

QSS20: Modern Statistical Computing

Unit 10: APIs

Goals for today

- ▶ Recap of text as data
- ▶ API: terminology and basics
- ▶ Example 1: API with no credentials and no wrapper (NAEP data API)
- ▶ Example 2: API with credentials and no wrapper (Yelp API)
- ▶ Example 3: API with credentials and wrapper (wrapper for Twitter API)

Goals for today

- ▶ **Recap of text as data**
- ▶ API: terminology and basics
- ▶ Example 1: API with no credentials and no wrapper (NAEP data API)
- ▶ Example 2: API with credentials and no wrapper (Yelp API)
- ▶ Example 3: API with credentials and wrapper (wrapper for Twitter API)

Recap of text as data

What do you remember?

Review of key terms/jargon/acronyms in text as data

- ▶ *Natural Language Processing (NLP)*: Science of processing language to extract signal, find patterns, connect with social world
- ▶ *Tokens*: Units of language, e.g., words, bigrams, phrases, punctuation
"I like Dartmouth" (string) -> ["I", "like", "Dartmouth"] (list of str)
- ▶ *Supervised methods*: Guided by pre-existing knowledge, often a list of words or doc labels
- ▶ *Unsupervised methods*: Inductively discover patterns in text data, e.g. topics, key words
- ▶ *Part of Speech (POS) tagging*:

```
tokens = word_tokenize(text) # Tokenize  
tokens_pos = pos_tag(tokens) # get POS tags
```
- ▶ *Named Entity Recognition (NER)*: NLP pipeline for identifying named entities, e.g. people, places

```
spacy_obj = nlp(text) # run NER pipeline  
[entity.label_ for entity in spacy_obj.ents] # get tags
```
- ▶ *Sentiment analysis*: Use dictionary to score positive/negative "feel" at document level
`SentimentIntensityAnalyzer().polarity_scores(text)`
- ▶ *Document-Term Matrix (DTM)*: each row is a doc, each col is a term, values are 0, 1, ... n for # of times that term occurs in that doc
- ▶ *Latent Dirichlet Allocation (LDA)*: Generative model of language based on idea that language comes from "topics". LDA estimates topics (mixture of words) and their distro over docs (mixture of topics).
`ldamodel.LdaModel(corpus_word_map, num_topics=5, id2word=text_raw_dict)`

Where we are

- ▶ Recap of text as data
- ▶ **API: terminology and basics**
- ▶ Example 1: API with no credentials and no wrapper (NAEP data API)
- ▶ Example 2: API with credentials and no wrapper (Yelp API)
- ▶ Example 3: API with credentials and wrapper (wrapper for Twitter API)

Terminology

- ▶ **API:** application programming interface; way to ask an app or website for something and get something in return
- ▶ **Call the API:** sending a request for something to the API
- ▶ **Response:** can think of this as a message back telling us *whether* we got something back or whether the call returned an error
- ▶ **JSON:** if we get something back, oftentimes it'll be stored in json format, which is basically a text string with a particular structure that is similar to the *data structure* of a dictionary; can pretty easily convert to a pandas dataframe
- ▶ **Wrapper:** a language-specific module or package that helps simplify the process of calling an API with code written in a particular language (e.g., later we'll review tweepy, a Python wrapper for the Twitter API; there are also R wrappers for the Twitter API)

Main use in our context: data acquisition

Three general routes to acquiring data:

Exists already:

Great first step to check out, can save a lot of time if matches your research question; examples include the csv/excel data we've been working with for problem sets

API:

A “front door” to a website, where the developers provide easy access to content but also set limits (e.g., what content or how much); most relevant for “high-velocity” data that changes frequently (e.g., tweets; job postings) and also for using code rather than point/click to get data

Scraping:

A “back door” to web content for when there's no API or when we need content beyond the structured fields the API returns; can be time-consuming and code can break if website structure changes (can happen often)

Why go through the effort to use data that doesn't exist already?

I got the data for my thesis from web scraping, **which I mostly learned from Google**. I highly recommend learning how to scrape website information. It is often a guarantee that you are working with original data, which means you are uncovering an original thesis topic (Source: https://qss.dartmouth.edu/sites/program_quantitative_social_science/files/zach-schnell-thesis-advice.pdf)

Where we are

- ▶ Recap of text as data
- ▶ API: terminology and basics
- ▶ **Example 1: API with no credentials and no wrapper (NAEP data API)**
- ▶ Example 2: API with credentials and no wrapper (Yelp API)
- ▶ Example 3: API with credentials and wrapper (wrapper for Twitter API)

High-level overview of steps: APIs that don't need credentials

1. Construct a query that tells the API what we want to pull
2. Use `requests.get(querystring)` to call the API
3. Examine the response: message from the API telling us whether it returned something
4. If the response returned something, extract the content of the response and make it usable

Step 1: construct a query

- ▶ Generic example:
“<https://baseurl.com/one thing=something&another thing=something else>”
- ▶ Specific NAEP example (use the (syntax to split across lines)

```
1 example_naep_query = (  
2 'https://www.nationsreportcard.gov/'  
3 'Dataservice/GetAdhocData.aspx?'  
4 'type=data&subject=writing&grade=8&  
5 'subscale=WRIRP&variable=GENDER&',  
6 'jurisdiction=NP&stattype=MN:MN&',  
7 'Year=2011')
```

- ▶ Breaking things down:
 - ▶ nationsreportcard: this is the “base url” we’re using for the API call and what we add parameters to
 - ▶ subject: type of parameter
 - ▶ subject=writing: specific value for that parameter (error if we feed it a subject that doesn’t exist)
 - ▶ And so on...

Steps 2-4: call the API, examine the response, and if response indicates something usable, extract content

```
1 ## use requests.get to call the API
2 naep_resp = requests.get(example_naep_query)
3
4 ## we got usable response, so get json of status and
   result
5 naep_resp_j = naep_resp.json()
6
7 ## extract contents in `result` key
8 ## and convert to dataframe
9 naep_resp_d = pd.DataFrame(naep_resp_j['result'])
```

What do I mean by “no wrapper”?

- ▶ We write a query to request something from the API
- ▶ While the request syntax differs across languages, the query is the same— eg could use same query and run below R code to get content

```
1 ## packages
2 library(httr)
3 library(jsonlite)
4
5 ## ping API
6 return_q = GET(example_naep_query)
7
8 ## get data from that ping
9 data = fromJSON(rawToChar(return_q$content))$result
```

Activity 1: practice pulling data using the NAEP API

Notebook: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/08_apis_partI_blank.ipynb

- ▶ Example of executing a query that doesn't have errors
- ▶ Example of executing a query that returns nothing
- ▶ Working together to write a function to do multiple calls to the API

Where we are

- ▶ Recap of text as data
- ▶ API: terminology and basics
- ▶ Example 1: API with no credentials and no wrapper (NAEP data API)
- ▶ **Example 2: API with credentials and no wrapper (Yelp API)**
- ▶ Example 3: API with credentials and wrapper (wrapper for Twitter API)

What changes about the general steps?

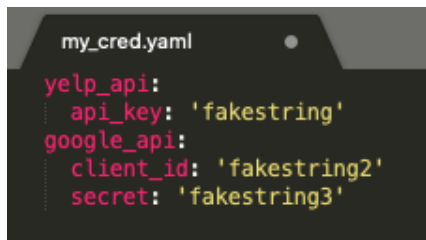
1. Acquire credentials for the API: these may be an API key (single string) or a client ID and secret (strings; can store in a `yaml` creds file that I'll outline)
2. Construct a query that tells the API what we want to pull
3. Two paths:
 - 3.1 **Use credentials to authenticate and then call the API:** we'll later see example of this with Twitter API/wrapper
 - 3.2 **Feed API your credentials when you call the API:** we'll see example of this with Yelp
4. Examine the response: message from the API telling us whether it returned something
5. If the response returned something, extract the content of the response and make it usable

Step 1: acquire credentials

- ▶ Varies across APIs, but general involves going to the “developer’s portal,” creating an account, and obtaining credentials
- ▶ Examples:
 - ▶ Google developer’s console (things like google geocoding API; maps API): <https://console.cloud.google.com>
 - ▶ Facebook: <https://developers.facebook.com/docs/development>
 - ▶ Twitter (via Tweepy wrapper):
https://docs.tweepy.org/en/latest/auth_tutorial.html
 - ▶ Yelp: <https://www.yelp.com/developers/documentation/v3/authentication>
- ▶ Note weird-ish terminology for social science applications since you often set up “an application” in order to get credentials (but we’re often doing a one-way pull of data and not developing an app. that repeatedly calls it)

Step 1: store those credentials somewhere

- ▶ Your key or client ID/secret are meant to be unique to you like a password, so you shouldn't generally print in code
- ▶ Can use any text editor to make a yaml file (structured similar to a dictionary); screenshot below from Sublime text with fake credentials



```
my_cred.yaml
yelp_api:
  api_key: 'fakestring'
google_api:
  client_id: 'fakestring2'
  secret: 'fakestring3'
```

Step 1: load the file with credentials

```
1 ## imports
2 import yaml
3
4 ## load creds
5 with open("../.. / private_data / my_cred .yaml" , 'r') as
   stream:
6     try:
7         creds = yaml.safe_load(stream)
8     except yaml.YAMLError as exc:
9         print(exc)
10
11 ## can then get the relevant key
12 creds['yelp_api']['api_key']
```

Step 2: construct a query

Same exact process as before; here focusing on **Yelp Fusion API**; API has different endpoints shown in the screenshot; we'll initially focus on Business Search, since that returns a Yelp-specific ID (https://www.yelp.com/developers/documentation/v3/get_started)

Name	Path	Description
Business Search	/businesses/search	Search for businesses by keyword, category, location, price level, etc.
Phone Search	/businesses/search/phone	Search for businesses by phone number.
Transaction Search	/transactions/{transaction_type}/search	Search for businesses which support food delivery transactions.
Business Details	/businesses/{id}	Get rich business data, such as name, address, phone number, photos, Yelp rating, price levels and hours of operation.
Business Match	/businesses/matches	Find the Yelp business that matches an exact input location. Use this to match business data from other sources with Yelp businesses.
Reviews	/businesses/{id}/reviews	Get up to three review excerpts for a business.
Autocomplete	/autocomplete	Provide autocomplete suggestions for businesses, search keywords and categories.

Step 2: construct a query

```
1 ## defining inputs
2 base_url = "https://api.yelp.com/v3/businesses/search?"
3 my_name = "restaurants"
4 my_location = "Hanover,NH,03755"
5
6 ## combining them into a query
7 yelp_genquery = (
8     '{base_url}'
9     'term={name}'
10    '&location={loc}').format(
11        base_url = base_url,
12        name = my_name,
13        loc = my_location)
```

Step 3: authenticate and call the API

For Yelp, we feed a dictionary with our key directly into the get call via the optional `header` parameter; for other APIs, we sometimes authenticate in a separate step

```
1 ## construct my http header dict
2 header = {'Authorization': f'Bearer {API_KEY}'}
3
4 ## call the API
5 yelp_genresp = requests.get(yelp_genquery, headers =
    header)
```

Step 3: output of successful and unsuccessful call

- Successful call:

<Response [200]>

- Unsuccessful call (put Hanover,WY,09999 as the location, which doesn't exist)

<Response [400]>

```
{'error': {'code': 'LOCATION_NOT_FOUND',  
  'description': 'Could not execute search, try specifying a more exact location.'}}
```

Step 4: if output successful, make results usable

See that 'businesses' key of json file has a dictionary for each business, but some nesting to deal with variable lengths (e.g., within 'location', 'address1', 'address2', etc.) that might produce odd things when we concat. to a df:

```
{'id': '8ybF6YyRldtZmU9jil4xlg',
 'alias': 'mollys-restaurant-and-bar-hanover',
 'name': "Molly's Restaurant & Bar",
 'image_url': 'https://s3-media2.fl.yelpcdn.com/bphoto/1YkJFic4Czt9b2FsZyOrwQ/o.jpg',
 'is_closed': False,
 'url': 'https://www.yelp.com/biz/mollys-restaurant-and-bar-hanover?adjust_creative=Ag=yelp_api_v3&utm_medium=api_v3_business_search&utm_source=ABQTB3e9fTiSiyqs0c-3Bg',
 'review_count': 403,
 'categories': [{'alias': 'tradamerican', 'title': 'American (Traditional)'},
 {'alias': 'burgers', 'title': 'Burgers'},
 {'alias': 'pizza', 'title': 'Pizza'}],
 'rating': 4.0,
 'coordinates': {'latitude': 43.701144, 'longitude': -72.2894249},
 'transactions': ['delivery'],
 'price': '$$',
 'location': {'address1': '43 South Main St',
 'address2': '',
 'address3': '',
 'city': 'Hanover',
 'zip_code': '03755',
 'country': 'US',
 'state': 'NH',
 'display_address': ['43 South Main St', 'Hanover, NH 03755']},
 'phone': '+16036432570',
 'display_phone': '(603) 643-2570',
 'distance': 250.8301601841674}
```

Approach 1 to step 4: more automatic `pd.DataFrame` that leaves those as lists

```
1 yelp_gendf = pd.DataFrame(yelp_genjson[ 'businesses ' ])
```

Approach 2 to step 4: only retaining columns that are already strings

```
1 def clean_yelp_json(one_biz):
2
3     ## restrict to str cols
4     d_str = {key:value for key, value in one_biz.items()
5               if type(value) == str}
6
7     df_str = pd.DataFrame(d_str, index = [d_str['id']])
8     return(df_str)
9
10 yelp_stronly = [clean_yelp_json(one_b)
11                 for one_b in yelp_genjson['businesses']]
12 yelp_stronly_df = pd.concat(yelp_stronly)
```

Activity 2: practice with the Yelp API

Same url: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/08_apis_partI_blank.ipynb

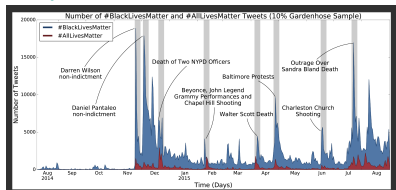
- ▶ Try running a business search query for your hometown or another place by constructing a query similar to 'yelp_genquery' but changing the location parameter
- ▶ Other endpoints require feeding what's called the business' fusion id into the API. Take an id from 'yelp_stronly.id' and use the documentation here to pull the reviews for that business:
https://www.yelp.com/developers/documentation/v3/business_reviews
- ▶ **Challenge:** generalize the previous step by writing a function that (1) takes a list of ids as an input, (2) calls the reviews API for each id, (3) returns the results, and (4) rowbinds all results

Where we are

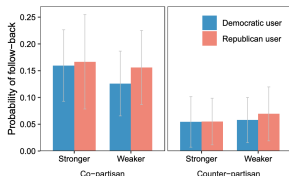
- ▶ Recap of text as data
- ▶ API: terminology and basics
- ▶ Example 1: API with no credentials and no wrapper (NAEP data API)
- ▶ Example 2: API with credentials and no wrapper (Yelp API)
- ▶ **Example 3: API with credentials and wrapper (wrapper for Twitter API)**

Academic research using Twitter data

- Gallagher et al. 2018. “Divergent discourse between protests and counter-protests: BlackLivesMatter and AllLivesMatter”



- Mosleh et al. 2021. “Shared partisanship dramatically increases social tie formation in a Twitter field experiment”



- And many more!

Two routes

1. **API as the “front door” route:** controlled access to Twitter content; see discussion of access tiers here:
<https://developer.twitter.com/en/docs/twitter-api/getting-started/about-twitter-api#v2-access-level>; we're using the most basic tier (essential); if want to do more, apply for extended or academic tier
2. **Scraping as the “back door” route:** messier and need to deal with pagination, site blocking API calls from your IP address, etc.

Tweepy: wrapper for Twitter API

- ▶ In previous examples with NAEP and Yelp we:
 1. Set up credentials (where relevant)
 2. Constructed a query based on the API documentation
 3. Used `requests.get(query)` to call the API
 4. Got a response
 5. Extracted the content of the response and turned it into usable data
- ▶ Wrappers can simplify those steps by simplifying the process of **calling the API**
- ▶ Instead of a long string with a complex query, can feed methods within the wrappers arguments that specify what we want to pull
- ▶ **Downside:** need to understand how to structure functions within that wrapper (but can be easier than constructing queries yourself)
- ▶ Documentation for **tweepy** (one python-based wrapper):
<https://docs.tweepy.org/en/latest/index.html>

High-level overview of steps

1. Acquire credentials for the API: see Canvas message; for our purposes, we'll just be using the `bearer_token`

2. Use those credentials to establish a connection to the Twitter API:

```
client = tweepy.Client(bearer_token=
                        creds['twitter_api']['bearer_token'])
```

3. Read the documentation here to learn about different methods for pulling information about tweets, engagement, and users:

[https:](https://dev.to/twitterdev/a-comprehensive-guide-for-using-the-twitter-api-v2-using-tweepy-in-python-15d9)

[//dev.to/twitterdev/a-comprehensive-guide-for-using-the-twitter-api-v2-using-tweepy-in-python-15d9](https://dev.to/twitterdev/a-comprehensive-guide-for-using-the-twitter-api-v2-using-tweepy-in-python-15d9)

4. Use a method to call the API and return a response- generic setup:

```
tweet_resp = client.some_method(some_args...)
```

5. Previous step just returns a tweepy client response (message back); similar to other examples, need to extract **content** of response; instead of via `.json()`, use `.data` attribute, which returns a list:

```
tweet_data = tweet_resp.data
```

6. Extract relevant information from that list (e.g., tweet content; tweet's unique id; creation date), and transform into a dataframe

Examples of three things we can do (even on the most limited “essential” tier)

1. **Pull tweets associated with a hashtag and attributes of people tweeting**
2. Examine connections between different accounts via follower/following relationships
3. Pull tweets from a specific user and examine others' engagement with those tweets

Follow along: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/solutions/08_apis_twitter_examplecode.ipynb

Step 1: call the API and pull recent tweets

```

1 ## construct a query (see notebook for guide)
2 query = "#metoo -is:retweet is:verified"
3 ## use the search_recent_tweets method
4 tweets_mt = client.search_recent_tweets(query = query,
5                                         max_results = 100,
6                                         tweet_fields = ['created_at', 'author_id', 'geo',
7                                                         'lang', 'public_metrics'],
8                                         user_fields = ['description', 'location',
9                                                         'verified', 'public_metrics'],
10                                         expansions = 'author_id',
11                                         end_time = "2022-02-15T01:00:00-00:00")

```

Breaking things down:

- ▶ query: what we'd type into the twitter search bar if using site
- ▶ search_recent_tweets(): method in tweepy client class to pull tweets from the last 7 days
- ▶ tweet_fields: argument that takes a list of metadata we want to pull about the tweet (full list here: <https://developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/tweet>)
- ▶ user_fields: argument that takes a list of metadata we want to pull about user tweeting (full list here: <https://developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/user>)
- ▶ expansions: helps us connect tweet authors to that user metadata

Step 2: extract info. about tweets (printing info)

First example: print the first 5 results

```

1 tweet_res = [print("""On {}, {} tweeted {} in {} language,
2               which was liked by {}""".format(tweet.created_at,
3               tweet.author_id, tweet.text, tweet.lang,
4               tweet.public_metrics['like_count']))
5               for tweet in tweets_mt.data[0:5]]

```

- ▶ `for tweet in tweets_mt.data:`
 - ▶ `tweets_mt` is the response from the previous step
 - ▶ `.data` gets a list containing its content
 - ▶ We then iterate over list items, where each item is a single tweet that matches the query (tweets about metoo from verified users)
- ▶ `tweet.author_id`, `tweet.text`, and so on: attributes of each tweet; single values
- ▶ `tweet.public_metrics`: rather than a single value, an attribute that's a dictionary of different tweet metrics, where the key is the type of metric (e.g., likes; retweets) and value is count for focal tweet

Previous example just printed things; what if we want to store them in a dataframe?

Step 2: extract info. about tweets (storing info)

```

1 ##### define the attributes (need to be pulled
2 ##### in the tweet_fields arg of search_recent_tweets call above)
3 tweet_attr = ['id',
4               'created_at', 'author_id',
5               'text', 'lang', 'geo', 'public_metrics']
6
7 ##### function to iterate over attributes
8 def pull_attr(one_tweet, which_attr):
9     all_attr = [one_tweet[attr] if attr != 'public_metrics'
10                else one_tweet[attr]['like_count']
11                for attr in which_attr]
12     return(all_attr)
13
14 ##### iterate over tweets and pull tweet info
15 tweets_info_list = [pull_attr(one_tweet, tweet_attr)
16                     for one_tweet in tweets_mt.data]
17
18 ##### transform into a dataframe
19 tweets_info_df = pd.DataFrame(tweets_info_list,
20                               columns = tweet_attr)

```

Output of previous step (with most likes)

created_at	author_id	text	lang	geo	public_metrics
2022-02-13 4:28:24+00:00	2835451658	South Korea: The wife of a top presidential candidate has caused controversy after saying she'd jail journalists & suggested #MeToo women are doing it for the money. Despite the comments in a fiery interview, her husband's polling has actually increased. https://t.co/Ex6hRD1peL	en	None	900
2022-02-13 3:10:45+00:00	453704729	I'm sending love to @sistadbarnes today and wondering: \n\nWhy are we still platforming unrepentant abusers? Does #MeToo matter? \n\nAnd if #BlackLivesMatter why did artists in the #SuperBowl HalfTimeShow take a knee with @Kaepernick7 only to rock the mic for an unrepentant NFL? https://t.co/oNvF5iJSk6	en	None	256

See that it's missing key info on the users who are tweeting these statements!

Step three: extract info. about users who tweeted those tweets

```
1 ##### list with user ids for relevant tweets
2 users = {user["id"]: user for user in tweets_mt.includes['users']}
3
4 ##### define the user attributes
5 user_attr = ['username', 'description',
6              'location', 'verified', 'public_metrics']
7
8 ##### function to iterate over user attributes
9 def pull_user_attr(one_tweet, which_attr):
10     one_user = users[one_tweet.author_id]
11     all_attr = [one_tweet.author_id] + [one_user[attr]
12     if attr != "public_metrics"
13         else one_user[attr]['followers_count']
14         for attr in which_attr]
15     return(all_attr)
16
17 ##### iterate over tweets and execute to pull user info
18 users_info_list = [pull_user_attr(one_tweet, user_attr)
19                     for one_tweet in tweets_mt.data]
20 user_info_df = pd.DataFrame(users_info_list,
21                              columns = ['author_id'] + user_attr)
```


Can then merge on author_id to get annotated tweets and users

text	public_metrics_tweet	username	description	public
South Korea: The wife of a top presidential candidate has caused controversy after saying she'd jail journalists & suggested #MeToo women are doing it for the money. Despite the comments in a fiery interview, her husband's polling has actually increased. https://t.co/Ex6hRD1peL	900	MrAndyNgo	Independent journalist & author of NYT bestseller, 'Unmasked.' Editor-at-large: @TPostMillennial. Contact, support, follow: https://t.co/6ZvZk6Gwgx	
I'm sending love to @sistadbarnes today and wondering: \n\nWhy are we still platforming unrepentant abusers? Does #MeToo matter? \n\nAnd if #BlackLivesMatter why did artists in the #SuperBowl HalfTimeShow take a knee with @Kaepernick7 only to rock the mic for an unrepentant NFL? https://t.co/oNvF5i.lSk6	256	deardrewdixon	Silence Breaker. Producer. Writer. Mom. She/Her/Hers. As seen in @OnTheRecordDoc Press Requests: ryanmaziepr@gmail.com	

Examples of three things we can do

1. Pull tweets associated with a hashtag and attributes of people tweeting
2. **Examine connections between different accounts via follower/following relationships**
3. Pull tweets from a specific user and examine others' engagement with those tweets

Network structure of twitter

- ▶ For each focal “user”—e.g., Oprah; Dartmouth College—there are followers of that user
- ▶ Similarly, each follower of a user has followers
- ▶ We’re going to explore connections to a focal account and their 2nd-degree connections (though building a network would require access beyond essential endpoint; since we’re limited to most recent 100 followers)

Step one: get id for a username and pull their followers

```
1 ##### pull numeric id for a user name
2 focal_acc_id = client.get_user(username =
3     "MrAndyNgo").data['id']
4
5 ##### use the get_users_followers method
6 ##### to pull followers (pulling same user
7 ##### metadata as on previous slide)
8 follow_focal = client.get_users_followers(id =
9     focal_acc_id ,
10     user_fields = user_attr)
```

Breaking this down:

- ▶ `get_user()`: method to pull metadata about one username; we're then extracting the contents (`.data`) and pulling the id
- ▶ `get_users_followers()`: feed this method the numeric id and user fields we want to pull (using list from previous step with description, verified status, etc.); similar to other step, it returns an API response we need to extract contents of

Step two: extract content of response and put in a dataframe

```

1 user_data = pd.DataFrame(
2     {'uname': [user['username']]
3     for user in follow_focal.data],
4     'description': [user['description']]
5     for user in follow_focal.data],
6     'user_id': [user['id']]
7     for user in follow_focal.data],
8     'followers': [user['public_metrics']['followers_count']]
9     for user in follow_focal.data})

```

Breaking this down:

- ▶ Iterate over the content of the response (`follow_focal.data`) and pull different fields
- ▶ See `pull_attr` on earlier slide for a less manual way that iterates over metadata fields
- ▶ Can then repeat to get followers of follower (see notebook)

Examples of three things we can do

1. Pull tweets associated with a hashtag and attributes of people tweeting
2. Examine connections between different accounts via follower/following relationships
3. **Pull tweets from a specific user and examine others' engagement with those tweets**

Pull tweets from a specific user

```

1 ## step 1: choose a focal account and get their numeric id
2 focal_account = "SenatorHassan"
3 get_id = client.get_user(username= focal_account , user_fields =
4                             user_attr)
5 hassan_id = get_id.data['id']
6 ## step 2: use the get_users_tweets method to
7 ## pull recent tweets — here, i'm pulling most recent 100
8 hassan_tweets_resp = client.get_users_tweets(id = hassan_id ,
9                                               max_results = 100, tweet_fields = tweet_attr)
10 ## step 3: that returns a response with data as an attribute
11 ## to turn into a dataframe, use function above
12 hassan_tweets_list = [pull_attr(one_tweet , tweet_attr)
13                       for one_tweet in hassan_tweets_resp.data]
14 ## step 4: transform into a dataframe
15 senator_tweets_df = pd.DataFrame(hassan_tweets_list ,
16                                  columns = tweet_attr)

```

- ▶ Same process for numeric id as previous section (get_user)
- ▶ Use get_user_tweets to get tweets from that user and pull_attr function defined previous to pull attributes

Activity 3: practice with the Twitter API

Notebook: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/08_apis_partII_twitter_blank.ipynb

1. Choose a public user (e.g., a politician; celebrity) and pull 100 tweets from their timeline and metadata about those tweets. When pulling metadata, make sure to get the `conversation_id` and count of replies (latter is in `public_metrics`)
2. Choose one of their tweets to focus on that got a lot of replies and get the `conversation_id` of that tweet
3. Paste the conversation id of that tweet into a query using this documentation for query building: <https://developer.twitter.com/en/docs/twitter-api/tweets/search/integrate/build-a-query#examples>
4. Similar to example 1.1 in the example code, use the `search_recent_tweets()` method to pull tweets that are in response to the focal tweet from step 2
5. Place the replies in a dataframe and do some text analysis of the results (eg sentiment; tokenizing and top words)