Name: status, dtype: int64
```

```
movies = movies.loc[movies.status == 'Released'].copy()
```

```
movies.status.value_counts()
```

```
Released    44691
Name: status, dtype: int64
```

```
movies.drop(columns = ['status'], inplace = True)
```

---

20. Reorder columns:

```
columns = ["id", "title", "tagline", "release_date", "genres",
           "belongs_to_collection", "original_language", "budget_mil_usd",
           "revenue_mil_usd", "production_companies", "production_countries",
           "vote_count", "vote_average", "popularity", "runtime",
           "overview", "spoken_languages", "poster_path"]
```

```
movies = movies.loc[:, columns]
```

---

21. **Reset** the index / create a **RangeIndex.**

```
movies.reset_index(drop = True, inplace = True)
```

---

22. Cleaning poster_path:

- The paths listed in the database are fragments of paths
- Adding the base_path to the listed path

```
movies.poster_path[0]
```

```
'/rhIRbceoE9lR4veEXuwCC2wARtG.jpg'
```

```
base_poster_url = 'http://image.tmdb.org/t/p/w185/'

movies.poster_path = "<img src='" + base_poster_url + movies.poster_path + "' style='he
```

---

23. **Save** the cleaned dataset in a **csv-file.**

```
movies.to_csv('movies_clean.csv', index = False)
```

```
pd.read_csv('movies_clean.csv').info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 44691 entries, 0 to 44690
Data columns (total 18 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   id                     44691 non-null  int64
 1   title                  44691 non-null  object
 2   tagline                20284 non-null  object
```

```
 3   release_date           44657 non-null  object
 4   genres                 42586 non-null  object
 5   belongs_to_collection  4463 non-null   object
 6   original_language      44681 non-null  object
 7   budget_mil_usd         8854 non-null   float64
 8   revenue_mil_usd        7385 non-null   float64
 9   production_companies   33356 non-null  object
 10  production_countries   38835 non-null  object
 11  vote_count             44691 non-null  float64
 12  vote_average           42077 non-null  float64
 13  popularity             44691 non-null  float64
 14  runtime                43179 non-null  float64
 15  overview               43740 non-null  object
 16  spoken_languages       41094 non-null  object
 17  poster_path            44467 non-null  object
dtypes: float64(6), int64(1), object(11)
memory usage: 6.1+ MB
```