DS2000
Fall 2021
Handout: JSON format and APIs

JSON format

Like comma-separated value (CSV) format, JSON format is structured, predictable, and relatively easy to work with. If you end up with a dataset that's in JSON format, you've generally got a lot less cleaning up to do than if your dataset came from a messy text file.

JSON-formatted data could come your way in the form of a file (`*.json`) or be returned from an API function you call. Either way, JSON data is formatted like a Python dictionary, so that's the best data structure to use for it.

JSON formatted-data example of movie genres that I got from a website called themoviedb.

```
{"genres": [  {"id":28,"name":"Action"},
              {"id":12,"name":"Adventure"},
              {"id":16,"name":"Animation"},
              {"id":35,"name":"Comedy"} ]}
```

genres.json

So what is this structure? It's definitely a dictionary, but it has some other layers too. The overarching dictionary has only one key, a string ("genres").

```
{"genres": [  {"id":28,"name":"Action"},
              {"id":12,"name":"Adventure"},
              {"id":16,"name":"Animation"},
              {"id":35,"name":"Comedy"} ]}
```

genres.json with the key of the overall dictionary highlighted

The overarching dictionary also has only one value, which is a list of dictionaries:

```
{"genres": [  {"id":28,"name":"Action"},
              {"id":12,"name":"Adventure"},
              {"id":16,"name":"Animation"},
              {"id":35,"name":"Comedy"} ]}
```

genres.json with the value of the overall dictionary highlighted

Processing JSON Files

Sometimes the data that you download from a website, if you're lucky, is in a nice structured format like CSV or JSON. We already know how to process a CSV file and read it into a nested list. Processing a JSON file is similar, except we read the contents into a dictionary.

Here's how we'd process the file *genres.json* above. We'd just need to import the JSON module, and then we use the same with/open/as structure we already know. The piece that's new is the function json.load, which takes everything in the JSON file and saves it into a Python dictionary.

```
import json

def process_json(filename):
    ''' Function process_json
        Parameter: Name of the json file, a string
        Returns: Dictionary containing everything from the file
    '''
    with open(filename) as json_file:
        dct = json.load(json_file)
        return dct
```

process_json -- a function to open and read the contents of a JSON file into a Python dictionary

Once we've read from the JSON file into a dictionary, we can process it just like any other dictionary. At this point, we have all the data in the data structure we like, and what we do with it depends completely on the goals of your program.

Continuing with the movie genres example, if we just want to print the names of each genre, ignoring ID numbers and any other nonsense, we can call our *process_json* function from our driver and then iterate over the list of dictionaries we'll end up with:

```
if __name__ == "__main__":
    # This file contains a dictionary, where
    # the key is 'genres' and the value is a list of dictionaries
    all_data = process_json('genres.json')
    dcts = all_data['genres']

    # Print out just the genre names
    for dct in dcts:
        print(dct['name'])
```

A driver to call the process_json function and print out the names of the movie genres

The variable *all_data* will perfectly mirror the contents of the file *genres* -- it'll be a dictionary with only one key ("genres") and one value (a list of dictionaries).

Then we take it apart. In the function above, the variable *dcts* will contain the list of dictionaries:

```
[  {"id":28,"name":"Action"},
   {"id":12,"name":"Adventure"},
   {"id":16,"name":"Animation"},
    {"id":35,"name":"Comedy"} ]
```

Contents of the variable dcts

Finally, in the above driver we iterate over this list of dictionaries and print out each name. The driver will eventually print something like this:

```
Action
Adventure
Animation
Comedy
```

Output of the driver that calls process_json and iterates over the list of dictionaries, printing each genre name

Calling an API

An Application Programming Interface (API) allows us to call a function provided by a company to retrieve some of their data. It's a faster, more-programmatic version of manually downloading data files from a website like we did for HW6. It's usually more up-to-date as well; you don't need to wait for someone at the company to upload new files.

The format of data you gather from an API is often JSON. But now, the data is returned from a function call, rather than downloaded in a file.

API Example -- themoviedb.org

The JSON file above comes from https://www.themoviedb.org/. I can get the same data by calling their API; that way there's no file to download and I just have everything already in a variable waiting for me.

API Key

Whenever you're pulling data from a company's API, they usually require that you set up an API key first. It's unique to you, and it helps them track who is calling their API functions. Every company will be different, but for *themoviedb.org,* I got an API key by taking following steps:

1. Created an account
2. Logged in and went to my account
3. Clicked on settings/API
4. Filled out the form to request an API key

The API key is used every time you call one of the functions to retrieve data.

Calling an API Function in Python

Once I got my API key, I read the documentation on https://developers.themoviedb.org/3, which described the name, parameters, and return type for all of their functions -- just what we need to call them, of course!

Let's call the following function in Python:

- **Name**: genre/movie/list
- **Parameters**: api_key, language
- **Returns**: A dictionary of lists, all the genres that the website links with movies

(Even though we want to call this function in our Python code, you can actually test it out in your browser, by typing this into your address bar; this can be useful because you can see what kinds of things the API function returns without adding a potential complication of actual code:
`https://api.themoviedb.org/3/genre/movie/list?api_key=<<Laney's API Key>>&language=en-US_`

We'll need to import Python's *requests* library, so put this at the top of your .py file.

When we're just requesting information and not trying to modify anything it's called a **get request.** (E.g., in this example we're collecting information about the all the movie genres; we're **not** trying to, say, add our review to their database.) We'll use a function called *requests.get* and make our function call that way.

```
import requests

def get_genres(api_key):
    ''' Function: get_genres
        Parameters: api_key, a string
        Returns: dictionary, returned by the genres function
    '''
    http_params = {'api_key':api_key,
                   'language':'eng-US'}
    response = requests.get('https://api.themoviedb.org/3/genre/movie/list',
                            params = http_params)
    return response.json()
```

get_genres function. We're getting the same movie-genre data as before, except now we generate it by calling a function instead of downloading a file.

Let's break down these pieces:

- `http_params`. We've made a dictionary of our own; this one specifies the parameters we know the function expects: api_key and language.
- `requests.get`. We're calling a function that asks the API function for information and not modifying anything, so we use a get request.
- `response.json()`. The response given initially is not in JSON format, but it's pretty close. We make one last function call here to transform the http response into JSON.

What does this function return? The **exact same thing** as the first version that loaded a JSON file! Same dictionary, same contents. We can iterate over the result in the exact same way to print out the genre names.

Other APIs

Many companies make some of their data available via API calls, but it's also quite a lot of work for programmers to write an API so you definitely can't count on it. Worth looking for, though, as it's often a better option than manually downloading files from a website.

If you use an API in your DS2001 project or any data science endeavor in the future, it'll be set up similar to what we're describing here, but every company/organization is different. Most companies that have APIs are pretty good about documenting them, so start by reading up on what their available functions are and how they work. Here are some examples, we tried to grab them from all different kinds of areas.

- Google https://developers.google.com/apis-explorer/#p/
- data.gov https://www.data.gov/developers/
- Twitter https://developer.twitter.com/en/docs.html
- NOAA https://www.ncdc.noaa.gov/cdo-web/webservices/v2
- MLB https://appac.github.io/mlb-data-api-docs/
- Veterans' Affairs Health data: https://developer.va.gov/explore/health
- Yahoo Finance (stock market): https://rapidapi.com/apidojo/api/yahoo-finance1
- Spotify: https://spotipy.readthedocs.io/en/2.9.0/